

EXPERT INSIGHT

Agile Model-Based Systems Engineering Cookbook

Improve system development by applying proven recipes for effective agile systems engineering

Foreword by:
Dr. Christian von Holst,
Global Tractor Systems Engineering Lead at John Deere

Second Edition



Dr. Bruce Powel Douglass

<packt>

Agile Model-Based Systems Engineering Cookbook

Second Edition

Improve system development by applying proven recipes for effective agile systems engineering

Dr. Bruce Powel Douglass

<packt>

BIRMINGHAM—MUMBAI

Agile Model-Based Systems Engineering Cookbook

Second Edition

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Senior Publishing Product Manager: Denim Pinto

Acquisition Editor – Peer Reviews: Saby Dsilva

Project Editor: Meenakshi Vijay

Content Development Editor: Rebecca Robinson

Copy Editor: Safis Editing

Technical Editor: Srishty Bhardwaj

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Rajesh Shirsath

First published: March 2021

Second edition: December 2022

Production reference: 2271222

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80323-582-0

www.packt.com

Foreword

As a systems engineer, you may know that viewpoints are important to understand context. When you read this foreword, you should be aware that I wrote it from my individual viewpoint. My own professional role involves the development of methods, templates, and training to make Systems engineering applicable for organizations, and also producing complex systems consisting of mechanical and electrical hardware, software, and the human beings operating it or interacting with it. Systems engineering is nowadays challenged with several constraints and boundary conditions requiring permanent change in the organization and its processes, methods, and tools. The need to work closely cross-domain (e. g. mechanical and hardware designers working together with software programmers, testers, and stakeholders), the need to quickly react to external (and internal) disruptions, and the need to bridge cross-cultural gaps require several significant changes in parallel. And this is difficult for bigger organizations and hard to realize. This is where this book helps significantly and contributes to mastering such challenges. The answer to those challenges lies in the introduction and application of **Systems Engineering (SE)** and in agile methods. The definition, deployment, implementation, maturation, and management of those two changes is a huge challenge to bigger companies and this book provides simple (not easy!) recipes to approach those changes.

This book perfectly fills a couple of gaps existing today in this area. Typically, there are two ways for companies to start their Systems Engineering journey: either they start first with training and educating Systems engineers theoretically or they start introducing a **Model-Based Systems Engineering (MBSE)** tool to provide a tool to “do Systems Engineering”. I’m not saying that the one or other method is better or worse, but I have observed this for years. Actually, both are necessary as, Systems Engineers need a good theoretical background on the methods and on tools. For re-use and communication, modern MBSE tools are optimally suited to implement SE formally in all development processes. However, to implement Systems Engineering broadly in a cross-domain development organization it is critical to follow strict modelling rules and to install syntactic and semantic guidelines.

In this area this book perfectly meets the needs of any SE practitioner and each company implementing or applying SE. The practical guidelines, the “step-by-step” descriptions, the cross-domain designed examples, and the provided downloads allow companies and individuals to get quickly and easily to the core of the method and mind set. With the provided references and additional sources, this book offers many further paths to go into more detail in many domains if needed or desired.

The second critical area covered by this book is the “agile” aspects of modern development organizations. The beauty of this book is that it holds “agile” at its core, the agile manifesto, and does not reduce or limit it to one of the many agile process implementations. The book provides excellent directions and practices to become more agile, independent of the development domain (not limited to software development), and also builds the bridge on how to use SE and MBSE in particular to foster an agile development process in general. Using the different SE artifacts as deliverables, prioritizing them accordingly to build a useful backlog, employing methods to track progress and risk, etc. are well described and practically applicable. As said before, practical examples and “step-by-step” guidelines are the core enablers for immediate practical application.

Last but not least, while the book follows the full development cycle and covers the aspects of functional analysis, architecting, trade studies, system design, verification, and validation, it also details the important aspects of system safety and security.

I highly recommend this book to all SE practitioners and to anyone considering how to implement SE in a company or how to become more agile. This second edition, with all of its enhancements and improvements in the step-by-step description and its additional areas covering things such as system security, is a must have in each SE’s library.

Thank you Bruce, and well done!

Christian von Holst

Contributors

About the author

Dr. Bruce Powel Douglass has received an MS in exercise physiology from the University of Oregon and a Ph.D. in neurocybernetics from the USD Medical School. He has worked as a software developer and systems engineer in safety-critical real-time embedded systems for almost 40 years and is a well-known speaker, author, and consultant in the area of real-time embedded systems, UML, and SysML. He is a coauthor of the UML and SysML standards, and teaches courses in real-time systems and software design and project management. Bruce has also authored articles for many journals and periodicals, especially in the real-time domain, and authored several other books on systems and software development. He has worked at I-Logix, Telelogic, and IBM on the Rhapsody modeling tool. He is currently a senior principal agile systems engineer at MITRE, and the principal at A-Priori Systems.

About the reviewers

Jaime Robles has more than a decade of experience in the development of complex engineered systems across the entire lifecycle. Currently, he is working as a Systems Engineer at the ALMA Observatory, the world's most powerful telescope at millimeter and submillimeter wavelengths. Previously, he has worked in the development of small space systems for planetary surface exploration and as a consultant in systems engineering with MBSE focus at SPEX Systems.

He is an aerospace engineer, an **OMG certified Systems Modeling Professional – Model Builder Advanced (OCSMP-MBA)**, an INCOSE associate Systems Engineering professional, and an active member of this professional organization participating in several groups (Space Systems WG, Requirements WG, LATAM Chapter). He also holds an MIT certificate in Architecture and Systems Engineering and has gained work experience in the United States, Switzerland, and Chile.

A word of acknowledgment to the author for sharing his vast knowledge with the MBSE community and for the opportunity to contribute with a grain of sand as a technical reviewer. Also, a special thanks to my family for their kind support and patience during the time invested in the review process.

Dr. Saulius Pavalkis is global MBSE Ecosystem Transformation Leader and MBSE R&D Cyber Portfolio Manager NAM at Dassault Systemes. He is also an INCOSE CAB Representative and on the CSE Board of Advisors at the University of Texas.

He has 20 years of MBSE solutions experience, and is a product owner and analyst with the Cameo core team, chief solution architect, consultant, and trainer. He is a world-leading expert in MBSE ecosystem, digital engineering, system architecture and simulation.

Throughout his career, he has been actively involved in building excellent MBSE ecosystem solutions in aerospace, defense, automotive and other areas as a former affiliate for JPL NASA for MBSE consulting and a contractor for Boeing's MBSE transformation.

Dr Saulius Pavalkis contributed to the *MBSE SysML Based Method and Framework MagicGrid* book and recently the *Agile MBSE Cookbook* by Dr. Bruce Douglass. He guides 2,000 subscribers—the largest SysML systems simulation community on YouTube (youtube.com/c/MBSEExecution). Saulius has INCOSE CSEP, OMG OCSMP, the No Magic lifetime modeling and simulation excellence award, a PhD in software engineering in the models query area, and a MS and BS in telecommunications and electronics from Kaunas University of Technology.

I would like to thank Dr. Bruce Douglass for amazing opportunity to review his latest book on Agile MBSE application. Dr. Bruce is a leading expert in effective and efficient MBSE application and has made this book is the best in class as an MBSE transformation guide. It is a must-have for any company or expert. Thank you, Dr. Bruce Douglass!

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>

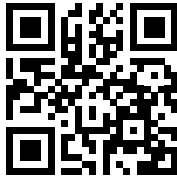


Table of Contents

Preface	xix
<hr/>	
Chapter 1: Basics of Agile Systems Modeling	1
<hr/>	
What's agile all about?	3
Incremental development • 3	
Continuous verification • 3	
Continuous integration • 4	
Avoid big design up front • 4	
Working with stakeholders • 4	
Model-Based Systems Engineering (MBSE)	4
Managing your backlog	6
Purpose • 8	
Inputs and preconditions • 8	
Outputs and postconditions • 8	
How to do it • 9	
Example • 13	
Measuring your success	19
How to do it • 21	
Example • 24	
Some considerations	27
Managing risk	28
Purpose • 29	

Inputs and preconditions • 29	
Outputs and postconditions • 29	
How to do it • 30	
Example • 33	
Product roadmap	38
Purpose • 39	
Inputs and preconditions • 39	
How to do it • 39	
Example • 43	
Release plan	46
Purpose • 46	
Inputs and preconditions • 46	
Outputs and postconditions • 46	
How to do it • 46	
Example • 48	
Iteration plan	51
Purpose • 51	
Inputs and preconditions • 51	
Outputs and postconditions • 52	
How to do it • 52	
Example • 55	
Estimating Effort	58
Purpose • 58	
Inputs and preconditions • 59	
Outputs and postconditions • 59	
How to do it • 59	
How it works • 59	
Example • 62	
Work item prioritization	66
Purpose • 67	
Inputs and preconditions • 67	

How to do it • 67	
How it works • 71	
Example • 72	
Iteration 0	78
Purpose • 78	
Inputs and preconditions • 78	
Outputs and postconditions • 79	
How to do it • 79	
Example • 83	
Architecture 0	85
Subsystem and component view • 85	
Concurrency and resource view • 85	
Distribution view • 85	
Dependability view • 86	
Deployment view • 86	
Purpose • 87	
Inputs and preconditions • 87	
Outputs and postconditions • 87	
How to do it • 87	
Example • 92	
Additional note • 100	
Organizing your models	101
Purpose • 101	
Inputs and preconditions • 101	
Outputs and postconditions • 101	
How to do it • 102	
How it works • 112	
Example • 112	
Managing change	113
Purpose • 114	
Inputs and preconditions • 115	

How to do it • 115

Example • 117

Chapter 2: System Specification 131

Recipes in this chapter 131

Why aren't textual requirements enough? 132

Definitions 133

Functional Analysis with Scenarios 134

Purpose • 134

Inputs and preconditions • 135

Outputs and postconditions • 135

How to do it • 135

Example • 138

Functional analysis with activities 153

Inputs and preconditions • 153

Outputs and postconditions • 153

How to do it • 153

Functional analysis with state machines 171

Purpose • 172

Inputs and preconditions • 172

Outputs and postconditions • 172

How to do it • 172

Example • 176

Functional Analysis with User Stories 185

A little bit about user stories • 185

Purpose • 189

Inputs and preconditions • 189

Outputs and postconditions • 189

How to do it • 189

Example • 191

Model-Based Safety Analysis	196
A little bit about safety analysis • 197	
Some Profiles • 198	
Hazard analysis	202
Purpose • 202	
Inputs and preconditions • 202	
Outputs and postconditions • 202	
How to do it • 202	
Example • 205	
Model-Based Threat Analysis	212
Basics of Cyber-Physical Security • 213	
Modeling for Security Analysis • 215	
Purpose • 220	
Inputs and preconditions • 220	
Outputs and postconditions • 220	
How to do it • 220	
Example • 223	
Specifying Logical System Interfaces	231
A Note about SysML Ports and Interfaces • 231	
Purpose • 235	
Inputs and preconditions • 235	
Outputs and postconditions • 235	
How to do it • 235	
Example • 237	
Creating the Logical Data Schema	246
Definitions • 246	
Example • 248	
Purpose • 251	
Inputs and preconditions • 251	
Outputs and postconditions • 251	

How to do it • 252

Example • 257

Chapter 3: Developing System Architectures 269

Recipes in this chapter 269

Five critical views of architecture 270

General architectural guidelines • 271

Architectural trade studies 272

Purpose • 272

Inputs and preconditions • 272

Outputs and postconditions • 273

How to do it • 274

Example • 279

Architectural merge • 286

Example • 290

Pattern-driven architecture • 299

Purpose • 301

Inputs and preconditions • 301

Outputs and postconditions • 301

How to do it • 301

Example • 303

Subsystem and component architecture • 310

Purpose • 312

Inputs and preconditions • 312

Outputs and postconditions • 313

How to do it • 313

Example • 315

Architectural allocation • 321

Creating subsystem interfaces from use case scenarios 334

Purpose • 335

Inputs and preconditions • 335

Outputs and postconditions • 335	
How to do it • 335	
Specializing a reference architecture	348
Purpose • 351	
Inputs and preconditions • 351	
Outputs and postconditions • 351	
How to do it • 352	
Chapter 4: Handoff to Downstream Engineering	363
Recipes in this chapter	363
Activities for the handoff to downstream engineering	364
Starting point for the examples • 365	
Preparation for Handoff • 372	
Federating Models for Handoff • 380	
Logical to Physical Interfaces • 389	
Deployment Architecture I: Allocation to Engineering Facets • 404	
Deployment Architecture II: Interdisciplinary Interfaces • 421	
Chapter 5: Demonstration of Meeting Needs: Verification and Validation	439
Recipes in this chapter	439
Verification and validation	441
Model simulation • 444	
Purpose • 445	
Inputs and preconditions • 445	
Outputs and postconditions • 445	
How to do it • 445	
Example • 448	
Model-based testing • 460	
Inputs and preconditions • 462	
Outputs and postconditions • 462	
How to do it • 463	

Example • 465	
Computable constraint modeling • 483	
Purpose • 483	
Inputs and preconditions • 484	
How to do it • 484	
Example • 486	
Traceability	493
Purpose • 497	
Inputs and preconditions • 497	
Outputs and postconditions • 498	
How to do it • 498	
Example • 502	
Effective Reviews and walkthroughs	504
Purpose • 505	
Inputs and preconditions • 506	
Outputs and postconditions • 506	
How to do it • 506	
Example • 510	
Managing Model Work Items	514
Purpose • 514	
Inputs and preconditions • 514	
How to do it • 515	
Example • 516	
Test Driven Modeling	519
Purpose • 520	
Inputs and preconditions • 520	
Outputs and postconditions • 520	
How to do it • 520	
Example • 522	

Appendix A: The Pegasus Bike Trainer	545
Overview	545
Pegasus High-Level Features	545
Highly customizable bike fit •	546
Monitor exercise metrics •	546
Export/upload exercise metrics •	546
Variable power output •	547
Gearing emulation •	547
Controllable power level •	547
Incline control •	547
User interface •	547
Online training system compatible •	548
Configuration and OTA firmware updates •	548
Other Books You May Enjoy	553
Index	557

Preface

Welcome to the *Agile Model-Based Systems Engineering Cookbook!* There is a plethora of published material for agile methods, provided that you want to create software. And the system is small. And the team is co-located. And it needn't be certified. Or safety-critical or high-reliability.

MBSE is none of these things. The output of MBSE isn't software implementation but system specification. It is usually applied to more complex and larger-scale systems. The teams are diverse and often spread out across departments and companies. Much of the time, the systems produced must be certified under various standards, including safety standards. So how do you apply agile methods to such an endeavor?

Most of the work in MBSE can be thought of as a set of workflows that produce a set of inter-related work products. Each of these workflows can be described with relatively simple recipes for creating the work products for MBSE including system requirements, systems architecture, system interfaces, and deployment architectures. That's what this book brings to the table and what sets it apart.

In this second edition, some new recipes have been added and all the examples and figures have been done using the Cameo Systems Modeler SysML tool.

Who this book is for

The book is, first and foremost, for systems engineers who need to produce work products for the specification of systems that include combinations of engineering disciplines, such as software, electronics, and mechanical engineering. More specifically, this book is about model-based systems engineering using the SysML language to capture, render, and organize the engineering data. Further, the book is especially about how to do all that in a way that achieves the benefits of agile methods – verifiably correct, adaptable, and maintainable systems. We assume basic understanding of the **Systems Modeling Language (SysML)** and at least some experience as a systems engineer.

What this book covers

Chapter 1, Basics of Agile Systems Modeling, discusses some fundamental agile concepts, expressed as recipes, such as managing your backlog, using metrics effectively, managing project risk, agile planning, work effort estimation and prioritization, starting up projects, creating an initial systems architecture, and organizing your systems engineering models. The recipes all take a systems engineering slant and focus on the work products commonly developed in a systems engineering effort.

Chapter 2, System Specification, is about agile model-based systems requirements – capturing, managing, and analyzing the system specification. One of the powerful tools that MBSE brings to the table is the ability to analyze requirements by developing computable and executable models. This chapter provides recipes for several different ways of doing that, as well as recipes for model-based safety and cyber-physical security analysis, and specifying details of information held within the system.

Chapter 3, Developing Systems Architecture, has recipes focused on the development of systems architectures. It begins with a way of doing model-based trade-studies (sometimes known as “analysis of alternatives”). The chapter goes on to provide recipes for integrating use case analyses into a systems architecture, applying architectural patterns, allocation of requirements into a systems architecture, and creating subsystem-level interfaces.

Chapter 4, Handoff to Downstream Engineering, examines one of the most commonly asked questions about MBSE: how to hand the information developed in the models off to implementation engineers specializing in software, electronics, or mechanical engineering. This chapter provides detailed recipes for getting ready to do the hand off, creating a federation of models to support the collaborative engineering effort to follow, converting the logical systems engineering interfaces to physical interface schemas, and actually doing the allocation to the engineering disciplines involved.

Chapter 5, Demonstration of Meeting Needs Verification and Validation, considers a key concept in agile methods: that one should never be more than minutes away from being able to demonstrate that, while the system may be incomplete, what’s there is correct. This chapter has recipes for model simulation, model-based testing, computable constraint modeling, adding traceability, how to run effective walkthroughs and reviews, and – my favorite – Test-driven modeling.

Appendix, The Pegasus Bike Trainer, details a case study that will serve as the basis for most of the examples in the book. This is a “smart” stationary bike trainer that interacts with net-based athletic training systems to allow athletes to train in a variety of flexible ways.

It contains aspects that will be implemented in mechanical, electronic, and software disciplines in an ideal exemplar for the recipes in the book.

To get the most out of this book

To get the most out of this book, you will need a solid, but basic, understanding of the **Systems Modeling Language (SysML)**. In addition, to create the models, you will need a modeling tool. The concepts here are expressed in SysML so any standards-compliant SysML modeling tool can be used.

All the example models in this book are developed using the Cameo Systems Modeler tool. To execute models and run simulations, you will need the Simulation Toolkit, included with the tool:

Software/hardware covered in the book	OS requirements
Cameo Systems Modeler	Windows, macOS, or Linux

Download the example models

You can download the example models for this book from the author's website at www.bruce-douglass.com. Note that these models are all in Cameo-specific format and won't generally be readable by other modeling tools.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Where to go from here

Visit the authors, website at www.bruce-douglass.com for papers, presentations, models, engineering forums, to download the models from this book, and more.

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://packt.link/av1n2>.

Conventions used

There are a number of text conventions used throughout this book.

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Select **System info** from the **Administration** panel.



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you reported this to us. Please visit <http://www.packtpub.com/submit-errata>, click **Submit Errata**, and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Share your thoughts

Once you've read *Agile Model-Based Systems Engineering Cookbook, Second Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there- you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781803235820>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

1

Basics of Agile Systems Modeling

For the most part, this book is about systems modeling with SysML, but doing it in an agile way. Before we get into the detailed practices of systems modeling with that focus, however, we're going to spend some time discussing important project-related agile practices that will serve as a backdrop for the modeling work.

Almost all of the agile literature focuses on the “three people in a garage developing a simple application” scope. The basic assumptions of such projects include:

- The end result is software that runs on a general-purpose computing platform (i.e., it is not embedded).
- Software is the only truly important work product. Others may be developed but they are of secondary concern. *Working software is the measure of success.*
- The software isn't performance, safety, reliability, or security-critical.
- It isn't necessary to meet regulatory standards.
- The development team is small and co-located.
- The development is time-and-effort, not fixed-price cost.
- The development is fundamentally code-based and not model- (or design)-based.
- Any developer can do any task (no specialized skills are necessary).
- Formalized requirements are not necessary.

Yes, of course, there is much made about extensions to agile practices to account for projects that don't exactly meet these criteria. For example, some authors will talk about a "scrum of scrums" as a way to scale up to larger teams. That works to a point, but it fails when you get to much larger development teams and projects. I want to be clear – I'm not saying that agile methods aren't application to projects that don't fall within these basic guidelines – only that the literature doesn't address how it will do so in a coherent, consistent fashion. The further away your project strays from these assumptions, the less you will find in the literature for agile ways to address your needs.

In this book, we'll address a domain that is significantly different than the prototypical agile project. Our concerns will be projects that:

- Are systems-oriented, which may contain software but will typically also contain electronic and mechanical aspects. *It's about the system and not the software.*
- Employ a **Model-Based Systems Engineering (MBSE)** approach using the SysML language.
- May range from small- to very large-scale.
- Must develop a number of different work products. These include, but are not limited to:
 - Requirements specification
 - Analysis of requirements, whether it is done with use case or user stories
 - System architectural specification
 - System interface specification
 - Trace relations between the elements of the different work products
 - Safety, reliability, and security (and resulting requirements) analyses
 - Architectural design trade studies
- Have a handoff to downstream engineering that includes interdisciplinary subsystem teams containing team members who specialize in software, electronics, mechanical, and other design aspects.

But at its core, the fundamental difference between this book and other agile books is that *the outcome of systems engineering isn't software, it's system specification*. Downstream engineering will ultimately do low-level design and implementation of those specifications. Systems engineering provides the road map that enables different engineers with different skill sets, working in different engineering disciplines, to collaborate together to create an integrated system, combining all their work into a cohesive whole.

The **International Council of Systems Engineering (INCOSE)** defines systems engineering as “a transdisciplinary and integrative approach to enable the successful realization, use, and retirement of engineered systems, using systems principles and concepts, and scientific, technological, and management methods” (<https://www.incose.org/about-systems-engineering/system-and-se-definition/systems-engineering-definition>). This book will not provide a big overarching process that ties all the workflows and work products together, although it is certainly based on one. That process – should you be interested in exploring it – is detailed in the author’s *Agile Systems Engineering* book; a detailed example is provided with the author’s *Harmony aMBSE Deskbook*, available at www.bruce-douglass.com. Of course, these recipes will work with any other reasonable MBSE process. It is important to remember that:



The outcome of software development is implementation;

The outcome of systems engineering is specification.

What’s agile all about?

Agile methods are – first and foremost – a means for improving the quality of your engineering work products. This is achieved through the application of a number of practices meant to continuously identify quality issues and immediately address them. Secondly, agile is about improving engineering efficiency and reducing rework. Let’s talk about some basic concepts of agility.

Incremental development

This is a key aspect of agile development. Take a big problem and develop it as a series of small increments, each of which is verified to be correct (even if incomplete).

Continuous verification

The best way to have high-quality work products is to continuously develop and verify their quality. In other books, such as *Real-Time Agility* or the aforementioned *Agile Systems Engineering* books, I talk about how verification takes place in three timeframes:

- Nanocycle: 30 minutes to 1 day
- Microcycle: 1–4 weeks
- Macrocycle: Project length

Further, this verification is best done via the execution and testing of computable models. We will see in later chapters how this can be accomplished.

Continuous integration

Few non-trivial systems are created by a single person. Integration is the task of putting together work products from different engineers into a coherent whole and demonstrating that, as a unit, it achieves its desired purpose. This integration is often done daily, but some teams increment this truly continuously, absorbing work as engineers complete it and instantly verifying that it works in tandem with the other bits.

Avoid big design up front

The concept of incremental development means that one thing that we *don't* do is develop big work products over long periods of time and only then try to demonstrate their correctness. Instead, we develop and verify the design work we need right now, and defer design work that we won't need until later. This simplifies the verification work and also means much less rework later in a project.

Working with stakeholders

A key focus of the Agilista is the needs of the stakeholders. The Agilista understands that there is an “air gap” between what the requirements say and what the stakeholder actually needs. By working with the stakeholder, and frequently offering them versions of the running system to try, they are more likely to actually meet their needs. Additionally, user stories – a way to organize requirements into short usage stakeholder-system usage scenarios – are a way to work with the stakeholder to understand what they actually need.

Model-Based Systems Engineering (MBSE)

Systems engineering is an independent engineering discipline that focuses on system properties – including functionality, structure, performance, safety, reliability, and security. MBSE is a model-centric approach to performing systems engineering. Systems engineering is largely independent of the engineering disciplines used to implement these properties. Systems engineering is an interdisciplinary activity that focuses more on this integrated set of system properties than on the contributions of the individual engineering disciplines. It is an approach to developing complex and technologically diverse systems. Although normally thought of in a V-style process approach (see *Figure 1.1*), the “left side of the V” emphasizes the specification of the system properties (requirements, architecture, interfaces, and overall dependability), the “lower part of the V” has to do with the discipline-specific engineering and design work, and the “right side of the V” has to do with the verification of the system against the specifications developed on the left side:

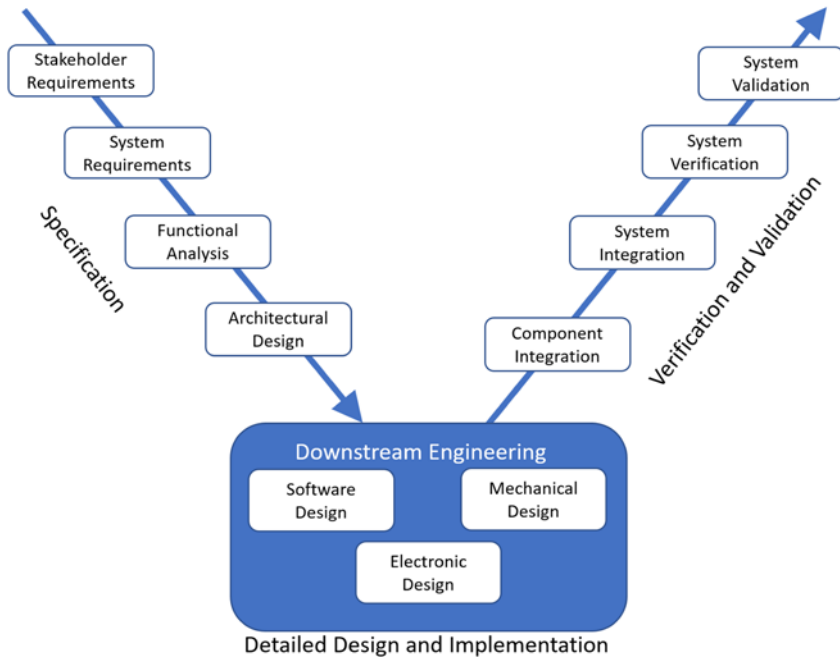


Figure 1.1: Standard V model life cycle

Of course, we'll be doing things in a more agile way (Figure 1.2). Mostly, we'll focus on incrementally creating the specification work products and handing them off to downstream engineering in an agile way:

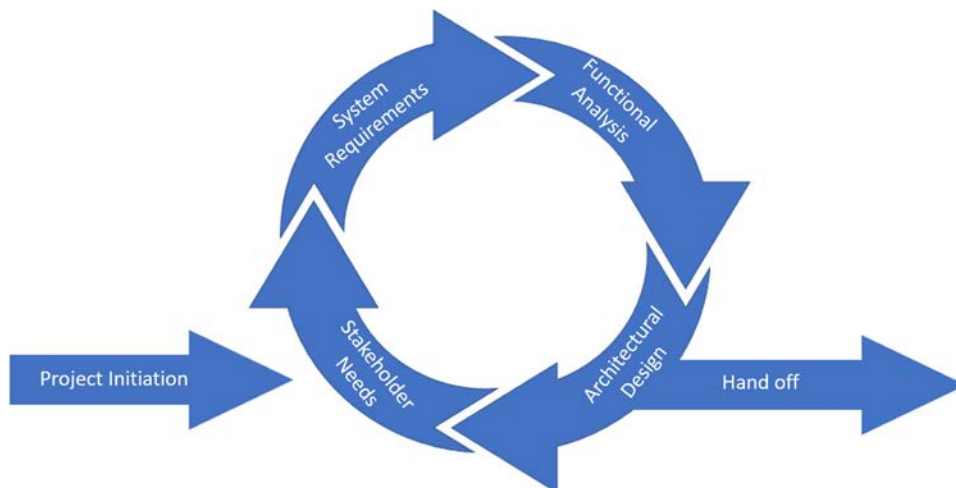


Figure 1.2: Basic Agile systems engineering workflow

The basis of most of the work products developed in MBSE is, naturally enough, *the model*. For the most part, this refers to the set of engineering data relevant to the system captured in a SysML model. The main model is likely to be supplemented with models in other languages, such as performance, safety, and reliability (although you can use SysML for that too – we’ll discuss that in *Chapter 2, System Specification—Functional, Safety and Security Analysis*). The other primary work product will be *textual requirements*. While they are imprecise, vague, ambiguous, and hard to verify, they have the advantage of being easy to communicate. Our models will cluster these requirements into usage chunks – epics, use cases, and user stories – but we’ll still need requirements. These may be managed either as text or in text-based requirements management tools, such as IBM DOORS™, or they can be managed as model elements within a SysML specification model.

Our models will consist of formal representations of our engineering data as *model elements* and the relationships among them. These elements may appear in one or more views, including diagrams, tables, or matrices. The model is, then, a coherent collection of model elements that represent the important engineering data around our system of interest.

In this book, we assume you already know SysML. If you don’t, there are many books around for that. This book is a collection of short, high-focused workflows that create one or a small set of engineering work products that contain relevant model elements.

Now, let’s talk about some basic agile recipes and how they can be done in a model-centric environment.

Managing your backlog

The **backlog** is a prioritized set of **work items** that identify work to be done. There are generally two such backlogs. The **project backlog** is a prioritized list of all work to be done in the current project. A subset of these is selected for the current increment, forming the **iteration backlog**. Since engineers usually work on the tasks relevant to the current iteration, that is where they will go to get their tasks. *Figure 1.3* shows the basic idea of backlogs:

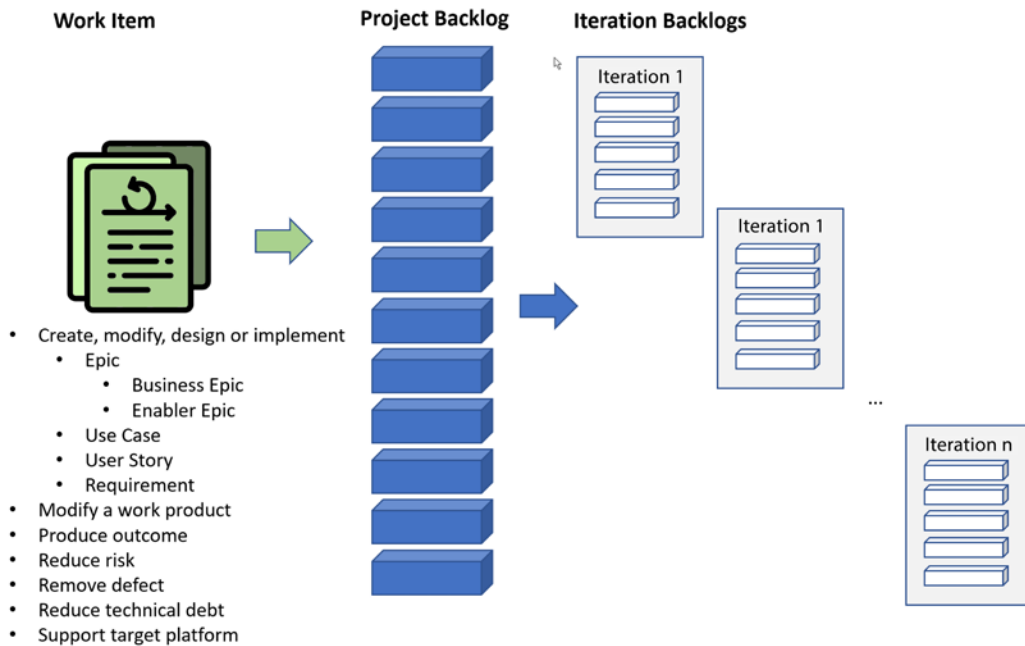


Figure 1.3: Backlogs

The work to be done, nominally referred to as **work items**, is identified. Work items can be application work items (producing work that will be directly delivered) or technical work items (doing work that enables technical aspects of the product or project). Work items identify work to do such as:

- Analyzing, designing, or implementing an epic, use case, or user story, to ensure a solid understanding of the need and the adequacy of its requirements
- Creating or modifying a work product, such as a requirements specification or a safety analysis
- Arranging for an outcome, such as certification approval
- Addressing a risk, such as determining the adequacy of the bus bandwidth
- Removing an identified defect
- Supporting a target platform, such as an increment with hand-built mechanical parts, lab-constructed wire wrap boards, and partial software

The work items go through an acceptance process, and if approved, are put into the project backlog. Once there, they can be allocated to an iteration backlog.

Purpose

The purpose of managing your backlog is to provide clear direction for the engineering activities, to push the project forward in a coherent, collaborative way.

Inputs and preconditions

The inputs are the work items. The functionality-based work items originate with one or more stakeholders, but other work items might come from discovery, planning, or analysis.

Outputs and postconditions

The primary outputs are the managed project and iteration backlogs. Each backlog consists of a set of work items around a common purpose, or *mission*. The mission of an iteration is the set of work products and outcomes desired at the end of the iteration. An iteration mission is defined as shown in *Figure 1.4*:

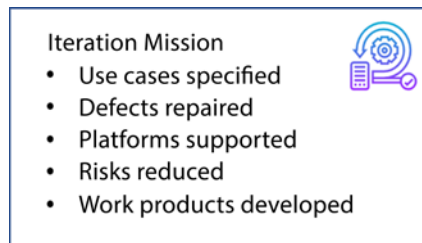


Figure 1.4: Iteration mission

In a modeling tool, this information can be captured as metadata associated with tags.



The term “metadata” literally means “data about data”; in this context, we add metadata to elements using tags.

How to do it

There are two workflows to this recipe. The first, shown in *Figure 1.5*, adds a work item to the backlog. The second, shown in *Figure 1.6*, removes it:

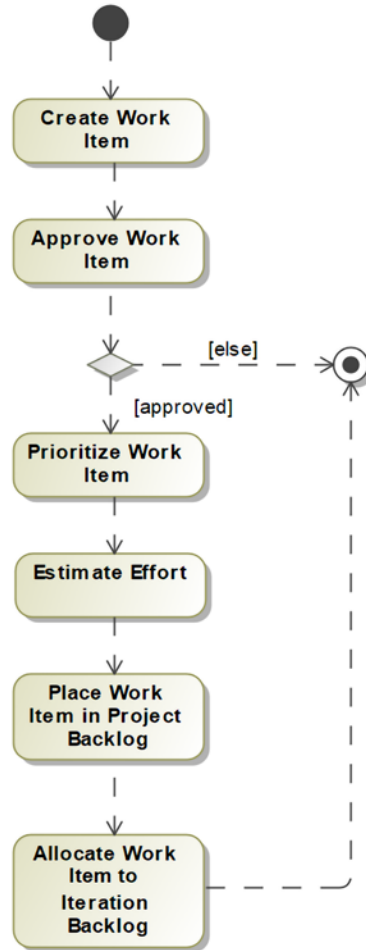


Figure 1.5: Add work item

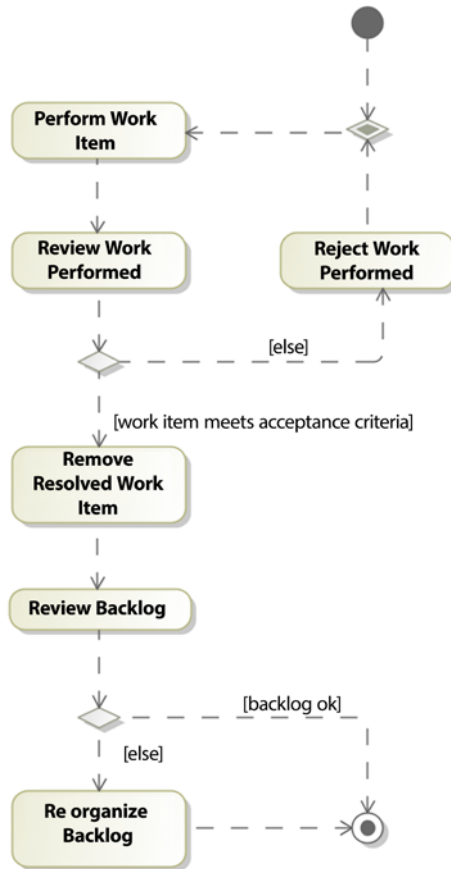


Figure 1.6: Resolve work item

Create a workflow item

From the work to be done, a work item is created to put into the backlog. The work item should include the properties shown in *Figure 1.7*:

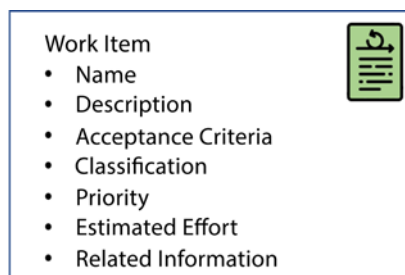


Figure 1.7: Work item

- Name.
- Description of the work to be done, the work product to be created, or the risk to be addressed.
- The acceptance criteria – how the adequacy of the work performed, the work product created, or the outcome produced will be determined.
- The work item classification identifies the kind of work item it is, as shown on the left side of *Figure 1.3*.
- The work item’s priority is an indication of how soon this work item should be addressed. This is discussed in the *Prioritize work item* step of this recipe.
- The estimated effort is how much effort it will take to perform the task. This can be stated in absolute terms (such as hours) or relative terms (such as user story points). This topic is addressed in the *Estimating effort* recipe later in this chapter.
- Links to important related information, such as standards that must be met, or sources of information that will be helpful for the performance of the work.

Approve work item

Before a work item can be added, it should be approved by the team or the project leader, whoever is granted that responsibility.

Prioritize work item

The priority of a work item determines in what iteration the work will be performed. Priority is determined by a number of factors, including the work item’s criticality (how important it is), its urgency (when it is needed), the availability of specialized resources needed to perform it, usefulness to the mission of the iteration, and risk. The general rule is that high-priority tasks are performed before lower-priority tasks. This topic is covered in the *Work item prioritization recipe* later in this chapter.

Estimate effort

An initial estimate of the cost of addressing the work item is important because as work items are allocated to iterations, the overall effort budget must be balanced. If the effort to address a work item is too high, it may not be possible to complete it in the iteration with all of its other work items. The agile practice of work item estimation is covered in the *Estimating effort* recipe later in this chapter.

Place work item in project backlog

Once approved and characterized, the work item can then be put into the project backlog. The backlog is priority-ordered so that higher-priority work items are “on top” and lower-priority work items are “below”.

Allocate work item to iteration backlog

Initial planning includes the definition of a planned set of iterations, each of which has a mission, as defined above. Consistent with that mission, work items are then allocated to the planned iterations. Of course, this plan is volatile, and later work or information can cause replanning and a reallocation of work items to iterations. Iteration planning is the topic of the *reIteration plan* recipe later in this chapter.

In the second work flow of this recipe, the work is actually being done. Of relevance here is how the completion of the work affects the backlog (*Figure 1.6*).

Perform work item

This action is where the team member actually performs the work to address the work item, whether it is to analyze a use case, create a bit of architecture, or perform a safety analysis.

Review work performed

The output and/or outcome of the work item is evaluated with respect to its acceptance criteria and is accepted or rejected on that basis.

Reject work performed

If the output and/or outcome does not meet the acceptance criteria, the work is rejected and the work item remains on the backlog.

Remove resolved work item

If the output and/or outcome does meet the acceptance criteria, the work is accepted and the work item is removed from the project and iteration to-do backlog. This usually means that it is moved to a “to-done” backlog, so that there is a history of the work performed.

Review backlog

It is important that as work progresses, the backlog is maintained. Often, valuable information is discovered that affects work item effort, priority, or value during project work. When this occurs, other affected work items must be reassessed and their location within the backlogs may be adjusted.

Reorganize backlog

Based on the review of the work items in the backlog, the set of work items, and their prioritized positions within those backlogs, may require adjustment.

Example

Consider a couple of use cases for the sample problem, the Pegasus Bike Trainer summarized in *Appendix A* (see *Figure 1.8*):

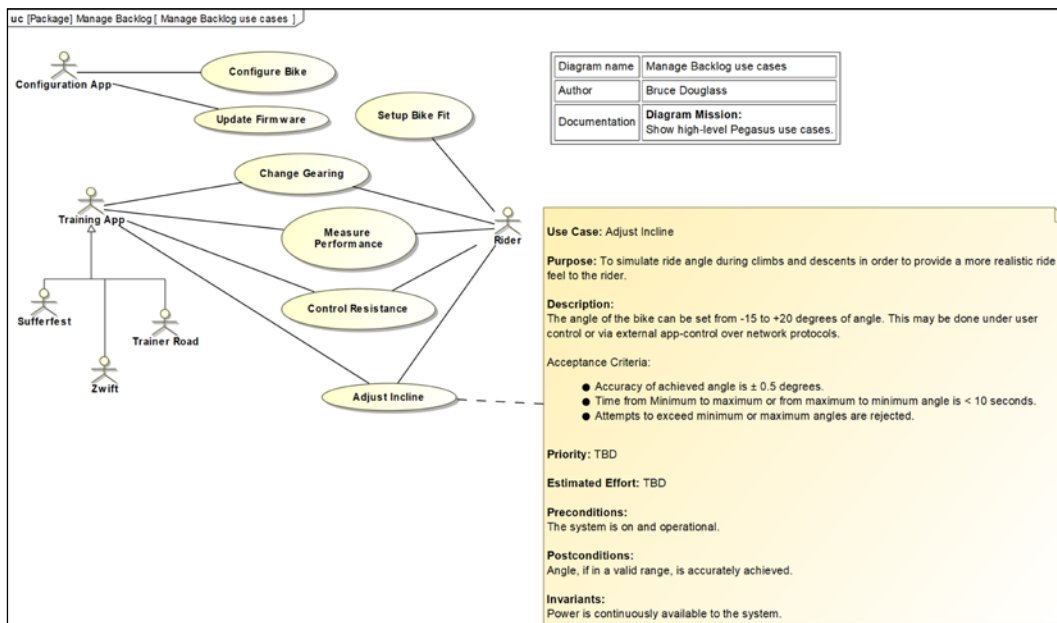


Figure 1.8: Example user case work items in backlog

You can also show at least high-level backlog allocation to an iteration on a use case diagram, as shown in *Figure 1.9*. You may, of course, manage backlogs in generic agile tools such as Rational Team Concert, Jira, or even with Post-It notes:

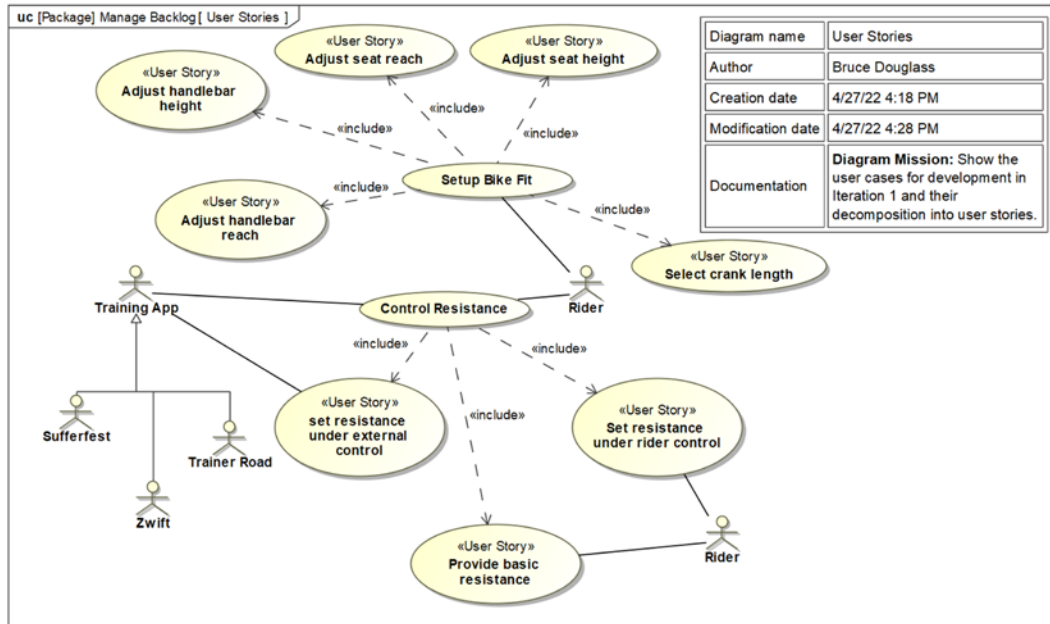


Figure 1.9: Use case diagram for iteration backlog

Let's apply the workflow shown in *Figure 1.5* to add the use cases and user stories from *Figure 1.8* and *Figure 1.9*.

Create work item

In *Figure 1.8* and *Figure 1.9*, we see a total of seven use cases and eight user stories. For our purpose, we will just represent the use case data in tabular form and will concentrate only on the two use cases and their contained user stories from *Figure 1.9*. The description of the user stories is provided in the canonical form of a user story (see the chapter introduction in *Chapter 2, System Specification: Functional, Safety, and Security Analysis* for more details).

#	Name	Documentation	Applied Stereotype	approved	acceptance criteria	priority	effort	iteration	related information
1	Setup Bike Fit	Enable rider to adjust bike fit prior to ride	-- work item [Element]	<input checked="" type="checkbox"/>	Standard riders* can replicate their road bike fit on the Pegasus.				*Standard riders in include 5 riders of height 60, 65, 70, 75, and 76 inches.
2	Adjust handlebar reach	As a rider, I want to replicate the handlebar reach on my fitted road	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their handlebar reach from their fitted road bikes.				
3	Adjust handlebar height	As a rider, I want to replicate the handlebar height on my fitted road	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their handlebar height from their fitted road bikes.				
4	Adjust seat reach	As a rider, I want to replicate the seat reach on my fitted road bike	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their seat reach from their fitted road bikes.				
5	Adjust seat height	As a rider, I want to replicate the seat height on my fitted road bike.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their seat height from their fitted road bikes.				
6	Select crank length	As a rider, I want to replicate the crank arm length on my road bike	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Support crank lengths of 165, 167.5, 170, 172.5, and 175 mm crank lengths.				
7	Control Resistance	Control the resistance to pedaling in a steady and well-controlled fashion within the limits of normal terrain road riding.	-- work item [Element]	<input type="checkbox"/>	Replicate pedal resistance to within 1% of measured pedal torque under the standard ride set*.				*Standard ride set includes ride of all combination of rider weights (50, 75, 100kg), inclines (-10, 0, 5, 10, 20%) and cadences (50, 70, 80, 90, 110).
8	Provide basic resistance	As a rider, I want basic resistance provided to the pedals so I can get a workout with an on-road feel in Resistance Mode.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance by setting the pedal resistance to 0 -2000W in 50 watt increments for standard ride set*.				
9	Set resistance under rider control	As a rider, I want to set the resistance level provided to the pedals to increase or decrease the effort for a given gearing, cadence and incline.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance via user input by manually setting incline, gearing, and cadence for standard ride set*.				
10	set resistance under external control	As a rider, I want the external training app to set the resistance to follow the app's workout protocol to get the desired workout.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance via app control manually setting incline, gearing, and allow the user to supply cadence for standard ride set*.				

Figure 1.10: Initial work item list

For the work item list, I created a stereotype **work item** that has the tag definitions shown as columns in the table and then applied it to the use cases and user stories.

Approve work item

#	Name	Documentation	Applied Stereotype	approved	acceptance criteria	priority	effort	iteration	related information
1	Setup Bike Fit	Enable rider to adjust bike fit prior to ride	-- work item [Element]	<input checked="" type="checkbox"/>	Standard riders* can replicate their road bike fit on the Pegasus.				*Standard riders in include 5 riders of height 60, 65, 70, 75, and 76 inches.
2	Adjust handlebar reach	As a rider, I want to replicate the handlebar reach on my fitted road	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their handlebar reach from their fitted road bikes.				
3	Adjust handlebar height	As a rider, I want to replicate the handlebar height on my fitted road	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their handlebar height from their fitted road bikes.				
4	Adjust seat reach	As a rider, I want to replicate the seat reach on my fitted road bike	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their seat reach from their fitted road bikes.				
5	Adjust seat height	As a rider, I want to replicate the seat height on my fitted road bike.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Standard riders* can replicate their seat height from their fitted road bikes.				
6	Select crank length	As a rider, I want to replicate the crank arm length on my road bike	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Support crank lengths of 165, 167.5, 170, 172.5, and 175 mm crank lengths.				
7	Control Resistance	Control the resistance to pedaling in a steady and well-controlled fashion within the limits of normal terrain road riding.	-- work item [Element]	<input type="checkbox"/>	Replicate pedal resistance to within 1% of measured pedal torque under the standard ride set*.				*Standard ride set includes ride of all combination of rider weights (50, 75, 100kg), inclines (-10, 0, 5, 10, 20%) and cadences (50, 70, 80, 90, 110).
8	Provide basic resistance	As a rider, I want basic resistance provided to the pedals so I can get a workout with an on-road feel in Resistance Mode.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance by setting the pedal resistance to 0 -2000W in 50 watt increments for standard ride set*.				
9	Set resistance under rider control	As a rider, I want to set the resistance level provided to the pedals to increase or decrease the effort for a given gearing, cadence and incline.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance via user input by manually setting incline, gearing, and cadence for standard ride set*.				
10	set resistance under external control	As a rider, I want the external training app to set the resistance to follow the app's workout protocol to get the desired workout.	-- User Story [UseCase] -- work item [Element]	<input type="checkbox"/>	Control resistance via app control manually setting incline, gearing, and allow the user to supply cadence for standard ride set*.				

Figure 1.11: Working with the team and the stakeholders, we get approval for the work items in

As we get approval, we marked the **Approved** column in the table.

Prioritize work item

Using the techniques from the *Work item prioritization* recipe later in this chapter, we add the priorities to the work items.

Estimate effort

Using the techniques from the *Estimating effort* recipe later in this chapter, we add the estimated effort to the work items.

Our final set of work items from this effort is shown in *Table 1.1*:

Name	OK	Description	Acceptance	Classification	Priority	Effort	Iteration	Related
Setup bike fit		Enable rider to adjust bike fit prior to ride	Standard riders* can replicate their road bike fit on the Pegasus.	Use Case	4.38	13		*Standard riders include five riders of heights 60, 65, 70, 75, and 76 inches.
Adjust handlebar reach		As a rider, I want to replicate the handlebar reach on my fitted road bike.	Standard riders* can replicate their handlebar reach from their fitted road bikes.	User Story	3.33	3		
Adjust handlebar height		As a rider, I want to replicate the handlebar height on my fitted road bike.	Standard riders* can replicate their handlebar height from their fitted road bikes.	User Story	4.33	3		
Adjust seat reach		As a rider, I want to replicate the seat reach on my fitted road bike.	Standard riders* can replicate their seat reach from their fitted road bikes.	User Story	11.67	3		
Adjust seat height		As a rider, I want to replicate the seat height on my fitted road bike.	Standard riders* can replicate their seat height from their fitted road bikes.	User Story	13.33	3		

Select crank length	As a rider, I want to replicate the crank arm length on my road bike.	Support crank lengths of 165, 167.5, 170, 172.5, and 175 mm.	User Story	1.2	1		
Control resistance	Control the resistance to pedaling in a steady and well-controlled fashion within the limits of normal terrain road riding.	Replicate pedal resistance to within 1% of measured pedal torque under the standard ride set*.	Use Case	2	115		*Standard ride set includes ride of all combination of rider weights (50, 75, and 100kg), inclines (-10, 0, 5, 10, and 20%) and cadences (50, 70, 80, 90, and 110).
Provide basic resistance	As a rider, I want basic resistance provided to the pedals so I can get a workout with an on-road feel in Resistance Mode.	Control resistance by setting the pedal resistance to 0–2000W in 50-watt increments for the standard ride set.*	User Story	1.42	55		

Set resistance under user control	As a rider, I want to set the resistance level provided to the pedals to increase or decrease the effort for a given gearing, cadence, and incline.	Control resistance via user input by manually setting incline, gearing, and cadence for the standard ride set.*	User Story	1.00	21		
Set resistance under external control	As a rider, I want the external training app to set the resistance to follow the app's workout protocol to get the desired workout.	Control resistance via app control, manually setting incline, gearing, and allow the user to supply cadence for the standard ride set.*	User Story	0.30	39		

Table 1.1: Final work item list

Place WI in project backlog

As we complete the effort, we put all the approved work items into the project backlog, along with other previously identified use cases, user stories, technical work items, and spikes. The backlog can be managed within the modeling tool, but usually external tools – such as Jira or Team Concert – are used.

Allocate WI to iteration backlog

Using the technique from the *Iteration plan* recipe later in this chapter, we put relevant work items from the project backlog into the backlog for the upcoming iteration. In *Table 1.1*, this would be done by filling in the **Iteration** column with the number of the iteration in which the work item is performed.

With regard to the second workflow from *Figure 1.6*, we can illustrate how the workflow might unfold as we perform the work in the current iteration.

Perform work item

As we work in the iterations, we detail the requirements, and create and implement the technical design. For example, we might perform the mechanical design of the handlebar reach adjust or the delivery of basic resistance to the pedals with an electric motor.

Review work performed

As the work on the use case and user stories completes, we apply the acceptance criteria via *verification testing and validation*. In the example we are considering, for the set of riders of heights 60, 65, 70, 75, and 76 inches, we would measure the handlebar height from their fitted road bikes and ensure that all these conditions can be replicated on the bike. For the **Provide Basic Resistance** user story, we would verify that we can create a pedal resistance of [0, 50, 100, 150, ... 2000] watts of resistance at pedal cadences of 50, 70, 80, 90, and 110 RPM \pm 1%.

Measuring your success

One of the core concepts of effective agile methods is to continuously improve how you perform your work. This can be done to improve quality or to get something done more quickly. In order to improve how you work, you need to know how well you're doing *now*. That means applying metrics to identify opportunities for improvement and then changing what you do or how you do it. Metrics are a general measurement of success in either achieving business goals or compliance with a standard or process. A related concept – a **Key Performance Indicator (KPI)** – is a quantifiable measurement of accomplishment against a crucial goal or objective. The best KPIs measure achievement of goals rather than compliance with a plan. The problem with metrics is that they measure something that you believe correlates to your objective, but not the objective itself. Some examples from software development:

Objective	Metric	Issues
Software size	Lines of code	Lines of code for simple, linear software aren't really the same as lines of code for complex algorithms
Productivity	Shipping velocity	Ignores the complexity of the shipped features, penalizing systems that address complex problems
Accurate planning	Compliance with schedule	This metric rewards people who comply with even a <i>bad</i> plan
Efficiency	Cost per defect	Penalizes quality and makes buggy software look cheap

Quality	Defect density	Treats all defects the same whether they are using the wrong-sized font or something that brings aircraft down
---------	----------------	--

Table 1.2: Examples from software development



See *The Mess of Metrics* by Capers Jones (2017) at <http://namcook.com/articles/The%20Mess%20of%20Software%20Metrics%202017.pdf>

Consider a common metric for high-quality design, cyclomatic complexity. It has been observed that highly complex designs contain more defects than designs of low complexity. Cyclomatic complexity is a software metric that computes complexity by counting the number of linearly independent paths through some unit of software. Some companies have gone so far as to require all software to not exceed some arbitrary cyclomatic complexity value to be considered acceptable. This approach disregards the fact that some problems are harder than others and any design addressing such problems must be more complex. A better application of cyclomatic complexity is to use the metric as a *guide*. It can identify those portions of a design that are more complex so that they can be subjected to additional testing. Ultimately, the problem with this metric is that complexity correlates only loosely to quality. A better metric for the goal of improving quality might be the ability to successfully pass tests that traverse all possible paths of the software.

Good metrics are easy to measure, and, ideally, easy to automate. Creating test cases for all possible paths can be tedious, but it is possible to automate with appropriate tools. Metrics that require additional work by engineering staff will be resented and achieving compliance with the use of the metric may be difficult.

While coming up with good metrics may be difficult, the fact remains that *you can't improve what you don't measure*. Without measurements, you're guessing where problems are and your solutions are likely to be ineffective or solve the wrong problem. By measuring how you're doing against your goals, you can improve your team's effectiveness and your product quality. However, it is important that metrics are used as indicators rather than as performance standards because, ultimately, the world is more complex than a single, easily computed measure.



Metrics should be used for guidance, not as goals for strict compliance.

Purpose

The purpose of metrics is to measure, rather than guess, how your work is proceeding with respect to important qualities so that you can improve.

Inputs and preconditions

The only preconditions for this workflow are the desire, ability, and authority to improve.

Outputs and postconditions

The primary output of this recipe is objective measurements of how well your work is proceeding or the quality of one or more work products. The primary outcome is the identification of some aspect of your project work to improve.

How to do it

Metrics can be applied to any work activity for which there is an important output or outcome (which should really be all work activities). The workflow is fairly straightforward, as shown in *Figure 1.11*:

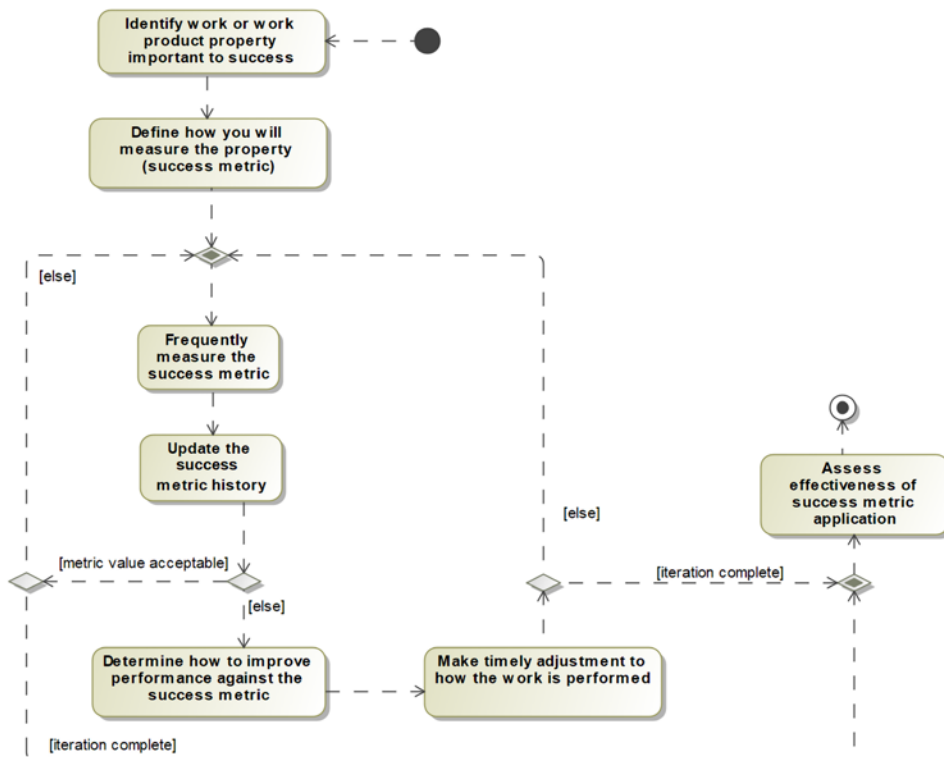


Figure 1.12: Measuring success

Identify work or work product property important to success

One way to identify a property of interest is to look where your projects have problems or where the output work products fail. For engineering projects, work efficiency being too low is a common problem. For work products, the most common problem is the presence of defects.

Define how you will measure the property (success metric)

Just as important to identifying what you want to measure is coming up with a quantifiable measurement that is simultaneously easy to apply, easy to measure, easy to automate, and accurately captures the property of interest. It's one thing to say "the system should be fast" but quite another to define a way to measure the speed in a fashion that can be compared to other work items and iterations.

Frequently measure the success metric

It is common to gather metrics for a review at the end of a project. This review is commonly called a *project post-mortem*. I prefer to do frequent retrospectives, at least one per iteration, which I refer to as a *celebration of ongoing success*. To be applied in a timely way, you must measure frequently. This means that the measurements must require low effort and be quick to compute. In the best case, the environment or tool can automate the gathering and analysis of the information without any ongoing effort by the engineering staff. For example, time spent on work items can be captured automatically by tools that check out and check in work products.

Update the success metric history

For long-term organizational success, recorded performance history is crucial. I've seen far too many organizations miss their project schedules by 100% or more, only to do the very same thing on the next project, and for exactly the same reasons. A metric history allows the identification of longer-term trends and improvements. That enables the reinforcement of positive aspects and the discarding of approaches that fail.

Determine how to improve performance against the success metric

If the metric result is unacceptable, then you must perform a root cause analysis to uncover what can be done to improve it. If you discover that you have too many defects in your requirements, for example, you may consider changing how requirements are identified, captured, represented, analyzed, or assessed.

Make timely adjustments to how the activity is performed

Just as important to measuring how you're doing against your project and organizational goals is acting on that information. This may be changing a project schedule to be more accurate, performing more testing, creating some process automation, or even getting training on some technology.

Assess the effectiveness of the success metric application

Every so often, it is important to look at whether applying a metric is generating project value. A common place to do this is the *project retrospective* held at the end of each iteration. Metrics that are adding insufficient value may be dropped or replaced with other metrics that will add more value.

Some commonly applied metrics are shown in *Figure 1.13*:

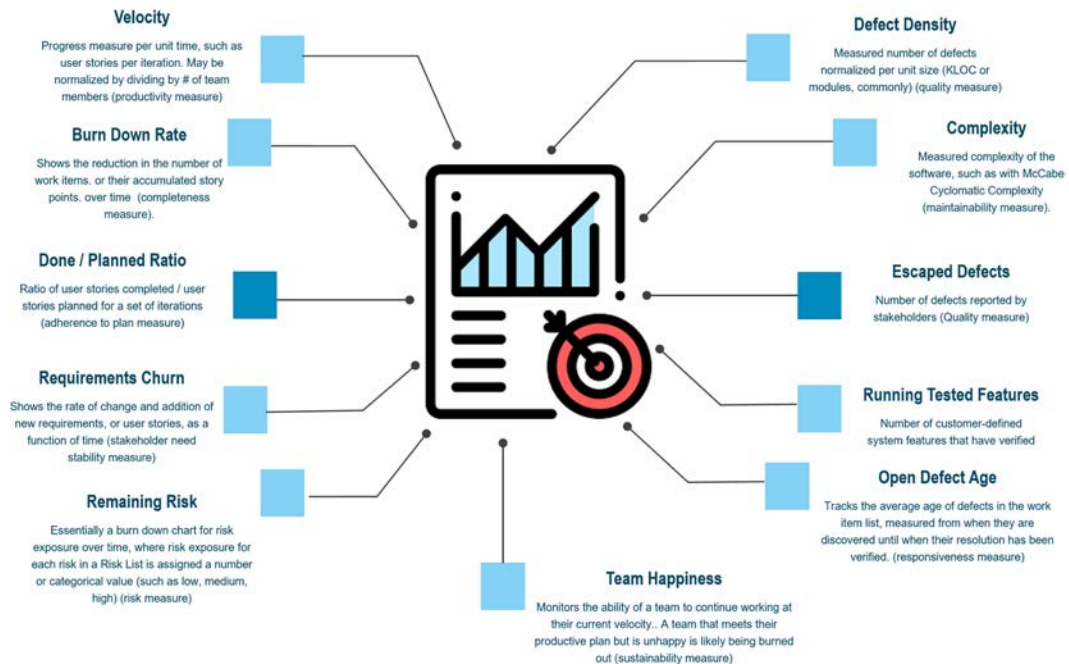



Figure 1.13: Some common success metrics

It all comes back to *you can't improve what you don't measure*. First, you must understand how well you are achieving your goals now. Then you must decide how you can improve and make the adjustment. Repeat. It's a simple idea.

Visualizing velocity is often done as a velocity or burn down chart. The former shows the planned velocity in work items per unit time, such as use cases or user stories per iteration. The latter shows the rate of progress of handling the work items over time. It is common to show both planned values in addition to actual values. A typical velocity chart is shown in *Figure 1.14*.



Velocity is the amount of work done per time unit, such as the number of user stories implemented per iteration. A burn down chart is a graph showing the decreasing number of work items during a project.

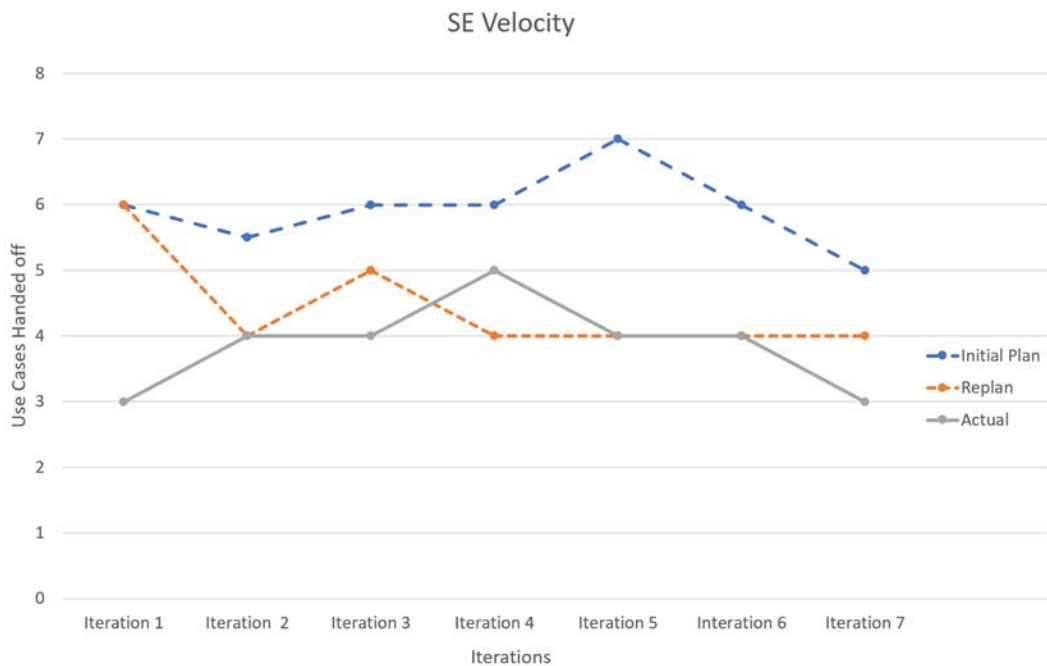


Figure 1.14: Velocity chart

Example

Let's look at an example of the use of metrics in our project:

Identify work or work product property important to success

Let's consider a common metric used in agile software development and apply them to systems engineering: *velocity*. Velocity underpins all schedules because it represents how much functionality is delivered per unit time. Velocity is generally measured as the number of completed user stories delivered per iteration. In our scope, we are not delivering implemented functionality, but we are incrementally delivering a hand-off to downstream engineering. Let's call this **SE Velocity**, which is "specified use cases per iteration" and includes the requirements and all related SE work products.

This might not provide the granularity we desire, so let's also define a second metric, **SE Fine-Grained Velocity**, which is the number of story points specified in the iteration:

Define how you will measure the property (success metric)

We will measure the number of use cases delivered, but have to have a "definition of done" to ensure consistency of measurement. SE Velocity will include:

- Use case with:
 - Full description identifying purpose, pre-conditions, post-conditions, and invariants.
 - Normative behavioral specification in which all requirements traced to and from the use case are represented in the behavior. This is a "minimal spanning set" of scenarios in which all paths in the normative behavior are represented in at least one scenario
- Trace links to all related functional requirements and quality of service (performance, safety, reliability, security, etc) requirements
- Architecture into which the implementation of the use cases and user stories will be placed
- System interfaces with a physical data schema to support the necessary interactions of the use cases and user stories
- Logical test cases to verify the use cases and user stories
- Logical validation cases to ensure the implementation of the use cases and user stories meets the stakeholder needs

SE Velocity will be simply the number of such use cases delivered per iteration. SE Fine-Grained Velocity will be the estimated effort (as measured in story points; see the *Estimating effort* recipe).

Frequently measure the success metric

We will measure this metric each iteration. If our project has 35 use cases, our project heartbeat is 4 weeks, and the project is expected to take one year, then our SE Velocity should be $35/12$ or about 3. If the average use case is 37 story points, then our SE Fine-Grained Velocity should be about 108 story points per iteration.

Update the success metric history

As we run the project, we will get measured SE Velocity and SE Fine-Grained Velocity. We can plot those values over time to get velocity charts:



Figure 1.15: SE velocity charts

Determine how to improve performance against the success metric

Our plan calls for 3 use cases and 108 story points per iteration; we can see that we are underperforming. This could be either because 1) we overestimated the planned velocity, or 2) we need to improve our work efficiency in some way. We can, therefore, simultaneously attack the problem on both fronts.

To start, we should replan based on our measured velocity, which is averaging 2.25 use cases and 81 story points per iteration, as compared to the planned 3 use cases and 108 story points. This will result in a longer but hopefully more realistic project plan and extend the planned project by an iteration or so.

In addition, we can analyze why the specification effort is taking too long and perhaps implement changes in process or tooling to improve.

Make timely adjustments to how the activity is performed

As we discover variance between our plan and our reality, we must adjust either the plan or how we work, or both. This should happen at least every iteration, as the metrics are gathered and analyzed. The iteration retrospective that takes place at the end of the iteration performs this service.

Assess the affectiveness of the success metric applicaiton

Lastly, are the metrics helping the project? It might be reasonable to conclude that the fine-grained metric provides more value than the more general SE Velocity metric, so we abandon the latter.

Some considerations

I have seen metrics fail in a number of organizations trying to improve. Broadly speaking, the reasons for failure are one of the following:

Measuring the wrong thing

Many qualities of interest are hard to identify precisely (think of “code smell”) or difficult to measure directly. Metrics are usually project qualities that are easy to measure but you can only imprecisely measure what you want. The classic measure of progress – lines of code per day – turns out to be a horrible measure because it doesn’t measure the quality of the code, so it cannot take into account the rework required when fast code production results in low code quality. Nor is refactoring code “negative work” because it results in fewer lines of code. A better measure would be velocity, which is a measure of tested and verified features released per unit of time.

Another often abused measure is “hours worked.” I have seen companies require detailed reporting on hours spent per project only to also levy the requirement that any hours worked over 40 hours per week should not be reported. This constrained metric does not actually measure the effort expended on project tasks.

Ignoring the metrics

I have seen many companies spend a lot of time gathering metric data (and yes, it does require some effort and does cost some time, even when mostly automated), only to make the very same mistake time after time. This is because while these companies capture the data, they never actually use the data to improve.

No authority to initiate change

Gathering and analyzing metrics is often seen as less valuable than “real work” and so personnel tasked with these activities have little or no authority.

Lack of willingness to follow through

I have seen companies pay for detailed, quantified project performance data only to ignore it because there was little willingness to follow through with needed changes. This lack of willingness can come from management being unwilling to pay for organizational improvement, or from technical staff being afraid of trying something different.

Metrics should always be attempting to measure an objective rather than a means. Rather than “lines of code per day,” it is better to measure “delivered functionality per day.”

Managing risk

In my experience, most unsuccessful projects fail because they don’t properly deal with project risk. Project risk refers to the potential for change that a team will fail to meet some or all of a project’s objectives. Risk is defined to be the product of an event’s likelihood of occurrence times its severity. Risk is always about the unknown. There are many different kinds of project risk. For example:

- Resource risk
- Technical risk
- Schedule risk
- Business risk

Risks are always about the *unknown* and risk mitigation activities – known as *spikes* in agile literature – are work undertaken to uncover information to reduce risk. For example, a technical risk might be that the selected bus architecture might not have sufficient bandwidth to meet the system performance requirements. A spike to address the risk might measure the bus under stress similar to what is expected for the product. Another technical risk might be the introduction of new development technology, such as SysML, to a project. A resulting spike might be to bring in an outsider trainer and mentor for the project.

The most important thing you want to avoid is *ignoring risk*. It is common, for example, for projects to have “aggressive schedules” (that is to say, “unachievable”) and for project leaders and members to ignore obvious signs of impending doom. It is far better to address the schedule risk by identifying and addressing likely causes of schedule slippage and replan the schedule.

Purpose

The purpose of the *Managing risk* recipe is to improve the likelihood of project success.

Inputs and preconditions

Project risk management begins early and should be an ongoing activity throughout the project. Initially, a project vision, preliminary plan, or roadmap serves as the starting point for risk management.

Outputs and postconditions

Intermediate outputs include a *risk management plan* (sometimes called a *risk list*) and the work effort resulting from it, allocated into the release and iteration plans. The risk management plan provides not only the name of the risk but also important information about it. Longer-term results include a (more) successful project outcome than one that did not include risk management.

How to do it

Figure 1.16 shows how risks are identified, put into the risk management plan, and result in spikes. Figure 1.17 shows how, as spikes are performed in the iterations, the risk management plan is updated:

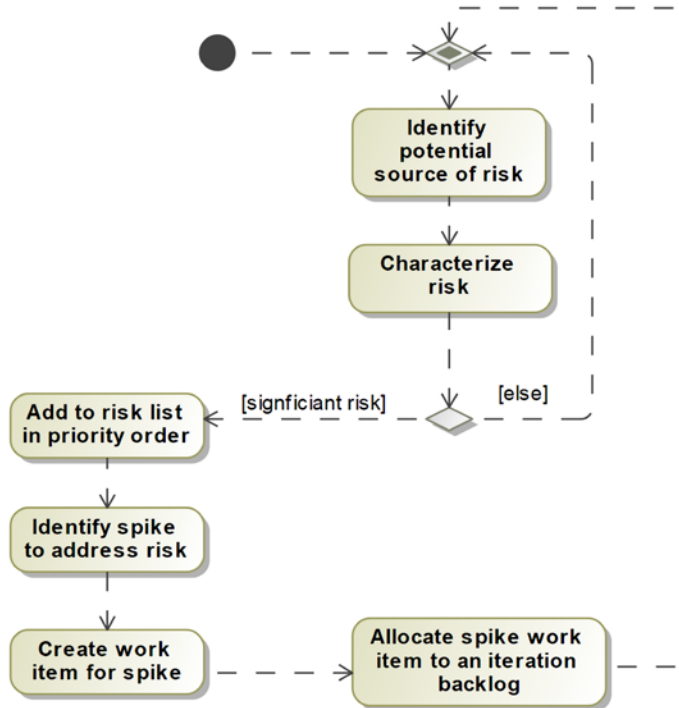


Figure 1.16: Managing risk

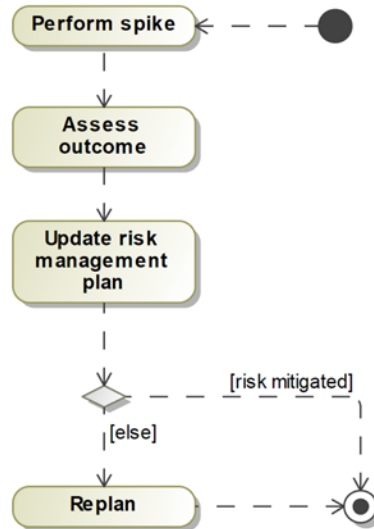


Figure 1.17: Reducing risk

Identify a potential source of risk

This is how it starts, but risk identification shouldn't just be done at the outset of the project. At least once per iteration, typically during the project retrospective activity, the team should look for new risks that have arisen as the project has progressed. Thus, the workflow in *Figure 1.16* isn't performed just once but many times during the execution of the project. In addition, it sometimes happens that risks disappear if their underlying causes are removed, so you might end up removing risk items, or at least marking them as avoided, during these risk reassessments.

Characterize risk

The name of the risk isn't enough. We certainly need a description of how the risk might manifest and what it means. We also need to know how likely the negative outcome is to manifest (likelihood) and how bad it is should that occur (severity). Some outcomes have a minor impact, while others may be show-stoppers.

Add to risk list in priority order

The risk management plan maintains the list in order sorted by risk magnitude. If you have quantified both the risk's likelihood and severity, then risk magnitude is the product of those two values. The idea is that the higher-priority risks should have more attention and be addressed earlier than the lower-priority risks.

Identify a spike to address risk

A spike is work that is done to reduce either the likelihood or the severity of the risk outcome, generally the former. We can address knowledge gaps with training; we can address bus performance problems with a faster bus; we can solve schedule risks with featurecide. Featurecide is the removal of features of low or questionable stakeholder value, or work items that you just don't have the bandwidth to address. Whatever the approach, a spike seeks to reduce risk, so it is important that the spike uncovers or addresses the risk's underlying cause.

Create a work item for a spike

Work items come in many flavors. Usually, we think of use cases or user stories (functionality) as work items. But work items can refer to any work activity, as we discussed in the earlier recipe for backlog management. Specifically, in this case, spikes are important work items to be put into the product backlog.

Allocate a spike work item to an iteration plan

As previously discussed, work items must be allocated to iterations to result in a release plan.

Perform a spike

This action means performing the identified experiment or activity. If the activity is to get training, then complete the training session. If it is to perform a lab-based throughput test, then do that.

Assess the outcome

Following the spike, it is important to assess the outcome. Was the risk reduced? Is a change in the plan, approach, or technology warranted?

Update the risk management plan

The risk management plan must be updated with the outcome of the spike.

Replan

If appropriate, adjust the plan in accordance with the outcome of the spike. For example, if a proposed technology cannot meet the project needs, then a new technology or approach must be selected and the plan must be updated to reflect that.

Example

Here is an example risk management plan, captured as a spreadsheet of information. Rather than show the increasing level of detail in the table step by step, we'll just show the end state (*Table 1.13*) to illustrate a typical outcome from the workflow shown in *Figure 1.16*.

It can be sorted by the **State** and **Risk Magnitude** columns to simplify its use:

Risk Management Plan (Risk List)													
Risk ID	Headline	Description	Type	Impact	Probability	Risk magnitude	State	Precision	Raised on	Iteration #	Impacted stakeholder	Owner	Mitigation strategy (spike)
1	Robustness of the main motor	The system must be able to maintain 2,000 W for up to 5 minutes and sustain 1,000 W for 4 hours, with an MTBF of 20,000 hours. The current motor is unsuitable.	Technical	80%	90%	72%	Open	High	1/5/2020	1	Main-tainer, user	Sam	Meet with motor vendors to see if 1) they have an existing motor that meets our needs, or 2) they can design a motor within budget to meet the need.
2	Agile MBSE impact	The team is using both agile and MBSE for the first time. The concern is that this may lead to poor technical choices.	Technical	80%	80%	64%	Open	Medium	1/4/2020	0	User, product owner	Jill	Bring in a consultant from aPriori Systems for training and mentoring

3	Robustness of USB connection	Users will be inserting and removing the USB while under movement stress, so it is likely to break.	Technical	40%	80%	32%	Open	Medium	2/16/2020	3	User, manufacturing	Joe	Standard USB connectors are too weak. We need to mock up a more robust physical design.
4	Aggressive schedule	Customer schedule is optimistic. We need to address this either by changing the expectations or figuring out how to satisfy the schedule.	Schedule	40%	100%	40%	Mitigated	Low	12/5/2019	0	Buyer	Susan	Iteration 0, work with the customer to see if the project can be delivered in phases, or if ambitious features can be cut.
5	Motor response lag time	To simulate short high-intensity efforts, the change in resistance must be fast enough to simulate the riding experience.	Technical	20%	20%	4%	Open	High	12/19/2019	6	User	Sam	Do a response time study with professional riders to evaluate the acceptability of the current solution.

6	Team availability	Key team members have yet to come off the Aerobike project and are delayed by an estimated 6 months.	Re-source	60%	75%	45%	Obsolete	Low	3/1/2020	0	Product owner, buyer		See if the existing project can be sped up. If not, work on a contingency plan to either hire more or delay the project start.
---	-------------------	--	-----------	-----	-----	-----	----------	-----	----------	---	----------------------	--	--

Table 1.3: Example risk list

For an example of the risk mitigation workflow in *Figure 1.17*, let's consider the first two risks in *Table 1.3*.

Perform a spike

For Risk 2, "Agile MBSE impact," the identified spike is "Bring in a consultant from *A Priori Systems* for training and mentoring." We hire a consultant from *A Priori Systems*. They then train the team on agile MBSE, gives them each a copy of their book *Agile Systems Engineering*, and mentors the team through the first three iterations. This spike is initiated in Iteration 0, and the mentoring lasts through Iteration 3.

For Risk 1, "Robustness of the main motor," the identified spike is "Meet with motor vendors to see if 1) they have an existing motor that meets our needs, or 2) they can design a motor within our budget to meet the need." Working with our team, the application engineer from the vendor assesses the horsepower, torque, and reliability needs and then finds a version of the motor that is available within our cost envelope. The problem is resolved.

Assess outcome

The assessment of the outcome of the spike for Risk 2 is evaluated in four steps. First, the engineers attending the agile MBSE workshop provide an evaluation of the effectiveness of the workshop. While not giving universally high marks, the team was very satisfied overall with their understanding of the approach and how to perform the work. The iteration retrospective for the next three iterations look at expected versus actual outcomes and find that the team is performing well. The assessment of the risk is that it has been successfully mitigated.

For Risk 1, the assessment of the outcome is done by the lead electronics engineer. He obtains five instances of the suggested motor variant and stress-tests them in the lab. He is satisfied that the risk has been successfully mitigated and that the engineering can proceed.

Update the risk management plan

The risk management plan is updated to reflect the outcomes as they occur. In this example, *Table 1.4*, we can see the updated **State** field in which the two risk states are updated to **Mitigated**:

Risk Management Plan (Risk List)													
Risk ID	Head-line	Description	Type	Impact	Probability	Risk Magnitude	State	Precision	Raised On	Iteration #	Impacted Stakeholder	Owner	Mitigation Strategy (Spike)
1	Robustness of the main motor	The system must be able to maintain 2,000 W for up to 5 minutes and sustain 1,000 W for 4 hours, with an MTBF of 20,000 hours. The current motor is unsuitable.	Technical	80%	90%	72%	Mitigated and updated motor selection to the appropriate variant	High	1/5/2020	1	Main-tainer, user	Sam	Meet with the motor vendors to see if 1) they have an existing motor that meets our needs, or 2) they can design a motor without our OEM costing to meet the need.

2	Agile MBSE impact	The team is using both agile and MBSE for the first time. The concern is that this may lead to back technical choices.	Technical	80%	80%	64%	Mitigated, updated modeling tool for Rhapsody, and MBSE workflows updated.	Medium	1/4/2020	0	User, buyer, product owner	Jill	Bring in a consultant from A Priori Systems for training and mentoring.
---	-------------------	--	-----------	-----	-----	-----	--	--------	----------	---	----------------------------	------	---

Table 1.4: Updated risk plan (Partial)

Replan

In this example, the risks are successfully mitigated and the changes are noted in the State field. For Risk 1, a more appropriate motor is selected with help from the motor vendor. For Risk 2, the tooling was updated to better reflect the modeling needs of the project, and minor tweaks were made to the detailed MBSE workflows.

Product roadmap

A *product roadmap* is a plan of action for how a product will be introduced and evolved over time. It is developed by the *product owner*, an agile role responsible for managing the product backlog and feature set. The product roadmap is a high-level strategic view of the series of delivered systems mapped to capabilities and customer needs. The product roadmap takes into account the market trajectories, value propositions, and engineering constraints. It is ultimately expressed as a set of initiatives and capabilities delivered over time.

Purpose

The purpose of the product roadmap is to plan and provide visibility to the released capabilities of the customers over time. The roadmap is initially developed in Iteration 0, but as in all things agile, the roadmap is updated over time. A typical roadmap has a 12–24 month planning horizon, but for long-lived systems, the horizon may be much longer.

Inputs and preconditions

A product vision has been established which includes the business aspects (such as market and broad customer needs) and technical aspects (the broad technical approach and its feasibility).

Outputs and postconditions

The primary work product is the product roadmap, a time-based view of capability releases of the system.

How to do it

The product roadmap is organized around larger-scale activities (epics) for the most part, but can contain more detail if desired. An *epic* is a capability whose delivery spans multiple iterations. Business epics provide visible value to the stakeholders, while technical epics (also known as enabler epics) provide behind-the-scenes infrastructure improvements such as architecture implementation or the reduction of technical debt.

In an MBSE approach, epics can be modeled as stereotypes of use cases that are decomposed to the use cases, which are in turn, decomposed into user stories (stereotyped use cases) and scenarios (refining interactions). While epics are implemented across multiple iterations, a use case is implemented in a single iteration. A user story or scenario takes only a portion of an iteration to complete. User stories and scenarios are comparable in scope and intent.

This taxonomy is shown in *Figure 1.18*, along with where they typically appear in the planning:

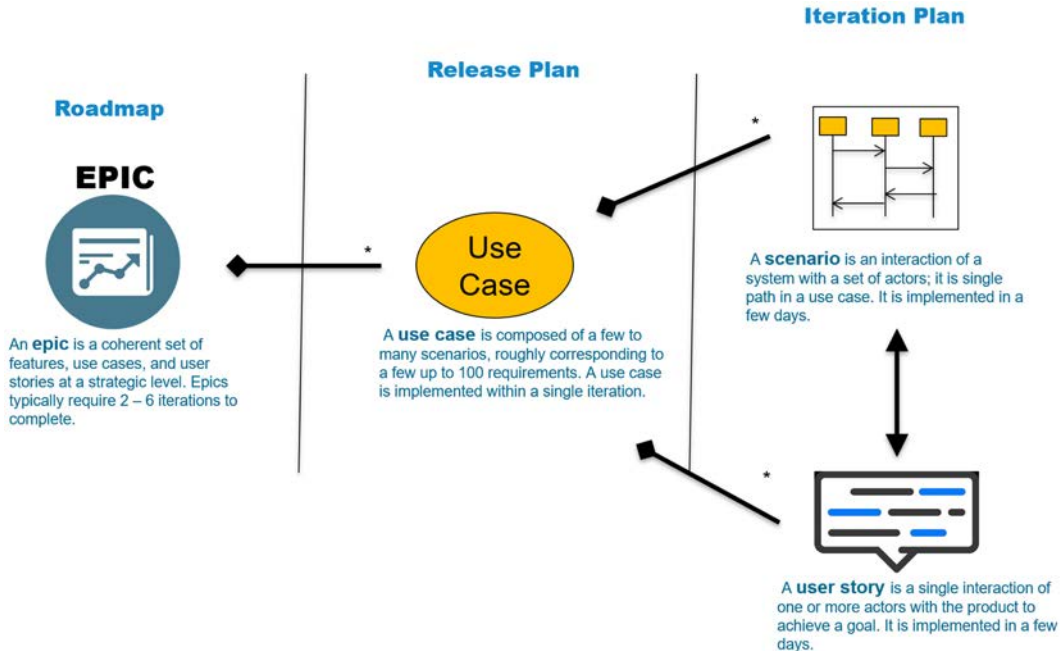
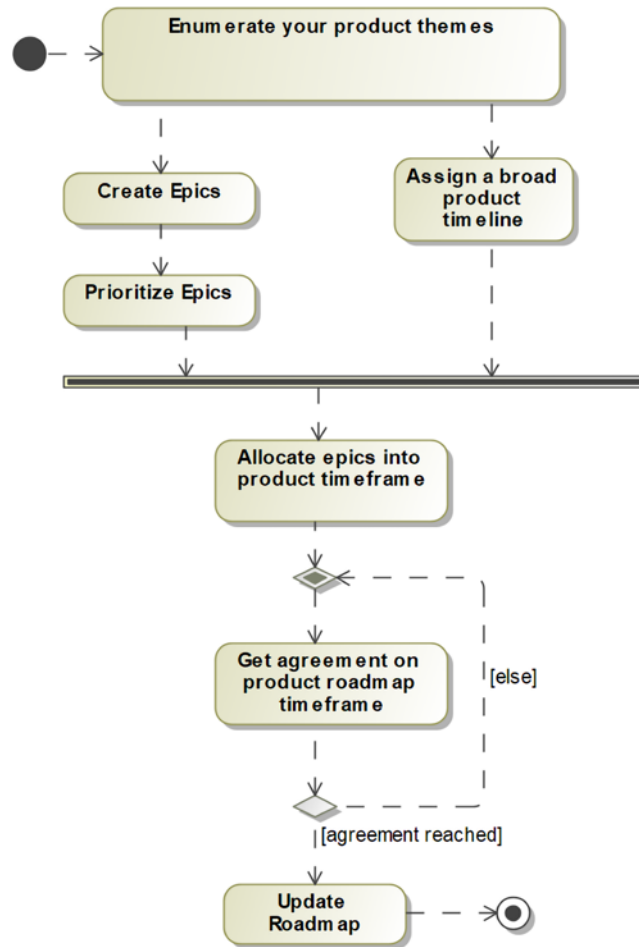


Figure 1.18: Epics, use cases, and user stories

The product roadmap is a simple planning mechanism relating delivered capability to time, iterations, and releases. Like all agile planning, the roadmap is adjusted as additional information is discovered, improving its accuracy over time. The roadmap updates usually occur at the end of each iteration during the iteration retrospective, as the actual iteration outcomes are compared with planned outcomes.

The roadmap also highlights milestones of interest and technical evolution paths as well. Milestones might include customer reviews or important releases, such as alpha, beta, an **Initial Operating Condition (IOC)**, or a **Final Operating Condition (FOC)**:



C

Figure 1.19: Create product roadmap

Enumerate your product themes

The product themes are the strategic objectives, values, and goals to be realized by the product. The epics must ultimately refer back to how they aid in the achievement of these themes. This step lists the product themes to drive the identification of the epics and work items going forward. In some agile methods, the themes correlate to *value streams*.

Create epics

Epics describe either the strategic capabilities of the system to realize the product themes. They can be either **business epics** that bring direct value to the stakeholders, or *technical* (aka *enabler*) *epics* that provide technological infrastructure to support the business epics. Epics may be thought of as large use cases that generally span several iterations. This step identifies the key epics to be put into the product roadmap.

Prioritize epics

Prioritization identifies the order in which epics are to be developed. Prioritization can be driven by urgency (the timeliness of the need), criticality (the importance of meeting the need), the usefulness of the capability, the availability of the required resource, reduction in project risk, natural sequencing, or meeting opportunities – or any combination of the above. The details of how to perform prioritization are the subject of their own recipe (see the *Work item prioritization* recipe in this chapter), but this is one place where prioritization can be effectively used.

Assign a broad product timeframe

The product roadmap ultimately defines a range of time in which capabilities are to be delivered. This differs from traditional planning, which attempts to nail down the second when a product will be delivered in spite of the lack of adequate information to do so. The product roadmap usually defines a large period of time – say a month, season, or even year – in which a capability is planned to be delivered, but with the expectation that this timeframe can be made more precise as the project proceeds.

Allocate epics in the product timeframe

Epics fit into the product timeframe to allow project planning at a strategic level.

Get agreement on the product roadmap

Various stakeholders must agree on the timeframe. Users, purchasers, and marketers must agree that the timeframe meets the business needs and that the epics provide the appropriate value proposition. Engineering staff must agree that the capabilities can be reasonably expected to be delivered with an appropriate level of quality within the timeframe. Manufacturing staff must agree that the system can be produced in the plan. Regulatory authorities must agree that the regulatory objectives will be achieved.

Update the roadmap

If stakeholders are not all satisfied, then the plan should be reworked until an acceptable roadmap is created. This requires modification and reevaluation of an updated roadmap.

Example

Let's create a product roadmap for the Pegasus system by following the steps outlined.

Enumerate your product themes

The product themes include:

- Providing a bike fit as close as possible to the fit of a serious cyclist on their road bike
- Providing a virtual ride experience that closely resembles outside riding, including:
 - Providing resistance to pedals for a number of conditions, including flats, climbing, sprinting, and coasting for a wide range of power outputs from casual to professional riders
 - Simulating gearing that closely resembles the most popular gearing for road bicycles
 - Incline control to physically incline or decline the bike
- Permitting programmatic control of resistance to simulate changing road conditions in a realistic fashion
- Interfacing with cycling training apps, including Zwift, Trainer Road, and the Sufferfest
- Gathering ride, performance, and biometrics for analysis by a third-party app
- Providing seamless **Over-The-Air (OTA)** updates of product firmware to simplify maintenance

Create epics

Epics describe either the strategic capabilities of the system to realize the product themes. This step identifies the key epics to be put into the product roadmap. Epics include:

Business epics:

- Physical bike setup
- Ride configuration
- Firmware updates

- Controlling resistance
- Monitoring road metrics
- Communicating with apps
- Emulating gearing
- Incline control

Enabler epics:

- Mechanical frame development
- Motor electronics development
- Digital electronics development

Prioritize epics

These epics are not run fully sequentially, as some can be done in parallel. Nevertheless, the basic prioritized list is:

1. Mechanical frame development
2. Motor electronics development
3. Digital physical bike setup
4. Monitor road metrics
5. Ride configuration
6. Control resistance
7. Emulating gearing
8. Communicating with apps
9. Firmware updates
10. Incline control
11. Electronics development

Assign a broad product timeframe

For this project, the total timeframe is about 18 months, beginning in early spring 2021 and ending at the end of 2022, with milestones for fall 2021 (a demo at the September Eurobike tradeshow), spring 2022 (the alpha release), summer 2022 (beta), and the official release (October 2022).

Allocate epics into a product timeframe

Figure 1.20 shows a simple product roadmap for the Pegasus system. At the top, we see the planned iterations and their planned completion dates. Below that, important milestones are shown. The middle part depicts the evolution plan for the three primary hardware aspects (the mechanical frame, motor electronics, and digital electronics). Finally, the bottom part shows the high-level system capabilities as epics over time, using color coding to indicate priority:

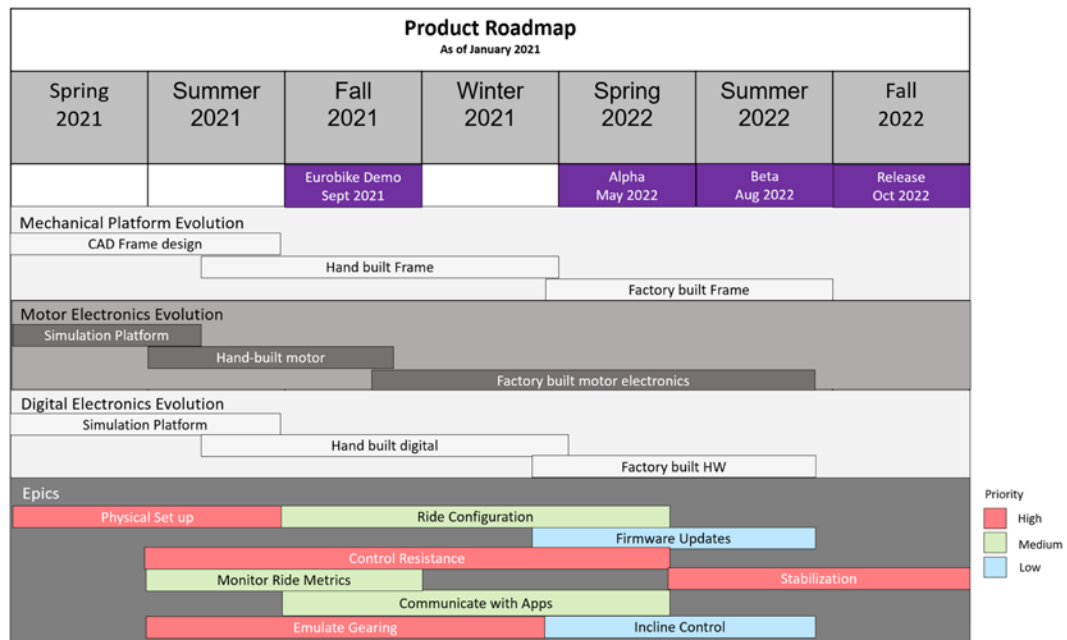


Figure 1.20: Pegasus Product Roadmap

Note the pseudo-epic “Stabilization” appears in the figure and indicates a period of removal of defects and refinement of capability.

Get agreement on the product roadmap

We discuss the roadmap with stakeholders from marketing, engineering, manufacturing, and our customer focus group to agree on the product themes, epics, and timeframes.

Update roadmap

The focus group identifies that there is another tradeshow in June 2022 that we should try to have an updated demo ready for. This is then added to the product roadmap.

Release plan

While the product roadmap is strategic in nature, the *release plan* is more tactical. The product roadmap shows the timing of release goals, high-level product capabilities, and epics that span multiple iterations, but the release plan provides more detail on a per-iteration basis. The product roadmap has a longer planning horizon of 12–24 months while a release plan is more near-term, generally three to nine months. This recipe relies on the *Managing your backlog* recipe that appears earlier in this chapter.

Purpose

The purpose of the release plan is to show how the product backlog is allocated to the upcoming set of iterations and releases over the next three to nine months.

Inputs and preconditions

The product vision and roadmap are sketched out and a reasonably complete product backlog has been established, with work items that can fit within a single iteration.

Outputs and postconditions

The release plan provides a plan for the mapping of work items to the upcoming set of iterations and releases. Of course, the plan is updated frequently – at least once per iteration – as work is completed and the depth of understanding of the product development increases.

How to do it

Epics and high-level goals need to be decomposed into work items that can be completed within a single iteration. Each of these work items is then prioritized and its effort is estimated. The release plan identifies the specifically planned iterations, each with a mission (as shown in *Figure 1.4*). There is some interplay between the missions of the iterations and the priority of the work items. The priority of a work item might be changed so that it is completed in the same iteration as a set of related work items.

Once that has been done, the mapping of the work items to the iterations can be made. The mapping must be evaluated for reasonableness and adjusted until the plan looks both good and achievable. This workflow is shown in *Figure 1.21*:

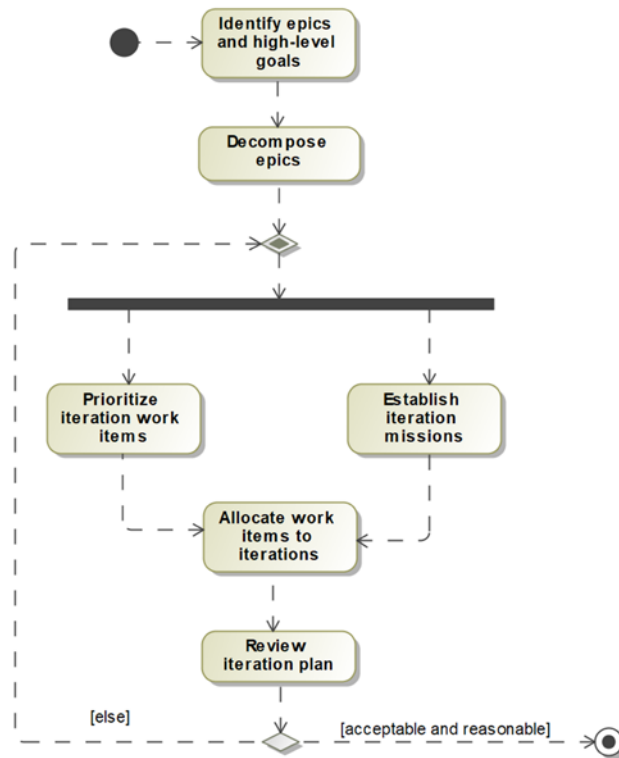


Figure 1.21: Release planning

Identify epics’ high-level goals

If you’ve done a product roadmap (see the *Product roadmap* recipe), then you are likely to already have established the epics and high-level goals (themes) for the product. If not, see the recipe for how to do that.

Decompose epics

Epics are generally too large to be completed in a single iteration, so they must be decomposed into smaller pieces – use cases and technical work items, and possibly user stories and scenarios – that can be completed within a single iteration. These will be the work elements allocated to the iterations.

Establish iteration missions

Each iteration should have a mission, including purpose, scope, and themes. This was discussed in the *Managing your backlog* recipe earlier in this chapter.

This mission includes:

- Use cases to be implemented
- Defects to be repaired
- Platforms to be supported
- Risks to be reduced
- Work products to be developed

Prioritize iteration work items

A work item's priority specifies the order in which it should be developed. Prioritization is a subject of its own recipe, *Work item prioritization*. Here it is enough to say that higher-priority work items will be performed in earlier iterations than lower-priority work items.

Allocate work items to iterations

This step provides a detailed set of work items to be performed within the iteration (known as the *iteration backlog*). Ultimately, all work items are either allocated to an iteration, decomposed into smaller work items that are allocated, or are removed from the product backlog.

Review iteration plan

Once the allocations are done, the iteration plan must be reviewed to ensure that the release plan is:

1. Consistent with the product roadmap
2. Has iteration allocations that can be reasonably expected to be achievable
3. Has work item allocations that are consistent with the mission of their owner iterations

Example

While the product roadmap example we did in the previous recipe focused on a somewhat-vague strategic plan, release planning is more tactical and detailed. Specific work items are allocated to specific iterations and reviewed and “rebalanced” if the release plan has discernable flaws. For this example, we'll look at a planning horizon of six iterations (plus Iteration 0) and focus on the allocations of functionality, technical work items, platforms to be supported, and spikes for the reduction of specific risks.

Identify high-level goals

The high-level goals are identified in the project plan from the previous recipe, as exemplified in the business and enabler epics.

Decompose epics

The epics to be implemented in the iterations in this planning horizon must be decomposed into use cases and technical work items achievable within the allocated iteration. *Figure 1.22* shows the decomposition of the epics into use cases and user stories. Note that epics (and, for that matter, user stories) are modeled as stereotypes of use cases, and the figure is a use case diagram with the purpose of visualizing that decomposition. Since epics and user stories are represented as stereotypes of use cases, the «include» relationship is used for decomposition:

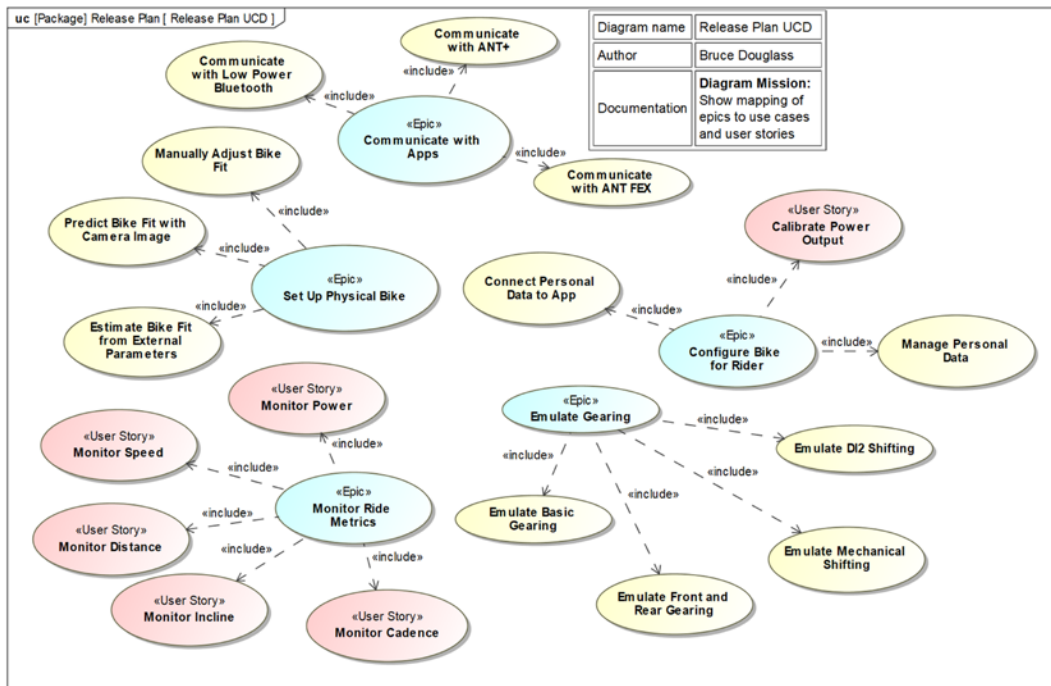


Figure 1.22: Mapping epics to use cases

Establish iteration missions

To establish the mission for each iteration, a spreadsheet is created (*Table 1.4*) with the iterations as columns and the primary aspects of the mission as rows.

Prioritize iteration work items

Work items are prioritized to help us understand the sequencing of the work in different iterations. As much as possible, elements with similar priorities are put within the same iteration.

As discussed in the *Work item prioritization* recipe, during a review, we may increase or decrease a work item's priority to ensure congruence with other work items done in a specific iteration.

Allocate work items to iterations

Based on the prioritization and the work capacity within an iteration, the work items are then allocated (*Table 1.5*).

Table 1.5 shows an example in which allocations are made based on priority (see the *Work item prioritization* recipe), the estimated effort (see the *Estimating effort* recipe), and the congruency of the functionality to the mission of the use case:

Release plan	Iteration 0	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6
Functionality		Initial Frame Mockup, Basic Motor Electronics, Basic Rider Controls, and Basic Resistance	Set Up the Bike Fit (seat), Basic Digital Electronics, Calibrate Power Output, and Basic gearing	Set up the Bike Fit (handle-bars), Manually adjust the bike fit, and Monitor Power	Set up the Bike Fit (Cranks), and Monitor Speed, Distance, Bluetooth, Cadence, and Data to the App	Bike fit with external parameters, Motorized Incline, Monitor Incline, and ANT+, and ANT FEC	Manage personal data, and Predict the Bike with a Camera Image, External Resistance control, and ERG Mode
Target Platforms		Hand-build mechanicals, Hand-built analog electronics, and Simulated digital electronics	Basic hand-built mechanicals and Hand-built electronics	Prototype mechanicals for manufacturing	First-run factory electronics	First run mechanicals	Second-run factory electronics and 2nd run factory mechanicals

Technical Work Items		Analyze frame stability and strength and Refine SW/EE deployment architecture	Design cable runs, Analyze electrical power needs, and Add in SW concurrency architecture	Add in an SW Distribution Framework	Finalize flywheel mass	EMI Conformance testing	
Spikes	Team Availability, Aggressive Schedule, and Agile MBSE Impact	Motor Response Time	Robustness of the main motor	USB Robustness			

Table 1.5: Release plan

Review iteration plan

We then look at the release plan and see that we think it is achievable, the missions of the iterations are reasonable, and the allocations of work items make sense for the project.

Iteration plan

The iteration plan plans out a specific iteration in more detail, so the planning horizon is a single iteration. This is typically 1–4 weeks in duration. This is the last chance to adjust the expectations of the iteration before work begins.

Purpose

The purpose of the iteration plan is to ensure that the work allocated to the iteration is achievable, decompose the larger-scale work items (for example, use cases and technical work items) into smaller work items, and plan for the completion of the iteration.

Inputs and preconditions

Preconditions include the release plan and the initial iteration backlog.

Outputs and postconditions

The resulting plan includes the complete work items, generated engineering work products, identified defects and technical work items (pushed into the product backlog), and uncompleted work items (also pushed back onto the product backlog).

How to do it

Use cases in the iteration backlog, which may take an entire iteration to fully realize, are decomposed into user stories or scenarios, each of which takes a few hours to a few days to realize. The iteration plan is created *just-in-time* before the start of the iteration but is based on the release plan. This flow is shown in *Figure 1.23*:

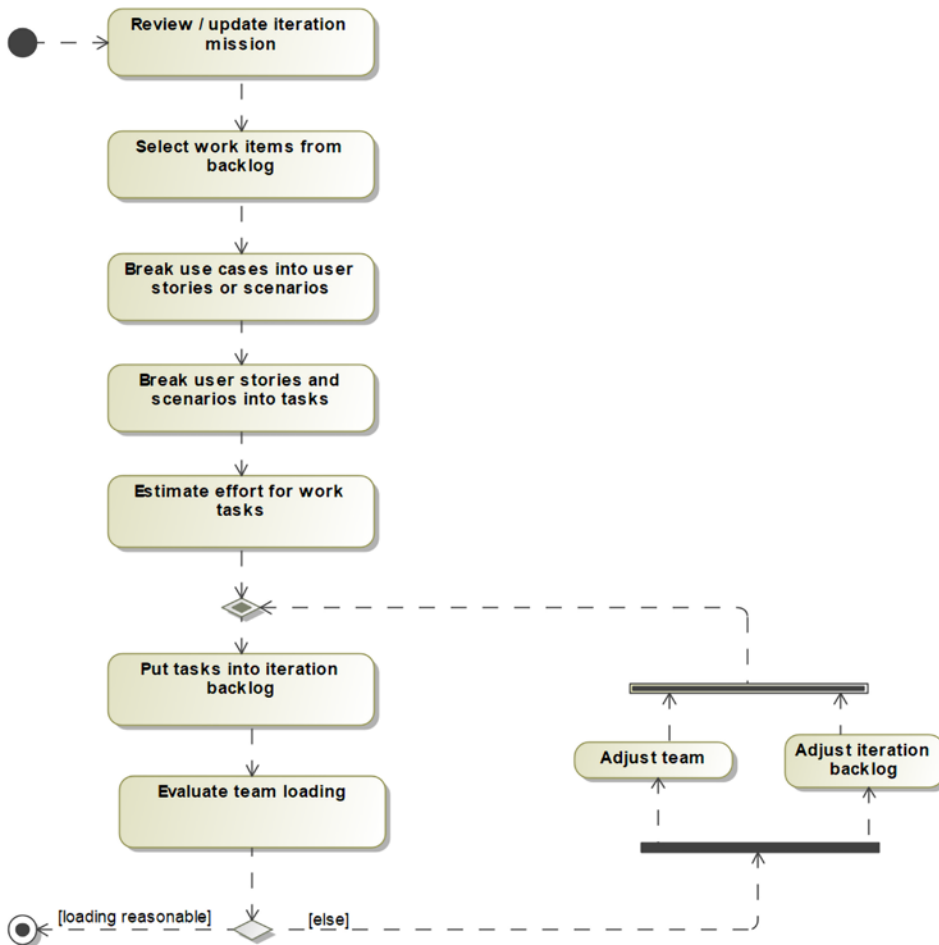


Figure 1.23: Iteration planning

Review/update the iteration mission

The iteration should already have a mission from the release plan. This will include:

- Functionality to be achieved (in use cases, user stories, and/or scenarios)
- Target platforms to be supported
- Architectural and other technical work items to support the functionality and technical epics
- Defects identified in previous iterations
- Spikes to reduce risks

There is a decent chance that this mission will require updating, based on lessons learned in preceding iterations, so this is a place to do that if it has not already been done. Any changes made here may impact the allocation of work items to the iteration backlog.

Select work items from the backlog

Based on the iteration mission, the list of work items allocated is reviewed. Some may be removed or new ones added, as necessary and appropriate.

Break use cases into user scenarios or user stories

Use cases themselves are generally rather large and it is useful to have smaller work items in the backlog. These work items might be estimated to take anywhere from a few hours to a few days. Note that estimation of epics and use cases is often done using relative measures (e.g., use case points) but once you get down a few hours in duration, estimates often transition to hour-based, as determined by the team's velocity.

Break use stories into tasks

If the user stories are small, then this step can be skipped. If they are still rather large, say a week or two, then they might be decomposed further into smaller tasks. This step is optional.

Estimate the effort for work tasks

If you've decomposed the original iteration backlog work items, then those elements should be estimated. This can be done either using relative measures, such as story points, or absolute measures, such as the number of hours to complete.

Put tasks into the iteration backlog

Any modified or newly created work item tasks must be added to the backlog for the iteration.

Evaluation team loading

Once we have a detailed vision of the expected work to do in the upcoming iteration and a pretty good idea of the effort, we can reevaluate whether the scope of work is reasonable.

Adjust team

The size or makeup of the team may be adjusted to better fit the more detailed understanding of the scope of work to be undertaken.

Adjust backlog

If the scope looks too demanding for the team, items can be removed from the iteration backlog and pushed back to the product backlog. This will spin off an effort later to rebalance the release plan. Note that this is also done at the end of the iteration, when the team can see what planned work was not achieved.

Iteration planning is pretty simple as long as you keep some guidelines in place. The larger-scale work items allocated to the iteration are sized to fit into a single iteration. However, they are decomposed into somewhat smaller pieces, each taking from a few hours to a few days to complete. For use cases, this will be either user stories or scenarios; this decomposition and analysis will be detailed in the recipes of the next chapter. The work items should all fit within the mission statement for the iteration, as discussed in the first recipe, *Managing your backlog*.

The work items should all contribute to the mission of the iteration. If not, they should either be pushed back to the product backlog or the iteration mission should be expanded to include them. It is also helpful to have the larger-scale work items broken down into relatively small pieces; you should be less concerned about whether they are called use cases, user stories, scenarios, or tasks, and more concerned that they 1) contribute to the desired functionality, and 2) are in the right effort scope (a few hours to a few days). Work items that are too large are difficult to estimate accurately and may not contribute to understanding the work to be done. Work items that are too small waste planning time and effort.

Example

For our example, let's plan Iteration 4.

Review/update the iteration mission

The mission for a hypothetical iteration is shown in *Table 1.6*:

Release plan	Iteration use cases	Iteration user stories	Effort (hours)
Functionality	Predict the Bike Fit with a Camera		
	Estimate the Bike Fit from External Parameters		
		Monitor the Distance	
		Calibrate the Power Output	
		Provide Basic Resistance	
		Set resistance under user control	
Target Platforms	First-run factory electronics Hand-built mechanical frame		
Technical Work Items	Finalize the flywheel mass		
Spikes	<none>		

Table 1.6: Iteration mission

Select work items from the backlog

These work items are selected from the product backlog and placed in the iteration backlog.

Break use cases into scenarios or user stories

Figure 1.24 shows the planned functionality for our hypothetical iteration of the Pegasus bike trainer. The ovals without stereotypes are use cases that are decomposed with the «include» relation into user stories.

Each of these is then estimated to get an idea of the scope of the work for the iteration:

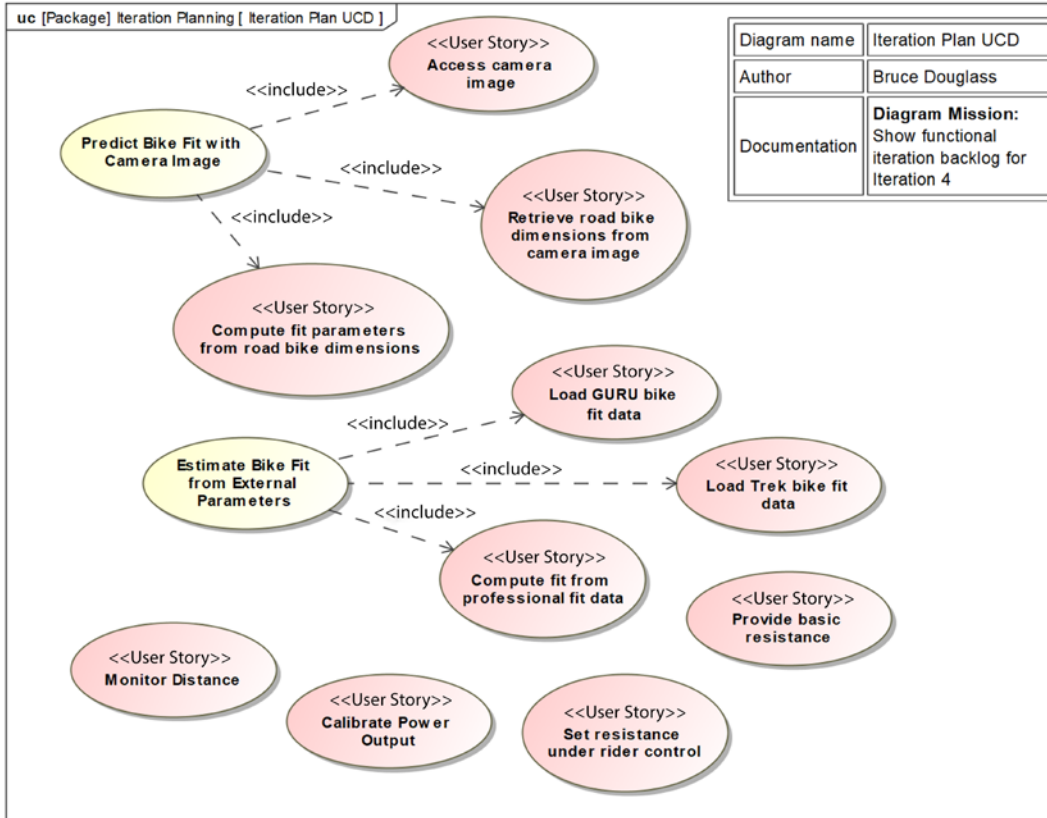


Figure 1.24: Iteration planning example

Break user stories into tasks

These user stories are all pretty small, so this optional step is skipped.

Estimate effort for work tasks

We then estimate the hours required to complete these small work items for the task. This results in the updated table in *Table 1.7*:

Release plan	Iteration use cases	Iteration user stories/work items	Effort (hours)	
Functionality	Predict the bike fit with a Camera	Access the camera image	2	
		Retrieve the road bike dimensions from the camera image	16	
		Compute the fit parameters from the road bike dimensions	4	
	Estimate the bike fit from the external parameters	Load GURU bike fit data	4	
		Load trek bike fit data	4	
		Compute fit from professional fit data	2	
			Monitor distance	6
			Calibrate power output	12
			Provide basic resistance	20
			Set resistance under user control	4
Target platforms	First-run factory electronics			
	Hand-built mechanical frame			
Technical work items		Finalize flywheel mass	4	
Spikes	<none>			
Totals			78	

Table 1.7: Iteration 4 mission with estimates

Put tasks into the iteration backlog

Work items from the **Iteration User Stories/Work Items** column are added to the backlog for the iteration.

Evaluate team loading

The six-person team executing the iteration should be able to complete the estimated 78 hours of work in the 2-week iteration timeframe.

Adjust the team

No adjustment to the team is necessary.

Adjust the backlog

No adjustment to the backlog is necessary.

Estimating Effort

Traditionally absolute duration measures – such as person hours – are used to estimate tasks. Agile approaches generally apply relative measures, especially for large work items such as epics, use cases, and larger user stories. When estimating smaller work items of a duration of a few hours, it is still common to use person-hours. The reasoning is that it is difficult to accurately estimate weeks- or months-duration work items, but there is better accuracy in estimating small work items of 1–4 hours.

There are a number of means by which effort can be estimated, but the one we will discuss in this recipe is called *planning poker*. This is a cooperative game-like approach to converge on a relative duration measure for a set of work items.

Purpose

The purpose of effort estimation is to understand the amount of effort required to complete a work item. This may be expressed in absolute or relative terms, with relative terms preferred for larger work items.

Inputs and preconditions

A backlog of work items for estimation.

Outputs and postconditions

The primary outcome is a set of relative effort estimates of the work required to complete each work item from the set, or shelving a set of work items that the team agrees requires additional clarification or information.

How to do it

Work durations come in different sizes. For the most part, epics are capabilities that require at least two iterations to perform. Epics are typically broken down into use cases that are expected to be completed within a single iteration. User stories and scenarios are singular threads within a use case that require a few hours to a few days to complete. To be comparable, the epic's work estimates must be, in some sense, the sum of the work efforts for all its contained use cases, and the use case work estimates are the sum of the effort of all its contained user stories and scenarios.

Of course, the real world is slightly more complex than that. The last sentence of the preceding paragraph is true only when the user stories and scenarios are both independent and complete; this means that all the primitive behaviors contained within the use case appear in exactly one use case or user story. If there is overlap – that is, a primitive behavior appears as a part of two scenarios – then the use case estimate is the sum of the user story estimates minus the overlapping behavior. This removes “double counting” of the common behavior. Since these are relative and approximate measures, such subtleties are generally ignored.

How it works

Use case points or user story points are a relative measure of effort. The project velocity (see the *Measuring your success* recipe for more details) maps points to person-hours. Velocity is often unknown early in the project but becomes better understood as the project progresses. The value of use case or user story points is that they remove the temptation of being overly (and *erroneously*) precise about estimated effort. All absolute work estimates assume an implied velocity, but in practice, velocity varies based on team size, team skill, domain knowledge, work item complexity, tools and automation, development environment factors, and regulation and certification concerns.

Figure 1.25 shows the workflow for planning poker:

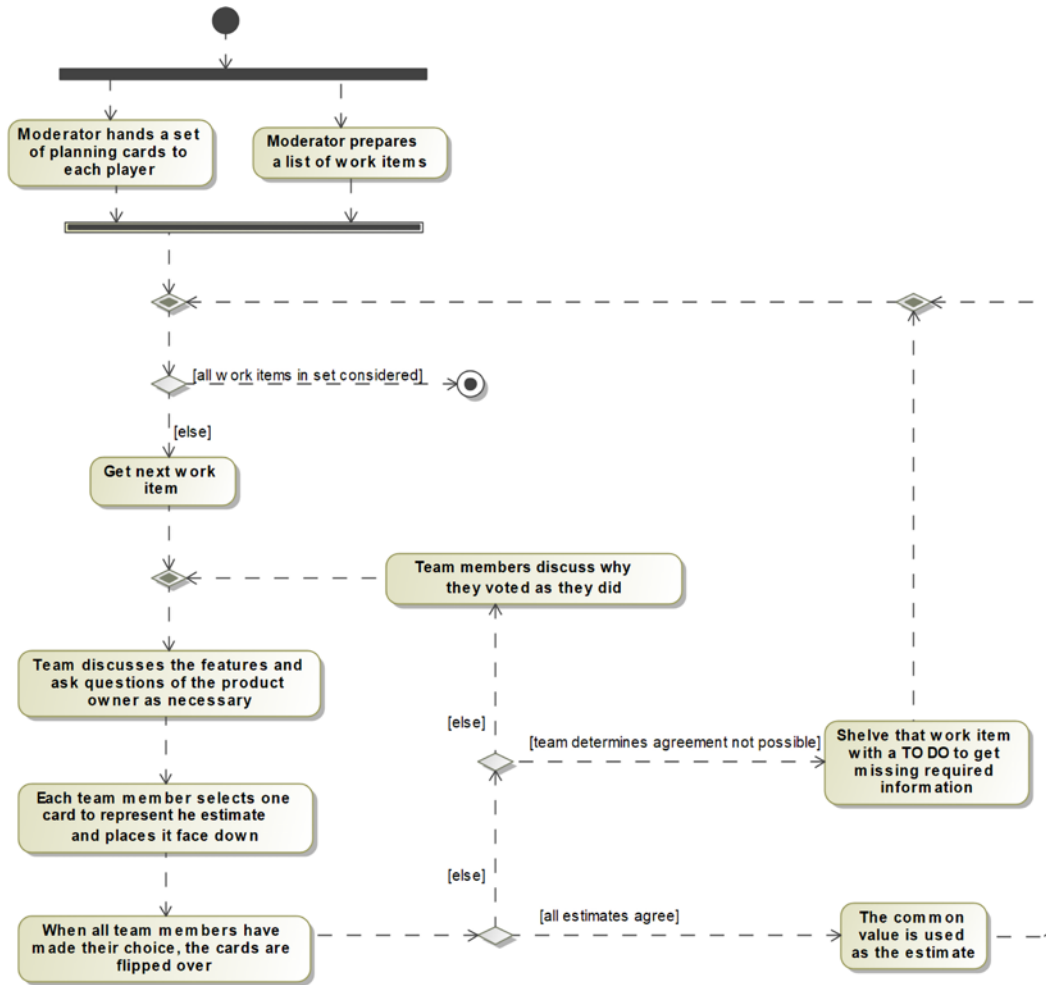


Figure 1.25: Planning poker

Moderator prepares a list of work items

The moderator of the planning sessions prepares a list of work items, which are generally epics, use cases, user stories, or scenarios. In addition to these common items, spikes, technical work items, defect repairs and other work items may be considered as a part of the session as well.

Moderator hands a set of planning cards to each player

These “planning cards” have an effort estimate on one side but are otherwise identical.

Most commonly, numbers in a Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, 34, 55, 89, and 144) or something similar are used.

Get the next work item

Start with the first work item. I recommend beginning with what appears to be the smallest work item, and this will often serve as a standard by which subsequent work items will be judged. As each work item is either estimated or shelved, go to the next.

Team discusses the features and asks questions of the product owner

It is crucial to have a common understanding of what the work item entails. The product owner is the person who generally has the best understanding of the user needs but others may play this role for technical work items.

Each team member selects one card to represent their estimate and places it face down

The estimates are approximately linear, so an estimate of “5” will be more than twice as much work as a work item estimate of “2” but less than twice the effort required for an estimate of “3.” The cards are placed face down to ensure that the initial estimate of the work item is the unbiased opinion of the team member.

When all team members have made their choice, the cards are flipped over

Flipping the card over exposes the estimates to the group.

The common value is used as the estimate

If the estimates all agree, then that value is used as the “job size” estimate for the work item, and the team moves on to the next work item.

Shelve that work item with a TO DO to get the missing information

If the team is unable to reach a consensus after multiple voting rounds on a single work item, then the item is shelved until it can be resolved. The underlying assumption is that there must be some crucial bit of misunderstanding or missing information needed. The team agrees to a task to identify the missing information and re-estimate this item in a later session.

Team members discuss why they voted as they did

If the estimates differ, then the team must share why they estimated as they did. This is particularly important for the lowest and highest estimated values.

Considerations

It is important that the relative size of the work items is consistent. If the average user story point is “8,” and on average, a use case contains four user stories, then you would expect the average use case size to be about 34–55 points. If the average epic is split across three use cases, you would expect the average epic estimate to be 144–233 (selecting numbers only from the Fibonacci series). While strict adherence isn’t crucial, planning well is made more difficult if you have a user story, a use case, and an epic with independent point scales.

Example

This example is for the user stories derived from the use case **Control Resistance**.

Use case: Control Resistance

Purpose: Provide variable resistance to the rider to simulate on-road riding experience for ad hoc and planned workouts in Resistance Mode. In ERG mode, the power output is held constant independent of the simulated incline or pedal cadence.

Description: This use case provides variable resistance to rider pedaling depending on a number of factors. The first is gearing. As with on-road cycling, a larger gear ratio results in a higher torque required to turn the pedals. The user can select gears from the emulated gearing (see *Use case: Emulate Gearing*) to change the amount of torque required to turn the pedals. Next, the user can set the “incline” of the bike. The incline adds or subtracts torque required based on the effort it would take to cycle up or down an incline. Lastly, the base level can be set as a starting point from which the previous factors may offset. By default, this is set by the estimated rider effort on a zero-incline smooth grade. The above are all factors in “Resistance Mode,” in which the power output varies as a function of the cadence, gearing, and incline, as described above. In ERG mode, the power is held constant regardless of these factors. ERG mode is intended to enforce power outputs independent of rider pedal cadence. The power level in ERG mode can be manually set by the user or externally set by a training application. In all modes, the power level can be controlled in a range of 0 to 2,000 W.

Now let's consider the user stories derived from this use case:

User story: Provide Basic Resistance



As a rider, I want basic resistance provided to the pedals so I can get a workout with an on-road feel in Resistance Mode.

This means that for a given gear ratio and simulated incline, the rider feels a smooth and consistent resistance to pedaling.

User story: Set Resistance under user control



As a rider, I want to set the resistance level provided to the pedals to increase or decrease the effort for a given gearing, cadence, and incline-simulated road riding.

User story: Set Resistance under external control



As a rider, I want the external training app to set the resistance to follow the app's workout protocol to get the desired workout.

User story: ERG mode



As a rider, I want to pedal at a constant power regardless of variations in simulated terrain, cadence, or gearing to follow the prescribed power settings for my workout protocol.

The other use cases and user stories will be similarly detailed. See the recipes in *Chapter 2, System Specification: Functionality, Safety, and Security Analysis*, for more details on use cases and user stories.

The team votes via planning poker on the efforts for each of these elements, negotiating when there is no agreement, until a consensus on the efforts is reached.

Table 1.8 shows the results:

Work item type				
Epic	Work item use case	Work item user story	Spike or technical work item	Job size (user story points)
			Spike: Team availability	2
			Spike: Aggressive schedule	3
			Spike: Agile MBSE impact	3
Resist	Control resistance	Provide basic resistance		55
			Spike: Motor response lag time	8
			Spike: Robustness of the main motor	5
Set up physical bike	Set up bike fit	Adjust seat height		3
Set up physical bike	Set up bike fit	Adjust seat reach		3
		Calibrate power output		8
Emulate gearing	Emulate front and rear gearing			34
Emulate gearing	Emulate mechanical gearing			34
Emulate gearing	Emulate basic gearing			89
Set up physical bike	Manually adjust bike fit			13
Set up physical bike	Set up bike fit	Adjust handlebar height		3
Set up physical bike	Set up bike fit	Adjust handlebar reach		3
Monitor ride metrics		Monitor power		13
Monitor ride metrics		Monitor speed		5
Monitor ride metrics		Monitor distance		5
Monitor ride metrics		Monitor cadence		5

Communicate with apps	Communicate with low-power Bluetooth			34
Set up physical bike	Set up bike fit	Select crank length		5
Resist	Control resistance	Set resistance under rider control		21
Configure bike for rider	Connect personal data to the app			21
Set up physical bike	Estimate bike fit with external parameters	Compute fit from professional fit data		1
Monitor ride metrics		Monitor incline		8
Communicate with apps	Communicate with ANT+			34
Communicate with apps	Communicate with ANT FEC			55
Set up physical bike	Estimate bike fit with external parameters	Load GURU bike fit data		13
Set up physical bike	Estimate bike fit with external parameters	Load trek bike fit data		13
Resist	Control resistance	ERG mode		55
	Manage personal data			5
Set up physical bike	Predict bike fit with a camera image	Access camera image		2
Set up physical bike	Predict bike fit with a camera image	Retrieve road bike dimensions from the camera image		5
Set up physical bike	Predict bike fit with a camera image	Compute fit parameters from road bike dimensions		2
Resist	Control resistance	Set resistance under external control		39
Emulate gearing	Emulate DI2 gearing			55
			Spike: USB robustness	5

Table 1.8: Story point estimates for work items

Work item prioritization

This recipe is about the prioritization of work items in a backlog. There is some confusion as to the meaning of the term *priority*. Priority is a ranking of when some task should be performed with respect to other tasks. There are a variety of factors that determine priority and different projects may weigh such factors differently. The most common factors influencing priority are:

- Cost of delay – the cost of delaying the performance of the work item, which in turn is influenced by:
 - Criticality – the importance of the completion of the work item
 - Urgency – when the outcome or output of the work item completion is needed
 - Usefulness – the value of the outcome of the work item to the stakeholder
 - Risk – how the completion of the work item affects project risk
 - Opportunity enablement – how the completion of the work item will enable stakeholder opportunity
- Cost – what is the cost or effort needed to complete the work item?
- Sensical sequencing – what are the preconditions of the work item and what other work items depend upon the completion of this work item?
- Congruency – consistency of the work item to the mission of the iteration to which it is assigned
- Availability of resources – what resources, including specialized resources, are needed to complete this work item, and what is their availability?

Some priority schemes will be dominated by urgency while others may be dominated by criticality or resource availability. The bottom line is that work item priority determines which iteration a work item will be allocated to from the project backlog and to a lesser degree when, within an iteration, the work item will be performed.

Purpose

The purpose of work item prioritization is to intelligently plan the work so as to achieve the product goals in an incremental, consistent fashion. Specifically, the goal of work item prioritization is to allocate work items to the iteration backlogs well.

Inputs and preconditions

The product backlog has been created.

Outputs and postconditions

Work items in the product backlog are prioritized so that iteration planning can proceed.

How to do it

There are many ways to prioritize the backlog. Some, such as the MoSCoW method, are qualitative. Must, Should, Could, Won't prioritization as described in the **International Institute of Business Analysis (IIBA) Business Analysis Body of Knowledge (BABOK) Guide**, www.iiba.org/babok-guide.aspx. In this approach, work items are categorized into the following four groups:

- **Must:** A requirement that must be satisfied in the final solution for the product to be considered a success
- **Should:** Represents a high-priority work item that should be included in the final solution if possible
- **Could:** A work item that is desirable but not necessary for success
- **Won't:** A work item that the stakeholders have agreed to not implement now, but might be considered in a future release

Priority poker is another means by which priority may be assigned. Priority poker is similar to planning poker used for the estimation of work item effort. Planning poker is discussed in more detail in the *Estimating effort* recipe and so won't be discussed here.

This recipe outlines the user of a prioritization technique known as **Weighted Shortest Job First (WSJF)** as defined by the **Scaled Agile Framework (SAFe)**, see www.scaledagileframework.com/wsjf. The basic formulation is shown below.

$$WSJF = \frac{\text{Cost of delay}}{\text{Job Duration}}$$

The SAFe definition of the cost of delay is provided in the equation below. This equation differs from the original SAFe formulation by adding a project value term.

$$\begin{aligned} \text{Cost of delay} = & \text{Business value} + \text{Project value} + \text{Time criticality} \\ & + \text{Risk reduction or Opportunity enablement} \end{aligned}$$

Business value is either critical or useful to the stakeholders or some combination of the two. Project value, the term I added to the formula, refers to the value of the project. For example, the reduction of technical debt may not add direct value to the stakeholders but does provide value to the project. Time criticality, also known as *urgency*, refers to when the feature provides value to the stakeholder. Risk reduction is the improvement in the likelihood of project success, while opportunity enablement refers to business opportunities, such as new markets, that a feature will enable.

Each of the aspects of the cost of delay is scaled using values such as the Fibonacci sequence (1, 2, 3, 5, 8, 21, 34, 55, 89, 144, and so on) with larger values indicative of a higher cost of delay. Since these are all relative measures, the summation provides a good quantitative idea of the cost of delay. For a given job size, a higher cost of delay results in a higher priority. For a given cost of delay, a larger job size reduces the priority.

Job duration is difficult to estimate until you know the resource loading, so we normally substitute Job cost for Job duration. Job cost is the topic of the *Estimating effort* recipe. WSJF does a good first stab at determining priority, but it needs to be adjusted manually to take into account congruency with iteration missions and specialized resource availability. The workflow is outlined in *Figure 1.26*:

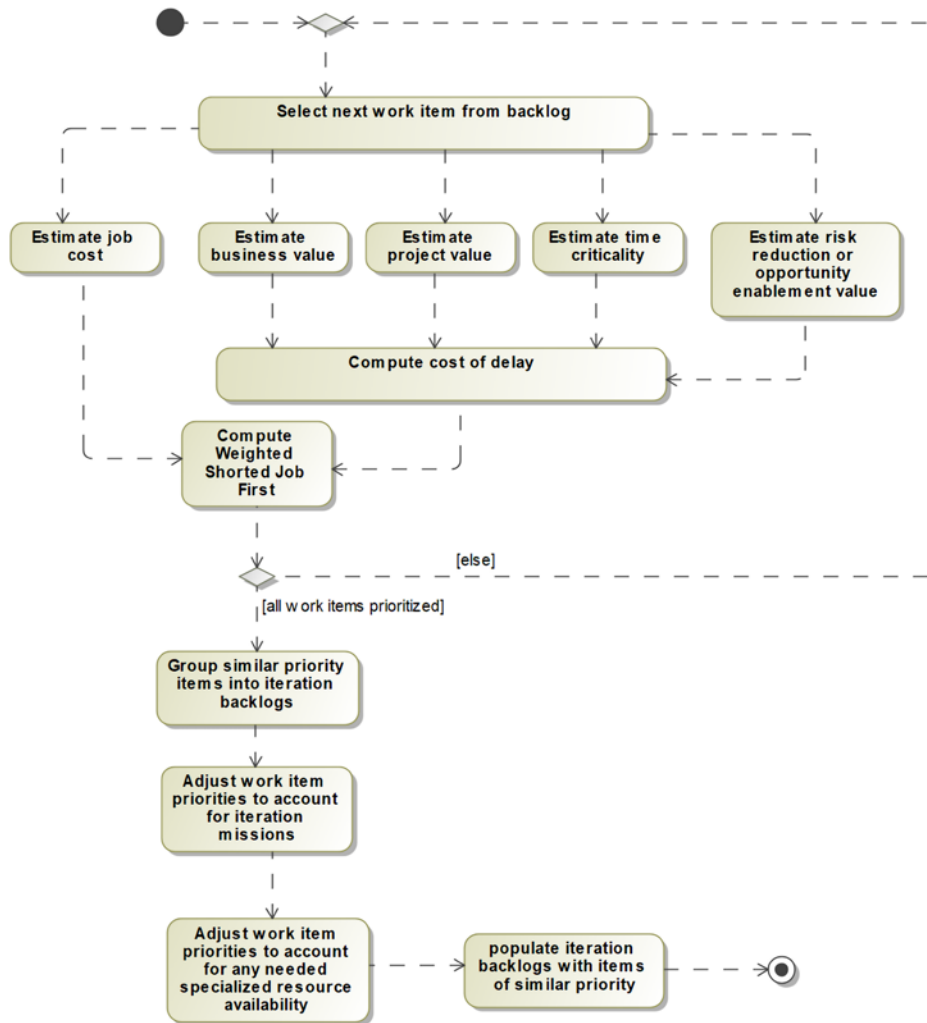


Figure 1.26: Work item prioritization

Select the next work item from the backlog

Starting with the first work item, select the work item to prioritize.

Estimate the job cost

Estimate the cost of performing the work item. The details of how do to this are discussed in the *Estimating effort* recipe.

Estimate the business value

Whether you are considering the criticality of the work item or its usefulness of the work item, or both, estimate its business value. For this and the other estimates contributing to the cost of delay, we are using relative measures with a Fibonacci sequence with the higher values corresponding to greater business value. Work items of similar business value should have the same value here.

Estimate the project value

The project value is the value that the completion of the work item brings to the project, such as the completion of a technical work item or paying down technical debt.

Estimate the time criticality

Estimate the time criticality of the work item, with more urgent work items having a higher value.

Estimate the risk reduction or opportunity enablement value

Estimate either the reduction of project risk or the enablement of business opportunity since the approach of the previous to steps. Greater risk reduction or greater opportunity means higher value.

Compute the Cost of Delay (CoD)

Compute CoD as the sum of the business value, project value, time criticality, and risk reduction.

Compute the weighted shortest job first

Compute WSJF as the cost of delay divided by the job cost.

Group similar priority items into the iteration backlog

The backlog for each iteration should contain elements of the same priority, depending on the availability of resources to perform the work. If there is capacity left over after allocating all elements of the same or similar priority, add work items from the next lowest priority. Similarly, if the accumulated cost of the set of work items of the same priority exceeds capacity, then move some to the next iteration backlog.

Adjust work item priorities to account for iteration missions

Examine the work items for congruence with the mission of the iteration. If there is no congruence, then is there another iteration where the work item is more in line with the iteration purpose? If so, adjust the priority to match that of the more relevant iteration.

Adjust work item priorities to adjust for any needed specialized resources available

Are there specialized resources needed for the completion of a work item? This might be the availability of a **Subject Matter Expert (SME)**, or the availability of computational or lab resources. Adjust the priority of work items to align with the availability of resources needed to accomplish the task.

Populate iteration backlogs with items of similar priority

Once the priorities have stabilized to account for all concerns, populate the iteration backlogs with the work items.

How it works

Prioritization is the ranking of elements on the basis of their desired sequencing. There are many means for prioritization with varying degrees of rigor. I prefer the WSJF approach because it takes into account most of the important aspects that should affect priority, resulting in a quantitative measure of the cost of delay divided by the size of the job.

Figure 1.27 shows a graph of WSJF isoclines. All curves show how the resulting value of WSJF changes as job size increases. Each separate curve represents a specific value for the cost of delay. You can see that the priority value diminishes rapidly as the size of the job grows. The practical effect of this is that higher-cost (i.e., higher-effort) tasks tend to be put off until later.

Just be aware that this is a bit problematic; since they require multiple iterations, there will be fewer iterations in which to schedule them:

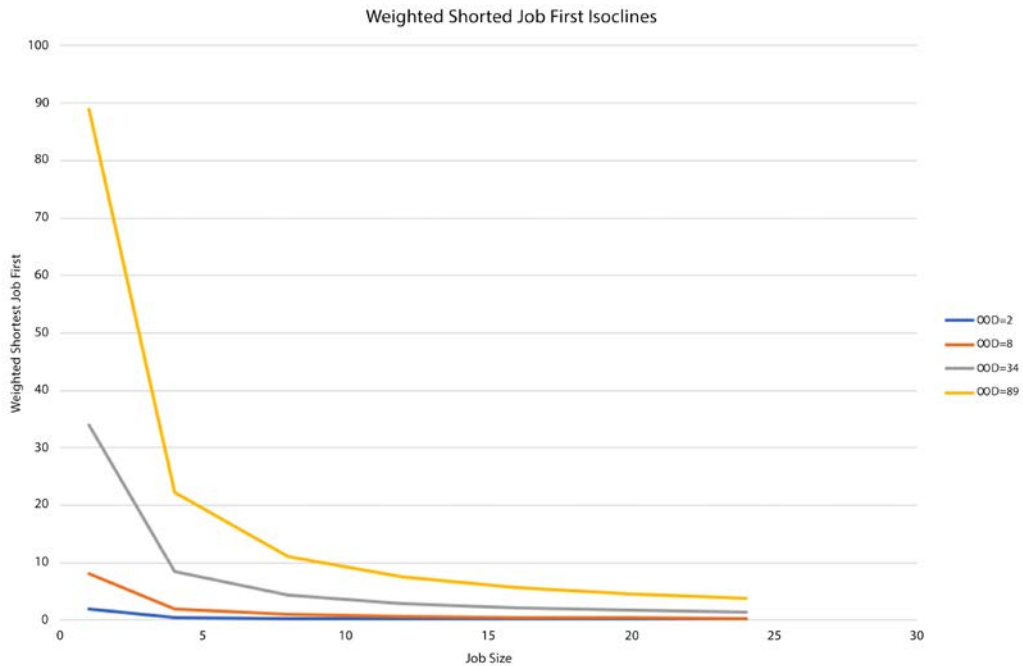


Figure 1.27: WSJF isoclines

While this method is recommended by the SAFe literature, in actual practice it must be modified so that you have congruence with the missions of the iterations. For example, it could happen that providing an encrypted message transfer has a high WSJF value while the creation of the base protocol stack is a lower value. Nevertheless, it makes no sense to work on the encryption design before you have a protocol in place over which the messages can be sent. Thus, you would likely raise the priority of the creation of the protocol stack and lower the priority of the encryption work to get “sensical sequencing.” Encryption math can be quite complex and if the encryption subject matter expert isn’t available for Iteration 6 but is available for Iteration 8, then it makes sense to adjust the priority of the encryption task to implement it when that expertise is available.

Example

Table 1.11 shows a worksheet that has a number of different kinds of work items, their previously estimated effort (job size), and the CoD terms. The spreadsheet sums up the terms to compute the **Cost of Delay (CoD)** column and then the WSJF shows the computed Weighted Shortest Job First.

The next column is the adjusted priority. This priority is generally the WSJF value but some of these are adjusted to move the work item into an appropriate iteration. The last column shows in which iteration a work item is planned to be resolved:

	Work item type			Cost of delay terms							Priority		
Epic	Work item use case	Work item user story	Spike or technical work item	User business value	Project value	Time criticality	RR OE	CoD	Job size (user story points)	WSJF	Priority	Planned Iteration	
			Spike: Team availability	1	55	1	21	78	2	39.00	39.00	0	
			Spike: Aggressive schedule	1	1	34	34	70	3	23.33	23.33	0	
			Spike: Agile MBSE impact	1	34	21	13	69	3	23.00	23.00	0	
Re-sist	Control resistance	Provide basic resistance		55	1	21	1	78	55	1.42	1.42	1	
			Spike: Motor response lag time	8	1	1	1	11	8	1.38	12.00	1	
			Spike: Robustness of main motor	34	1	1	34	70	5	14.00	14.00	1	

Set up physical bike	Set up bike fit	Adjust seat height		13	1	13	13	40	3	13.33	13.33	1
Set up physical bike	Set up bike fit	Adjust seat reach		13	1	8	13	35	3	11.67	11.67	2
		Calibrate power output		8	8	21	1	38	8	4.75	10.00	2
Emulate gearing	Emulate front and rear gearing			34	1	21	1	57	34	1.68	19.00	2
Emulate gearing	Emulate mechanical gearing			21	1	21	1	44	34	1.29	10.00	2
Emulate gearing	Emulate basic gearing			34	1	34	1	70	89	0.79	10.00	2
Set up physical bike	Manually adjust bike fit			34	1	21	1	57	13	4.38	4.38	3
Set up physical bike	Set up bike fit	Adjust handlebar height		8	1	1	3	13	3	4.33	4.33	3

Set up physical bike	Set up bike fit	Adjust handlebar reach		5	1	1	3	10	3	3.33	3.33	3
Monitor ride metrics		Monitor power		34	1	1	1	37	13	2.85	2.85	3
Monitor ride metrics		Monitor speed		21	1	1	1	24	5	4.80	4.80	3
Monitor ride metrics		Monitor distance		21	1	1	1	24	5	4.80	4.80	3
			Spike: USB robustness	21	8	1	8	38	5	7.60	7.60	3
Monitor ride metrics		Monitor cadence		8	1	1	1	11	5	2.20	2.20	4
Communicate with apps	Communicate with low-power Bluetooth			55	1	8	1	65	34	1.91	1.91	4

Set up physical bike	Set up bike fit	Select crank length		2	1	1	2	6	5	1.20	1.20	4
Re-sist	Control resistance	Set resistance under rider control		13	1	5	1	20	21	0.95	1.00	4
Configure bike for rider	Connect personal data to app			13	1	1	1	16	21	0.76	1.00	4
Set up physical bike	Estimate bike fit with external parameters	Compute fit from professional fit data		3	1	1	1	6	1	6.00	0.50	5
Monitor ride metrics		Monitor incline		13	1	1	1	16	8	2.00	0.50	5
Communicate with apps	Communicate with ANT+			34	1	5	1	41	34	1.21	0.50	5
Communicate with apps	Communicate with ANT FEC			34	1	13	1	49	55	0.89	0.89	5

Set up physical bike	Estimate bike fit with external parameters	Load GURU bike fit data		5	1	1	2	9	13	0.69	0.69	5
Set up physical bike	Estimate bike fit with external parameters	Load trek bike fit data		5	1	1	2	9	13	0.69	0.69	5
Re-sist	Control resistance	ERG mode		21	1	1	1	24	55	0.44	0.30	5
	Manage personal data			21	3	2	1	27	5	5.40	0.30	6
Set up physical bike	Predict bike fit with camera image	Access camera image		5	1	1	1	8	2	4.00	0.30	6
Set up physical bike	Predict bike fit with camera image	Retrieve road bike dimensions from camera image		8	1	1	1	11	5	2.20	0.30	6

Set up physical bike	Predict bike fit with camera image	Compute fit parameters from road bike dimensions		1	1	1	1	4	2	2.00	0.30	6
Resist	Control resistance	Set resistance under external control		21	1	5	1	28	39	0.72	0.30	6
Emulate gearing	Emulate DI2 gearing			13	1	5	1	20	55	0.36	0.30	6

Table 1.9: Prioritized work items

Iteration 0

Iteration 0 refers to the work done before incremental development begins. This includes early product planning, getting the development team started up and setting up their physical and tooling environment, and making an initial architectural definition. All this work is preliminary and most of it is expected to evolve over time as the project proceeds.

Purpose

The purpose of Iteration 0 is to prepare the way for the successful launch and ultimately the completion of the product.

Inputs and preconditions

The only inputs are initial product and project concepts.

Outputs and postconditions

By the end of Iteration 0, initial plans are in place and all that they imply for the product vision, the product roadmap, the release plan, and the risk management plan. This means that there is an initial product backlog developed by the end of Iteration 0, at least enough that the next few iterations are scoped out. Iterations further out may be more loosely detailed but, as mentioned, their content will solidify as work progresses. Additionally, the team is selected and enabled with appropriate knowledge and skills to do the work, their physical environment is set up, and their tools and infrastructure are all in place. In short, the engineering team is ready to go to develop the first increment and plans are in place to provide a project trajectory.

How to do it

Iteration 0 is “the work that takes place before there is any work to do.” That is, it is the preparatory work to enable the team to deliver the product.

There are four primary areas of focus:

Focus	Work to be done	Outputs
Product	Create an initial vision, product plan, and release plan	Product vision Product roadmap Release plan Risk management plan Initial product backlog
Team	Ready the team with knowledge, skills, tools, and processes	Assembled team
Environment	Install, configure, and test tooling and workspaces	Team environment set up
Architecture	Define the initial high-level architecture with expectations of technology and design approaches	Architecture 0

Table 1.10: Four primary areas of focus

It is important not to try for high precision. Most traditional projects identify a final release date with a finalized budget but these are in error. It is better to plan by successive approximation. Realize that early on, the error in long-range forecasts is high because of things you do not know and because of things you know that will change. As the project progresses, you gain knowledge of the product and the team’s velocity, so precision increases over time. These initial plans get the project started with a strong direction but also with the expectations that those plans will evolve.

It is important to understand that you cannot do detailed planning in Iteration 0 because you don't have a complete backlog, and you haven't yet learned all the lessons the project has to teach you. That doesn't mean that you shouldn't do any planning; indeed, four of the outputs – the product vision, the product roadmap, the release plan, and the risk management plan – are all plans. However, they are all incorrect to some degree or another, and those plans will require significant and ongoing modification, enhancement, and evolution. This is reflected in the Law of Douglass #3 (<https://www.bruce-douglass.com/geekosphere>):



Plan to re-plan.

Law of Douglas #3

We discussed earlier in this chapter the product roadmap, release plan, and risk management plan. Their initial preparations are the key ingredients of Iteration 0. The workflow for Iteration 0 is shown in *Figure 1.28*:

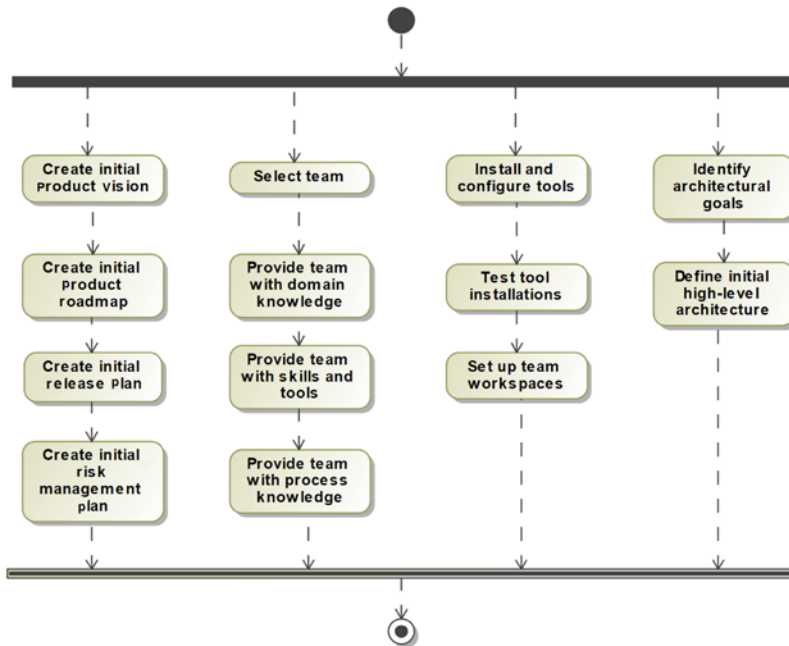


Figure 1.28: Iteration 0

Create an initial product vision

The product vision is a high-concept document about the product scope and purpose and the design approach to meet that purpose. It combines the company's business goals with the specific needs of the customer. It identifies how this product will differentiate itself from competing products and clarifies the value to both the company and the customers.

Create an initial product roadmap

The product roadmap is a strategic plan that defines how the product will evolve over time. See the *Product roadmap* recipe in this chapter for more detail.

Create an initial release plan

The release plan is a tactical plan for how features will be developed in the next several product iteration cycles. The *Release plan* recipe discusses this in more detail.

Create an initial risk management plan

The risk management plan is a strategic plan that identifies project risks and how and when they will be addressed by allocating spikes (experiments) during the iteration cycles. See the *Managing risk* recipe for information on this plan.

Select the team

The team is the set of people who will collaborate on the development of the product. This includes engineers of various disciplines (systems, software, electronics, and mechanical, typically), testers, configuration managers, integrators, a product manager, and a process lead, sometimes known as a *scrum master*. There may be specialized roles as well as a safety czar, reliability czar, security czar, biomedical engineer, or aerospace engineer, depending on the product.

Provide the team with domain knowledge

Unless the team has prior experience in the domain, it will probably be useful to expose the team to the customer domain concepts and concerns. This will enable them to make better choices.

Provide the team with skills and tools

Any new technology, such as the use of Java or SysML, should be preceded by training and/or mentoring. The introduction of new tools, such as Jira for project tracking or Cameo Systems Modeler for SysML modeling, should likewise involve training.

Provide the team with process knowledge

The team must understand the procedures and practices to be employed on the project to ensure good collaboration. The recipes in this book identify many such practices. The project may also employ a process that incorporates a set of practices, such as the Harmony aMBSE or OOSEM processes.



See *Agile Systems Engineering* by Bruce Powel Douglass, at <https://www.amazon.com/Agile-Systems-Engineering-Bruce-Douglass/dp/0128021209> and <https://www.incose.org/incose-member-resources/working-groups/transformational/object-oriented-se-method> for further information.

Install and configure tools

The tooling environment should be set up and ready for the team to use. This might include project enactment tools such as Jira in addition to modeling tools, compilers, editors, and so on.

Test tool installations

This step verifies that the tools are properly installed and the infrastructure for the tools works. This is especially important in collaborative environments such as team clouds.

Set up team workspaces

This action refers to the physical and virtual workspaces. It is common to co-locate teams where possible and this ensures that the teams have spaces where they can do individual “thought work,” as well as collaborative spaces where they can work together.

Identify architectural goals

Architecture, as we will see in the recipe *Architecture 0*, is the set of large-scale product organization and design optimization decisions. Goals for architecture are often focused on simplicity, understandability, testability, stability, extensibility, robustness, composability, safety, security, and performance. Frequently these properties are in conflict; something easy to understand may not be scalable, for example. Thus, the architectural goals identify the relative importance of the goals with respect to the success of the project and the product.

Define the initial high-level architecture

This action defines the high-level architecture, congruent with the architectural goals identified in the previous step. This is known as Architecture 0, the subject of the recipe *Architecture 0*.

Example

For the example problem outlined in *Appendix A*, the road map, release plan, risk management plan, and Architecture 0 are developed in other recipes in this chapter and need not be repeated here. The other aspects are discussed here.

Create an initial product roadmap

The initial product vision and roadmap are discussed in more detail in the Product roadmap recipe.

Create the initial release plan

The release plan is discussed in more detail in the Release plan recipe.

Create the initial risk management plan

The risk management plan is discussed in more detail in the Managing risk recipe.

Select the team

In our project, we select the systems, software, electronic, and mechanical engineers for the project. The team consists of three systems engineers, two mechanical engineers, three electronics engineers, and 10 software engineers. They will all be co-located on the fourth floor of the company's building, except for Bruce who will be working from home and come in when necessary. Each will have an individual office and there are two conference rooms allocated to the team.

Provide the team with domain knowledge

To provide the team with domain understanding, we bring in SMEs to discuss how they train themselves and others. The SMEs include professional cyclists, personal trainers, amateur cyclists, and triathletes. Members of the focus group lead the team through some workouts on existing trainers to give them an understanding of what is involved in different kinds of training sessions. Some classwork sessions are provided as well to give the team members a basic understanding of the development and enactment of training plans, including periodization of training, tempo workouts versus polarized training, and so on.

Install and configure tools

In addition to standard office tools, several engineering tools are installed and configured for the project:

- Systems engineers will use DOORS for requirements, Cameo Systems Modeler for SysML, and Groovy for simulation along with the Teamwork Cloud for configuration management.
- Mechanical engineers will use AutoCAD for their mechanical designs.
- Electronic engineers will use SystemC for discrete simulation and Allegro Cadence for their designs.
- Software engineers will use Rhapsody for UML and code generation and Cygwin for C++, along with the Rhapsody Model Manager.
- The collaboration environment will use the Jazz framework with Rational Team Concert for project planning and enactment.

Test tool installations

The IT department verifies that all the tools are properly installed, and can load, modify, and save sample work products from each. Specific interchanges between DOORS, Cameo Systems Modeler, Rhapsody, and both the Cameo Teamwork Cloud and Rhapsody Model Manager can successfully store and retrieve models.

Provide the team with skills and tools

- System engineers will receive week-long training on Cameo Systems Modeler and SysML.
- Software engineers will receive week-long training on Rhapsody and UML.
- All engineers have used DOORS before and require no additional training.

Provide the team with process knowledge

The team will use the Harmony aMBSE process and attends a 3-day workshop on the process. In addition, *A Priori Systems* will provide agile and modeling mentoring for the team through at least the first four iterations.

Set up team workspaces

Systems engineers are provided with a configuration computer environment that includes installed Cameo Systems Modeler, DOORS, the company's network, and Cameo Teamwork Cloud.

Software engineers are provided with a configured computer connected to the company's network, and can connect to the local Jazz team server to access the Jazz tooling – Rhapsody, DOORS Next Generation, Rational Team Concert, and Rhapsody Model Manager.

Define initial high-level architecture

Both of these actions are discussed in more detail in the recipe *Architecture 0*.

Architecture 0

Architecture is the set of strategic design optimization decisions for a system. Many different architectures can meet the same functional needs. What distinguishes them is their optimization criteria. One architecture may optimize worst-case performance, while another may optimize extensibility and scalability, and yet another may optimize safety, all while meeting the same functional needs.

The Harmony process has two primary components: the **Harmony Agile Model-Based Systems Engineering process (Harmony aMBSE)** and the **Harmony for Embedded Software process (Harmony ESW)**. They each describe workflows, work products, practices, and guidance for combining agile and model-based engineering in their respective disciplines. See the author's *Real-Time Agility* book for more details.

The Harmony process identifies five key views of architecture.

Subsystem and component view

This view focuses on the largest scale pieces of the system, and their organization, relations, responsibilities, and interfaces.

Concurrency and resource view

This view focuses on the concurrency units and management of resources within the system. Processes, tasks, threads, and the means for safely sharing resources across those boundaries are the primary concerns of this view.

Distribution view

This view focuses on how collaboration occurs between different computational nodes within the system and how the subsystems share information and collaboration. Communication protocols and middleware make up the bulk of this view.

Dependability view

The three pillars of dependability are freedom from harm (safety), the availability of services (reliability), and protection against attack (security).

Deployment view

Generally, subsystems are interdisciplinary affairs, consisting of some combination of software, electronic, and mechanical aspects. This view is concerned with the distribution of responsibility among the implementation of those disciplines (called facets) and the interfaces that cross engineering disciplinary boundaries.

Some recommendations for architecture in agile-developed systems are shown in *Figure 1.29*:

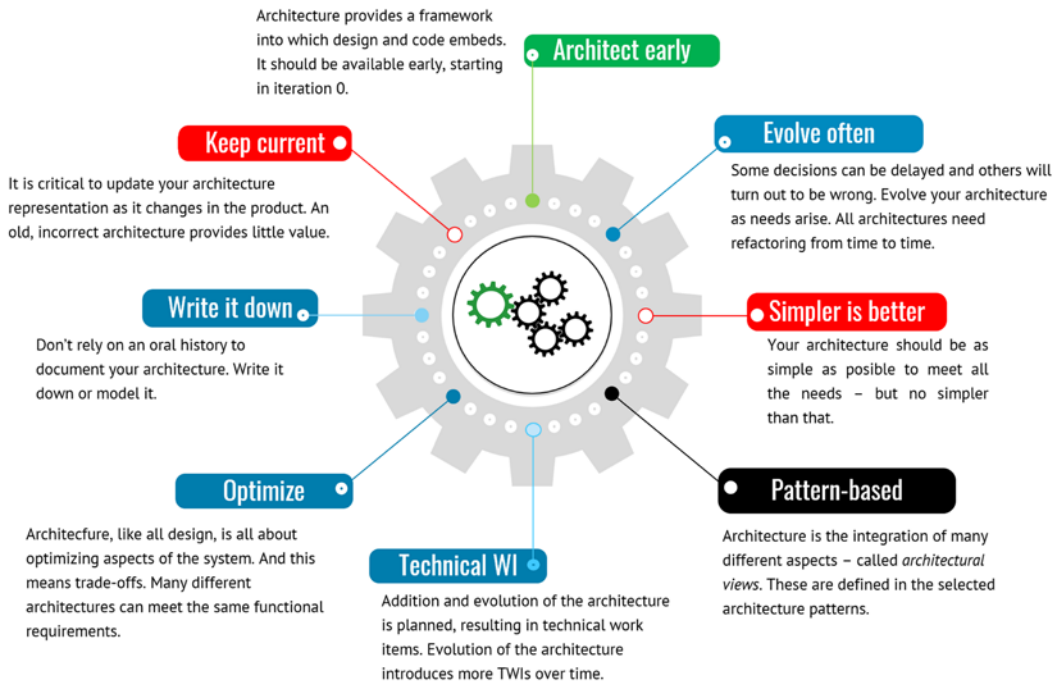


Figure 1.29: Agile architectural guidelines

Purpose

Because architecture provides, among other things, large-scale organization of design elements, as engineers develop those design elements, Architecture 0 provides a framework into which those elements fit. It is fully expected that Architecture 0 is minimalist, and therefore incomplete. It is expected that the architecture will change and evolve through the development process but it is an initial starting point.

Inputs and preconditions

A basic idea of the functionality and use of the system is necessary to develop the initial architectural concept. Thus, the preconditions for the development of Architecture 0 are the product vision and at least a high-level view of the epics, use cases, and user stories of the system.

Outputs and postconditions

The output is a set of architecture optimization criteria and an initial set of concepts from the different architectural views. This may be textual, but I strongly recommend this being in the form of a SysML architectural model. This model may have a number of different diagrams showing different aspects of the architecture. It is common, for example, to have one or more diagrams for each architectural view. In Architecture 0 many of these will be missing and will be elaborated on as the project proceeds.

How to do it

Architecture 0 is an incomplete, minimalist set of architectural concepts. Some thought is given to architectural aspects that will be given later, if only to assure ourselves that they can be integrated smoothly when they are developed. Again, it is expected that the architecture will be elaborated, expanded, and refactored as the product progresses.

Figure 1.30 shows the basic workflow.

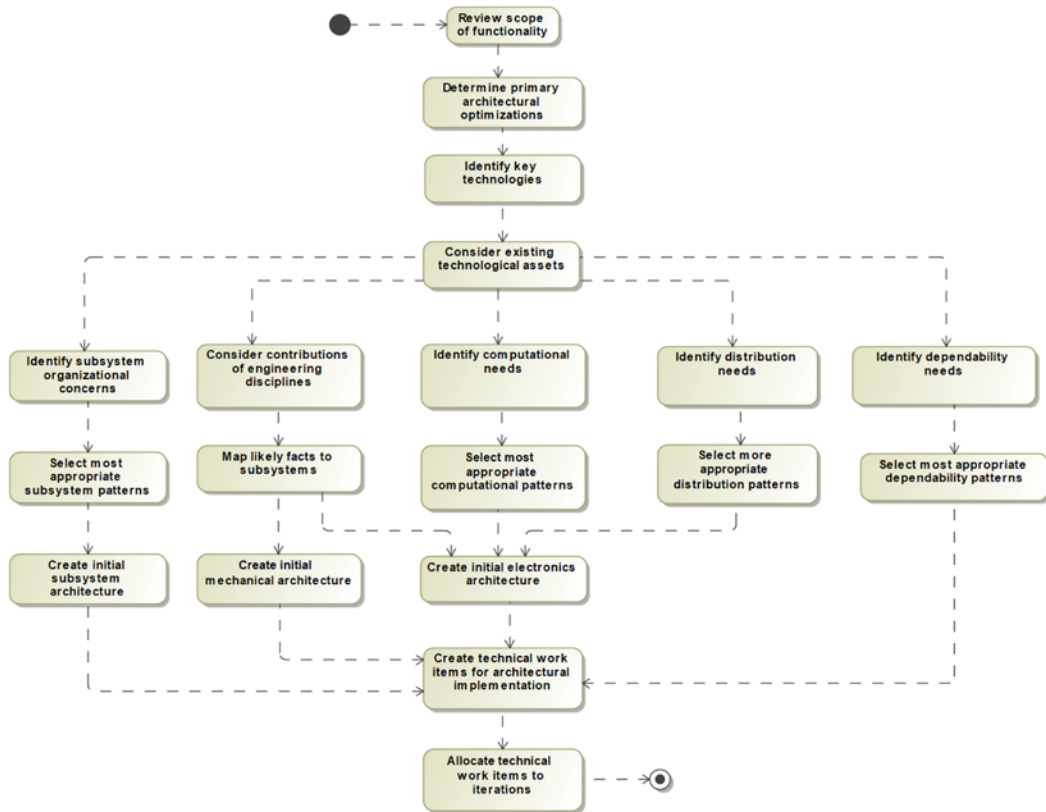


Figure 1.30: Architecture 0

Review the scope of functionality

Architectures must be fit for purpose. This means that while architectural decisions are largely about optimizations, the architecture must, first and foremost, achieve the functional needs of the system. This step reviews the essential required functionality that must be supported by the architecture.

Determine primary architectural optimizations

Selecting a design is an exercise in balancing competing optimization concerns making up the primary set of architectural goals. This step identifies and ranks the most important architectural considerations, such as worst-case performance, average performance, bandwidth, throughput, scalability, extensibility, maintainability, manufacturability, testability, and certifiability, to name a few.

Identify key technologies

It is common that one or more key technological ideas dominate the vision of the product. Electric cars, for example, have electric motors and electronic means to power them. While not necessarily defining the solutions here, it is important to at least identify the key technologies to constrain the solution space.

Consider existing technologies assets

Unless this is the very first time an organization has developed a similar product, there is likely some “prior art” that should be considered for inclusion in the new product. The benefits and costs can be considered as employing the organization’s existing technological intellectual property versus creating something entirely new.

Identify subsystem organizational concerns

Subsystems are the largest-scale pieces of the system and thus serve as the primary organization units holding elements of designs from downstream engineering. This step considers the pros and cons of different subsystem allocations and organizations. A good set of subsystems are:

- Coherent – provide a small number of services
- Internally tightly coupled – highly codependent elements should generally reside in the same subsystem
- Externally loosely coupled – subsystems should stand on their own with their responsibilities but collaborate in well-defined ways with other subsystems
- Collaborative with interfaces – interact with other subsystems in well-defined ways with a small number of interfaces

Consider contributions of engineering disciplines

The aspects of a design from a single engineering discipline is called a *facet*. There will typically be software facets, electronic facets, mechanical facets, hydraulic facets, pneumatic facets, and so on. The set of facets and their interactions are known as the *deployment architecture*, an important view of the system architecture. Early on, there may be sufficient information to engage engineers in these disciplines and consider how they are likely to contribute to the overall design.

Identify computational needs

Computational needs affect both software and electronics disciplines. If the system is an embedded system – the primary case considered in this book – then the computational hardware must be selected or developed with the particular system in mind.

These decisions can have a huge impact on performance and the ability of the software to deliver computational functionality. The software concurrency architectural concerns are not considered here, as they are solely a software concern. Nevertheless, the system must have adequate computational resources, and early estimates must be made to determine the number and type of CPUs and memory.

Identify distribution needs

Networks such as 1553 or CAN buses and other connection needs, as well as possible middleware choices including AUTOSAR, CORBA, and DDS, are the focus of this step.

Identify dependability needs

This step is crucial for safety-critical, high-reliability, or high-security systems. The initial concepts for managing dependability concerns must be considered early for such high-dependability systems and may be saved for later iterations in systems in which these are a minimal concern.

Select the most appropriate subsystem patterns

There are many organizational schemes for subsystem architecture, such as the layered pattern, microkernel pattern, and channel pattern, that provide different optimizations. See the author's book *Real-Time Design Patterns* for more detail on these patterns. The definition and use of patterns are discussed in more detail in *Chapter 3, Developing Systems Architecture*.

Map likely facets to subsystems

Facets are the contributions to an overall design from specific engineering disciplines, such as software, electronics, and mechanical engineering. We recommend that subsystem teams are interdisciplinary and contain engineers from all relevant disciplines. This step is focused on early concept deployment architecture.

Select the most appropriate computational patterns

Computational patterns concentrate on proposed computational approaches. While largely a software concern, electronics play a key role in delivering adequate computation power and resources. This is especially relevant when the computation approach is considered a key technology, as it is for autonomous learning systems or easy-to-certify cyclic executives for safety-critical systems.

Select the most appropriate distribution patterns

There are many ways to wire together distributed computing systems with networks, buses, and other communication links, along with supporting middleware. This architectural view focuses on that aspect of the system design. This impacts not just the software, but the electronic and, to a lesser degree, mechanical designs.

Select the most appropriate dependability patterns

Different patterns support different kinds of optimizations for safety, reliability, and security concerns. If these aspects are crucial, they may be added to Architecture 0 rather than leaving them for later design. Deciding to “make the product safe/reliable/secure” late in the development cycle is a recipe for project failure.

Create initial subsystem architecture

This aspect is crucial for the early design work so that the design elements have a place to be deployed. Subsystems that are not needed for early iterations can be more lightly sketched out than ones important for the early increments.

Create initial mechanical architecture

The initial mechanical architecture provides a framework for the development of physical structures, wiring harnesses, and moving mechanical parts.

Create initial electronic architecture

The initial electronics architecture provides a framework for the development of both analog electronics such as power management, motors, and actuators, as well as digital electronics, including sensors, networks, and computational resources.

Create technical work items or architectural implementation

A skeletal framework for the architecture is provided in Architecture 0 but beyond this, architectural implementation work results in technical work items that are placed in the backlog for development in upcoming iterations.

Allocate technical work items to iterations

Initial allocation of the technical work items is done to support the product roadmap and, if available, the release plan. These elements may be reallocated later as the missions of the iterations evolve.

Example

Review the scope of functionality

The Pegasus is a high-end smart cycling trainer that provides fine-grained control over bike fit, high-fidelity simulation of road feel, structured workouts, and interactions with popular online training apps. Read the content of *Appendix A* to review the functionality of the system.

Determine primary architectural optimizations

The primary optimization concerns in this example are determined to be:

- Recurring cost – the cost per shipped item
- Robustness – maintenance effort and cost-of-ownership should be low
- Controllability – fine-grained control of power over a broad range
- Enhance-ability – the ability to upgrade via Over-The-Air to add new sensors, capabilities, and training platforms is crucial

Identify key technologies

There are a number of key technologies crucial to the acceptance and success of the system:

- Low-energy Bluetooth (BLE) Smart for interacting with common sensors and app-hosting clients (Windows, iPad, iPhone, and Android)
- ANT+ for connecting to common sensors
- IEEE 802.11 wireless networking
- An electronic motor to provide resistance

These technologies are considered essential in this case, but we do want to take care not to overly constrain the design solution so early in the product cycle.

Consider existing technologies assets

This is a new product line for the company and so there are no relevant technological assets.

Identify subsystems organizational concerns

To improve manufacturability, we want to internalize cabling as well as minimize the number of wires. This means that we would like to co-locate the major electronics components to the greatest degree possible. However, user controls must be placed within convenient reach. Care must also be taken for adequate electric shock protection as the users are likely to expose the system to corrosive sweat.

Select the most appropriate subsystem patterns

We select the *Hierarchical Control Pattern* and *Channel Pattern*, from *Real-Time Design Patterns*, Addison-Wesley by Bruce Powel Douglass, 2003, as the most applicable for our systems architecture.

Create an initial subsystem architecture

Figure 1.31 shows the operational context of the Pegasus indoor training bike. This is captured in a **Block Definition Diagram (BDD)** in the architectural design package of the model. The operational context defines the environmental actors with which the architecture must interact during system operation:

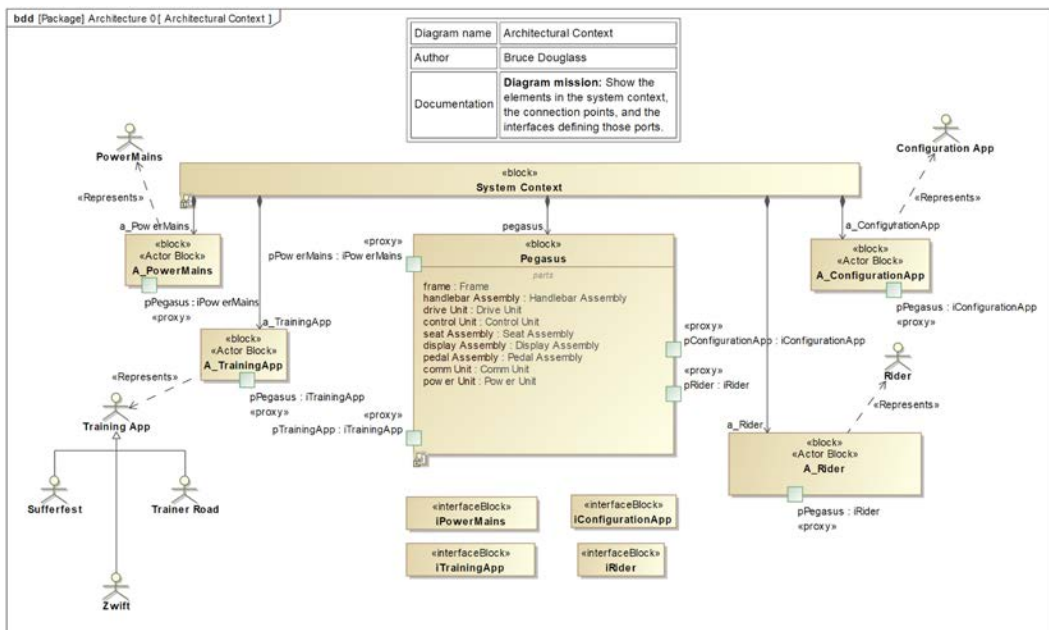


Figure 1.31: Pegasus context diagram

Figure 1.31 shows the elements in the **Pegasus** context. Note that we used blocks with the stereotype «Actor Block», each with a «represents» dependency to the actor they represent. This is done because in Cameo actors cannot have ports, and we wish to use ports to specify the interfaces used in the system context. These stereotypes are not defined in Cameo and so must be added in a user-created profile within the project.

Figure 1.32 shows how these elements connect in an internal block definition diagram owned by the **System Context** block:

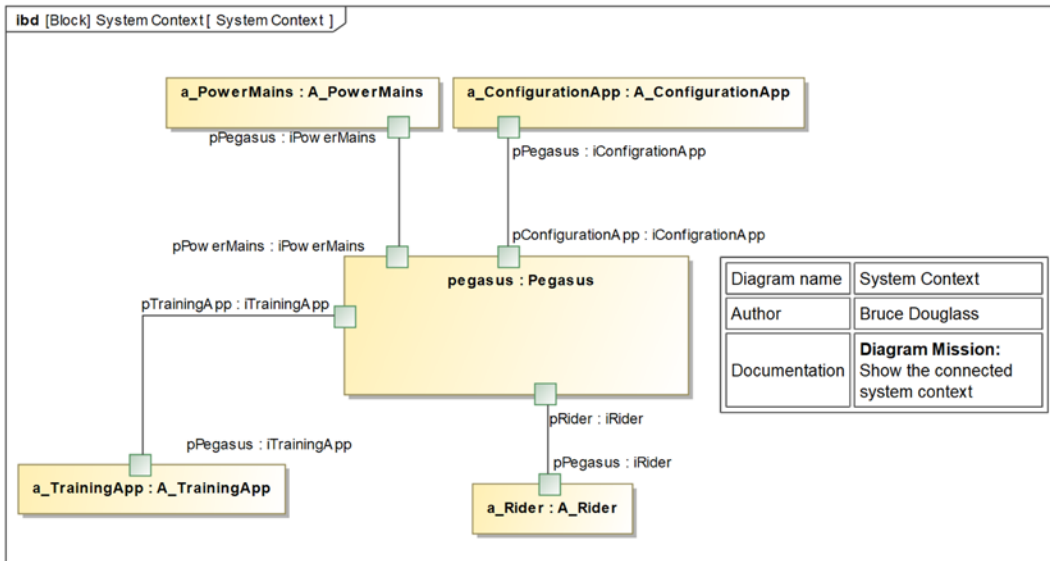


Figure 1.32: Pegasus connected context

Next, Figure 1.33 shows the set of subsystems. This diagram is like a high-level parts list and is very common in system designs in SysML:

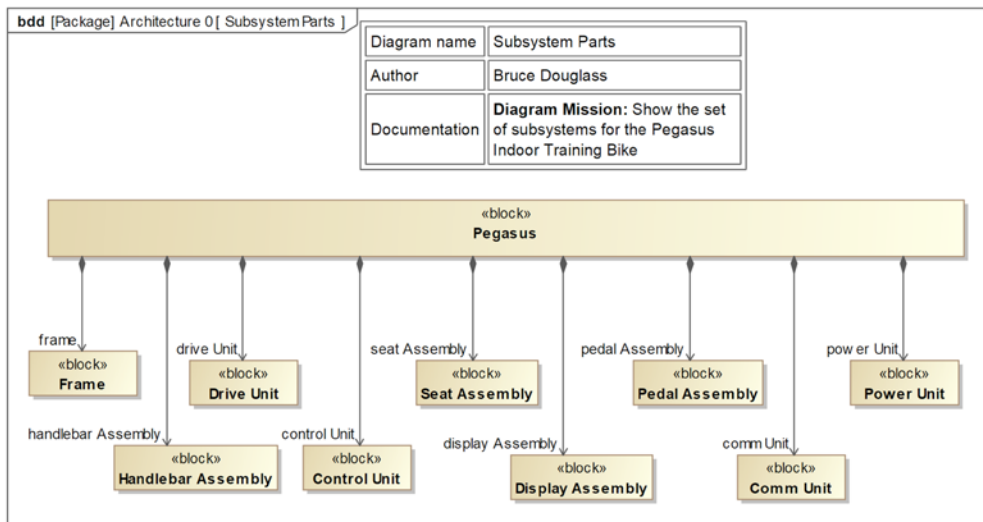


Figure 1.33: Pegasus subsystems

Perhaps more interesting is *Figure 1.34*, which shows how the subsystems connect to each other in an **Internal Block Diagram (IBD)**. This is also a commonly used architectural view in SysML models. Specifically, this figure shows the primary functional or dynamic interfaces.

I follow a convention in my architectural models in which dynamic connections – that is, ones that support runtime continuous or discrete flow – use ports, but static connections – such as when parts are bolted together – as shown using connectors with the «static» stereotype. I find this a useful visual distinction in my systems architecture diagrams. Thus, the relation between the **Frame** and the **Drive Unit** is an association but the relation between the **Pedal Assembly** and the **Drive Unit** employs a pair of ports, as there are runtime flows between the pedals and the drive motor during system operation.

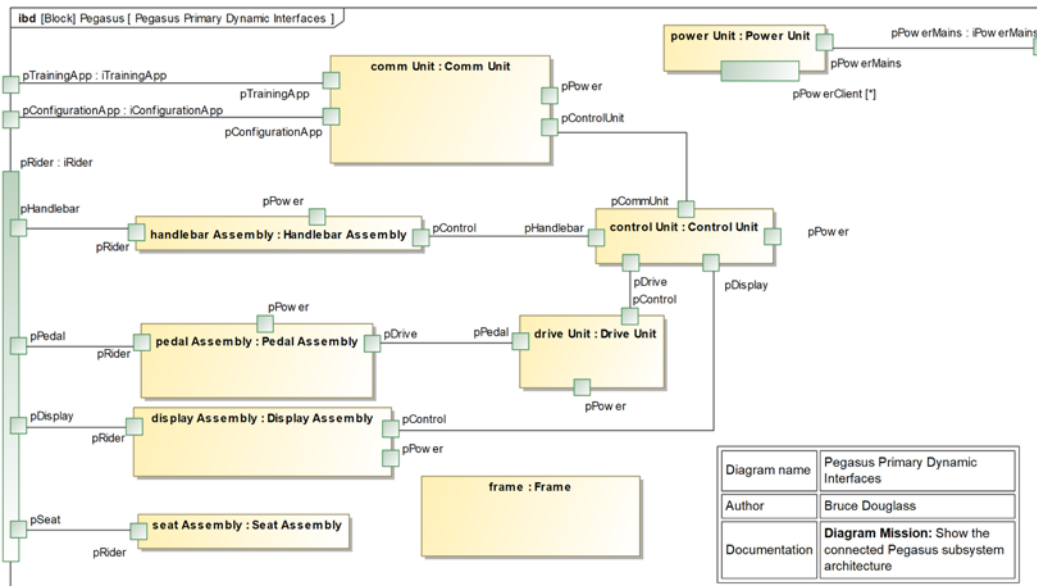


Figure 1.34: Pegasus connected architecture – Primary dynamic interfaces

Consider contributions of engineering disciplines

The electronics will provide the interfaces to the user for control of the trainer as well as physical protocols for app communication. It will also provide resistance via the application of torque from the electronic motor.

The mechanical design will provide the bike frame and all the bike fit adjustments. The pedals will accept torque from the user and provide resistance to applied force via the crank arms. Additionally, the weighted flywheel smooths out resistance by providing inertial load.

Finally, the mechanical design will provide all the cable routing.

The software will provide the “smarts” and use electronics to receive and process user input from controls, inputs from connected apps, as well as from the pedals. The software will also be responsible for messages to the apps for measured sensor data.

Map likely facets to subsystems

Facets, you will remember, are the contributions engineering disciplines provide to the system design. *Table 1.11* shows the initial concept for mapping the engineering facets to the subsystem architecture. This will result, eventually, in a full deployment architecture, but for now, it just highlights our current thinking about the work from the engineering disciplines that will map to the subsystems.

Subsystem	Mechanical	Electronics	Software
Frame	Mechanical only		
Handlebar assembly	Mechanical only		
Drive unit	Housing for motor, fly-wheel, and drive train	Motor electronics	Discrete outputs from control unit SW controlled
Control unit	Cabling and mounting	Primary CPU, memory, and electronic resources for SW, persistence storage for SW	Control motor, process incoming sensor and power data, process communications with apps, and Over-The-Air updates
Seat assembly	Mechanical only		
Display assembly	Cabling and mounting	Display and buttons, and USB connectors (future expansion)	User I/O management and USB interface support
Pedal assembly	Crank arms, LOOK-compatible shoe mount, connects to drive train	Power sensor in the pedal assembly	Discrete inputs to SW in control unit
Comm unit	Cabling and mounting	802.11, Bluetooth (BLE) Smart, and ANT+	SW in control unit controls and mediates communications from sensors and external apps
Power unit	Cabling and mounting	Converts wall power to internal power and distributes where needed	

Table 1.11: Initial deployment architecture

Identify computational needs

At a high level, the primary computational needs are to:

- Receive and process commands to vary the resistance either from the internal load, user input, or app control
- Actively control the resistance provided by the motor
- Monitor connected Bluetooth and ANT+ sensors, such as from heart-rate straps
- Send sensor data to connected apps
- Update the user display when necessary
- Perform Over-The-Air updates

Select the most appropriate computational patterns

An asymmetric dual-processor architecture is selected with a single primary CPU in the Control Unit and a secondary processor to manage communications.

Identify distribution needs

All communications needs will be performed by a (proposed) 16-bit communication processor housed in the comm unit; all other SW processing will be performed by the control unit on the primary CPU.

Select the most appropriate distribution patterns

To facilitate the timely distribution of information within the system, an internal serial interface, such as RS232, is deemed adequate.

Identify dependability needs

The primary safety concern is electric shock caused by faulty wiring, wear, or corrosion. Reliability needs to focus on resistance to corrosion due to sweat, and the durability of the main drive motor. Security is determined to not be a significant concern.

Select the most appropriate dependability patterns

Single-Channel Protected Pattern is selected as the most appropriate.

Create initial electronic architecture

The power interfaces are shown in *Figure 1.35*:

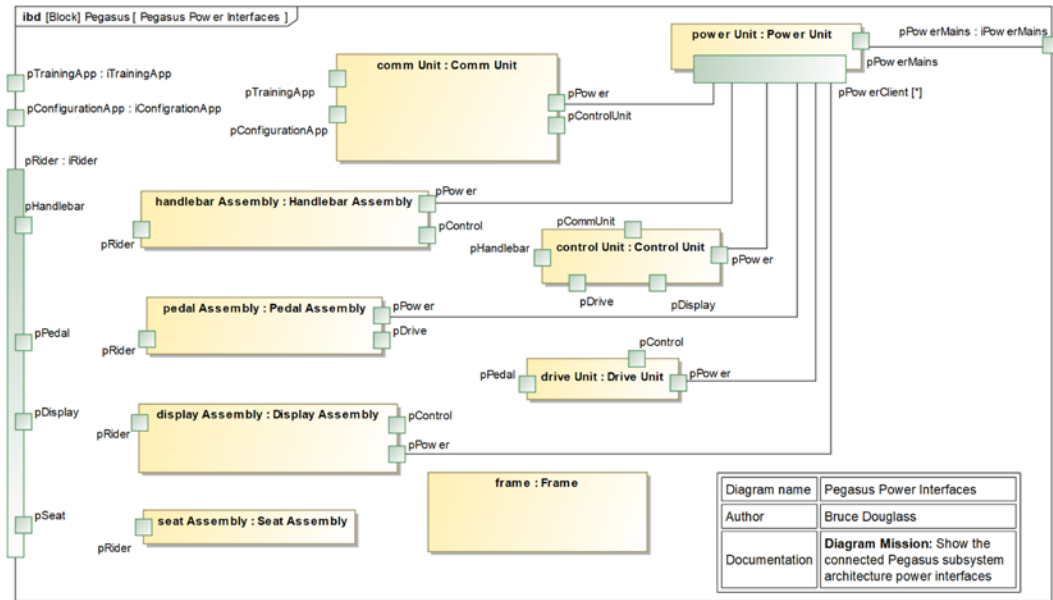


Figure 1.35: Pegasus connected architecture – Power interfaces

In addition, the drive unit hosts the primary motor to provide resistance to pedaling.

Create initial mechanical architecture

The internal mechanical connections are shown in *Figure 1.36*:

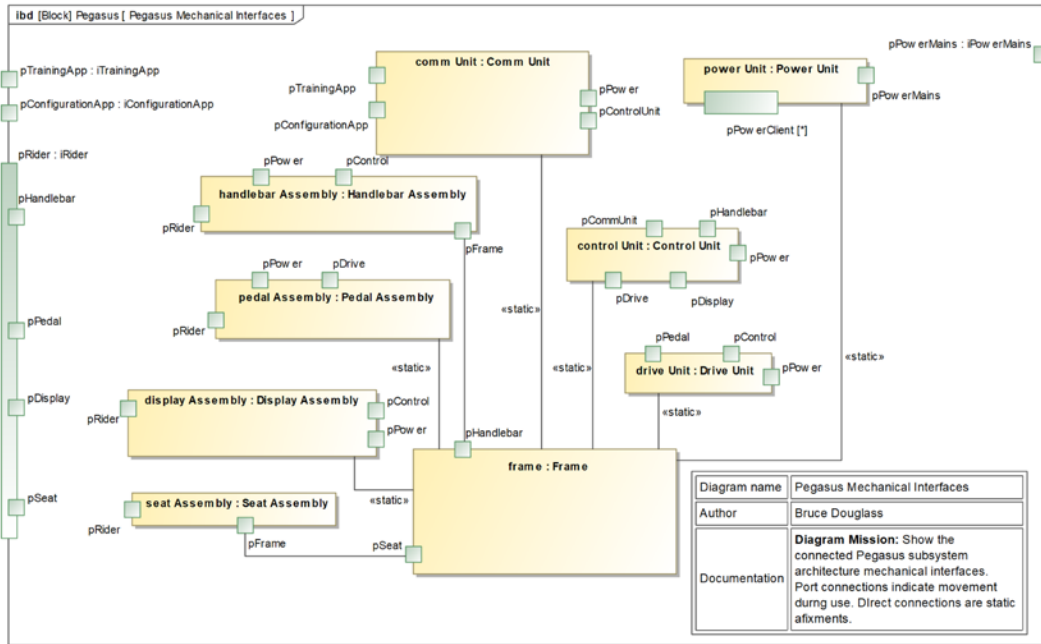


Figure 1.36: Pegasus connected architecture – Mechanical interfaces

The seat and handlebars are adjustable by the user and pedals can be added or removed, as well as providing a spot for the rider to “clip in” when they get on the bike. Thus, these connections use ports. The other connections are places where pieces are bolted together and do not vary during operation, except under catastrophic mechanical failure, so connectors represent these connections.

Create technical work items for the architectural implementation

We identify the following technical work items in Architecture 0:

Mechanical technical work items:

- CAD frame design
- Hand-built frame
- Factory built frame

Electronic technical work times:

- Motor simulation platform
- Hand-built motor
- Factory-built motor electronics
- CPU selection
- Simulated digital electronics platform
- Hand-built digital electronics platform
- Factory-built digital electronic platform

Allocate technical work items to the iterations

Allocation of such technical work items is discussed in detail in the Iteration plan recipe.

Additional note

Note that this is not a fully fleshed-out architecture. We know that in principle, for example, the pedals and the drive unit have flows between them because the pedals must provide more resistance when the main motor is producing greater resistance, but we don't know yet if this is a mechanical power flow perhaps via a chain, an electronic flow, or discrete software messages to active resistance motors at the site of the pedals. I added a power interface to the **Pedal assembly**. If it turns out that it isn't needed, it can be removed. That's a job for later iterations to work out.

I also didn't attempt to detail the interfaces – again, deferring that for later iterations. But I did add the ports and connectors to indicate the architectural intent.

Organizing your models

Packages are the principal mechanism for organizing models. In fact, a *model* is just a kind of package in the underlying SysML metamodel. Different models used for different purposes are likely to be organized in different ways. This recipe focuses on systems' engineering models to specifically support requirements capture, use case and requirements analysis, architectural trade studies, architectural design, and the hand-off of relevant Systems Model data to the subsystem teams. In this recipe, we create not only the systems model but also the federation of models used in systems engineering. Federations, in this context, are sets of interconnected models with well-defined usage patterns defined.

Purpose

Organizing well is surprisingly important. The reasons why include:

- Grouping information to facilitate access and use
- Supporting concurrent model use by different team members performing different tasks
- Serving as the basis for configuration management
- Allowing for relevant portions of models to be effectively reused
- Supporting team collaboration on common model elements
- Allowing for the independent building, simulation, and verification of model aspects

Inputs and preconditions

The product vision and purpose are the primary input for the systems engineering model. The hand-off model is created after the architecture is stable for the iteration and is the primary input for the subsequent Shared Model and Subsystem Models.

Outputs and postconditions

The outputs are the shells of the Systems, Shared, and Subsystem Models, a federation of models for downstream engineering. The Systems Model is populated with the system requirements, if they exist. The Shared Model is initially populated with (references to) the system requirements, (logical) system interfaces and (logical) data schema from the Systems Model. A separate Subsystem Model is created for each subsystem and is initially populated with a reference to its subsystem requirements from the Systems Model, and the physical interfaces and data schema from the Shared Model.

How to do it

Figure 1.37 is the workflow for creating and organizing the Systems Model. In the end you'll have logical places for the MBSE work, and your team will be able to work more or less independently on their portions of the effort:

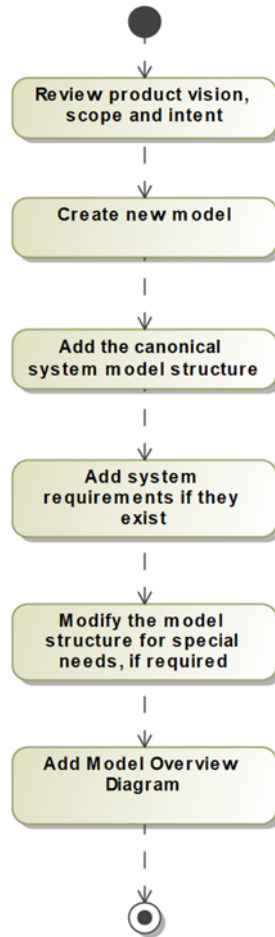


Figure 1.37: Organizing the systems engineering model

Review product vision, scope, and intent

Every product has a vision that includes its scope and intent. Similarly, every model you create has a scope, purpose, and level of precision. A model's scope determines what goes into the model and what is outside of it. A common early failure of modeling happens when you don't have a well-defined purpose for a model.

The purpose of the model is important because it determines the perspective of the model. George Box is famously attributed to the quote “All models are wrong, but some are useful.” What I believe this means is that *every model is an abstraction of reality*. A model fundamental represents information useful to the purpose of the model and ignores system properties that are not useful. The purpose of the model, therefore, determines what and how aspects of the product and its context will be represented.

The level of precision of the model is something often overlooked but is also important. It is too often that you will see requirements such as:



The aircraft will adjust the rudder control surface to ± 30 degrees.

That’s fine as far as it goes, but what does it mean to achieve a value of +15 degrees? Are 14.5 degrees close enough? How about 14.9? 14.999? How quickly will the position of the rudder be adjusted? If it took a minute, would that be OK? Maybe 1.0 seconds? 100ms? The degree to which you care about how close is “close enough” is the precision of the model, and different needs have different levels of required precision. This is sometimes known as “model fidelity.” The bottom line is to know *what* you are modeling, *why* you are modeling it, and *how close to reality* you must come to achieve the desired value from your model. In this case, the model is of the requirements, their analyses, supporting analyses leading to additional requirements, and the architecture structure.

Create a new model

In whatever tool you use, create a blank, empty model.

Add the canonical system model structure

This is the key step for this recipe. The author has consulted for decades to literally hundreds of systems engineering projects, and the Subsystem Model organization shown in *Figure 1.38* has emerged as a great starting point. You may well make modifications, but this structure is so common that I call it the **system canonical organization**.

It serves the purpose of MBSE well:

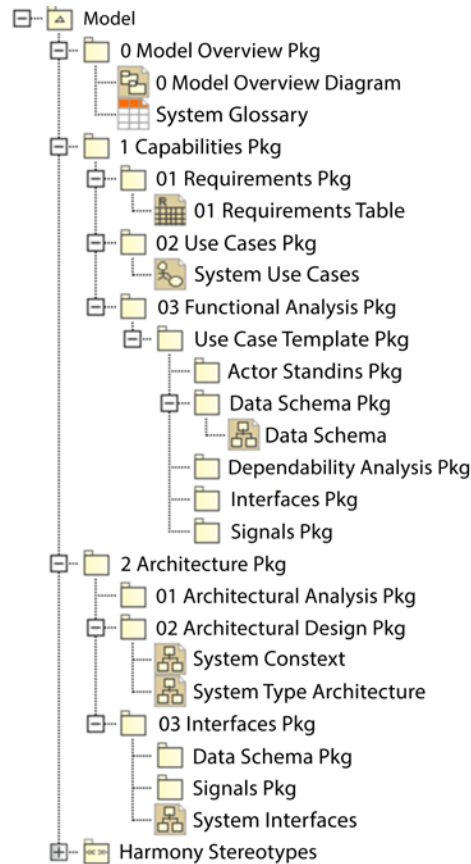


Figure 1.38: Systems model canonical structure

The main categories in *Figure 1.38* are packages that will hold the modeled elements and data.

The main packages are:

Model Overview Package – this package holds the model overview diagram, model summary information, and a glossary, if present.

Capabilities Package – holds all capability-related information, including requirements, use cases, and functional analyses.

Requirements Package – holds the requirements, either directly (most common) or as remote resources in the Jazz environment.

Use Case Package – holds the system use cases and use case diagrams.

Note that while this organization is labeled **canonical**, it is common to have minor variants on the structure.

Functional Analysis Package – holds the use case analyses, one (nested) package per use case analyzed. An example “template” package is shown with a typical structure for the analysis of a single use case.

Architecture Package – holds all design-related information for the Systems Model.

Architectural Analysis Package – hold architectural analyses, such as trade studies, usually in one nested package per analysis.

Architectural Design Package – holds the architectural design, the system and subsystem blocks, and their relations. Later in the process, it will hold the subsystem specifications, one (nested) package per subsystem.

Interfaces Package – holds the logical system and subsystem interfaces as well as the logical data schema for data and flows passed via those interfaces. Logical interfaces are discussed in more detail in *Chapter 2, System Specification: Functional, Safety, and Security Analysis*.

Add systems requirements, if they exist

It is not uncommon that you’re handed an initial set of system requirements. If so, they can be imported or referenced. If they are added later, this is where they go.

Modify the model structure for special needs, if required

You may identify special tasks or model information that you need to account for, so it’s OK to add additional packages as needed.

Add a model overview diagram

I like every model to contain a **Model Overview Diagram**. This diagram is placed in the **Model Overview** package and serves as a brief introduction to the model’s purpose, content, and organization. It commonly has hyperlinks to tables and diagrams of particular interest. The lower left-hand corner of *Figure 1.39* has a comment with hyperlinks to important diagrams and tables located throughout the model.

This aids model understanding and navigation, especially in large and complex models:

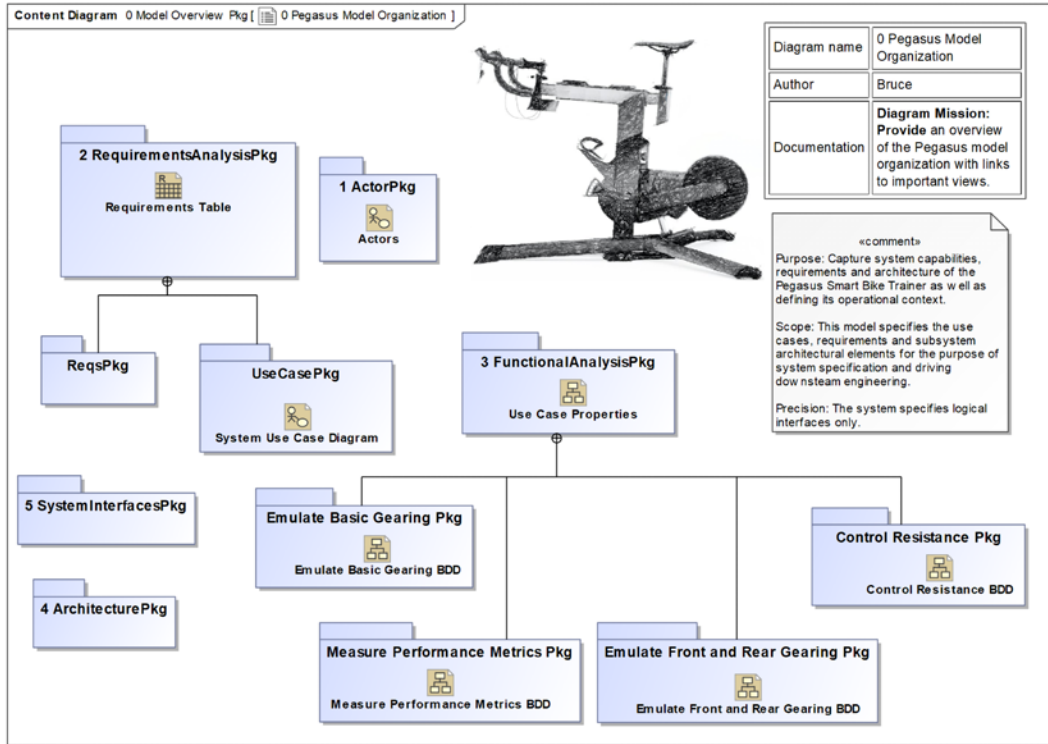


Figure 1.39: Example model overview diagram

After systems engineering work completes and the time comes to handoff to downstream engineering, more models must be created. In this case, the input is the Systems Model and the output is the Shared Model and a set of Subsystem Models. The Shared Model contains information common to more than one subsystem – specifically the physical system and subsystem interfaces and the corresponding physical data schema used by those interfaces. The details of elaborating those models are dealt with in some detail in *Chapter 4, Handoff to Downstream Engineering*, but their initial construction is shown in *Figure 1.40*:

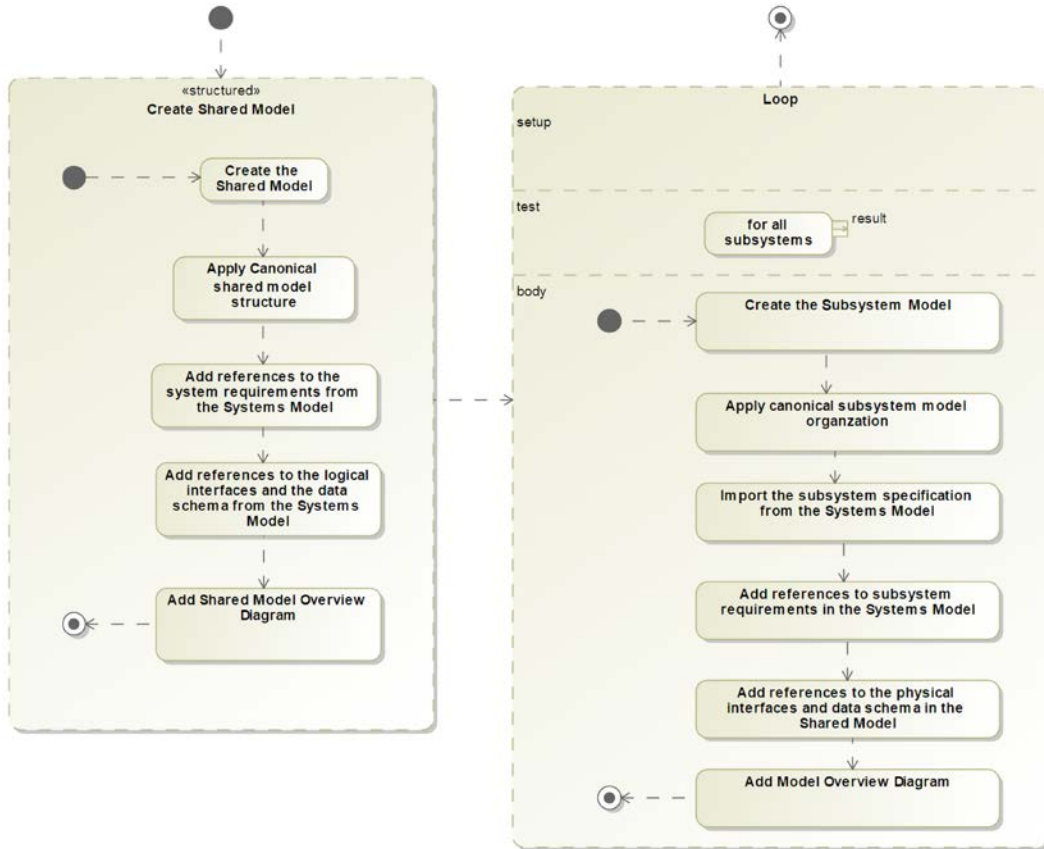


Figure 1.40: Organizing the shared and subsystem models

Create the Shared Model

This task creates an empty shared model. This model will hold the information shared by more than one subsystem; that is, each subsystem model will have a reference to these common interfaces and data definitions and elements.

Apply the canonical shared model structure

Apply the canonical shared model structure

The purpose of the shared model is twofold. First, using the logical interfaces and data schema as a starting point, derive the physical interfaces and data schema, a topic of *Chapter 4, Handoff to Downstream Engineering*. Secondly, the Shared Model serves as a common repository for information shared by multiple subsystems.

The organization shown in *Figure 1.41* does that:

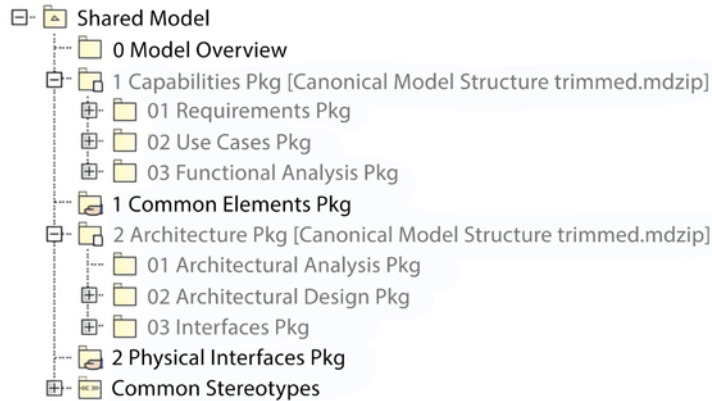


Figure 1.41: Shared model canonical structure

The **Requirements** and **Interfaces** packages in *Figure 1.41* reference the Systems Model packages of the same name. In this way, those elements are visible to support the work in the Shared Model. The **Physical Interfaces** package holds the physical interfaces and data schema to be used by the subsystems. The **Common Elements** package holds elements used in other places in the model or in multiple subsystems. The **Common Stereotypes** profile holds stereotypes either used in the **Physical Interfaces** package or created for multiple subsystems to use.

Add a reference to the system requirements

As shown above, the **Requirements** package of the systems model is referenced so that they are available to view but also so that the physical interfaces and data schema can trace to them, to provide a full traceability record.

Add references to the logical interfaces and data schema

The logical interfaces and related logical data schema specify the logical properties of those elements without specifying physical implementation details. The *Creating the Logical data schema* recipe from *Chapter 2, System Specification: Functional, Safety, and Security Analysis*, goes into the creation of those elements. By referencing these elements in the Shared Model, the engineer has visibility to them but can also create trace links from the physical interfaces and data schema to them.

Add model overview diagram

As in the systems model organization, every model should have a **Model Overview** diagram to serve as a table of contents and introduction to the model:

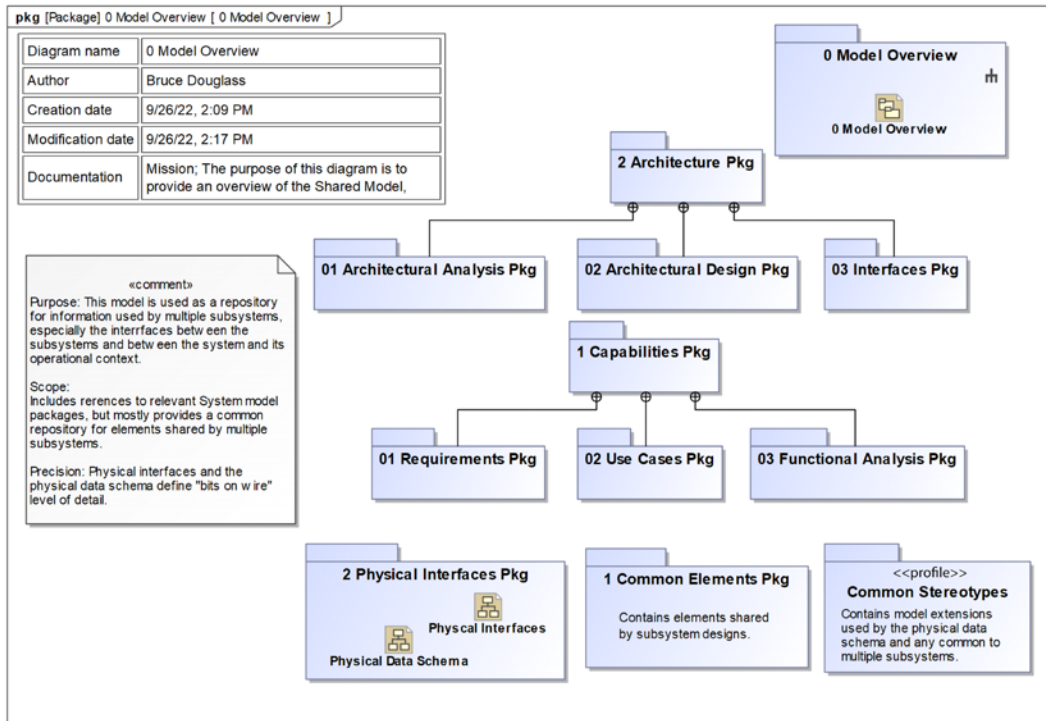


Figure 1.42: Example shared model overview diagram

Create the subsystem model

Commonly, each subsystem has its own interdisciplinary team, so the creation of a Subsystem Model per subsystem provides each team with a modeling workspace for their efforts.

Apply the canonical subsystem model organization

Figure 1.43 shows the canonical organization of a Subsystem Model. Remember, that each subsystem team has their own, with the same basic organization:

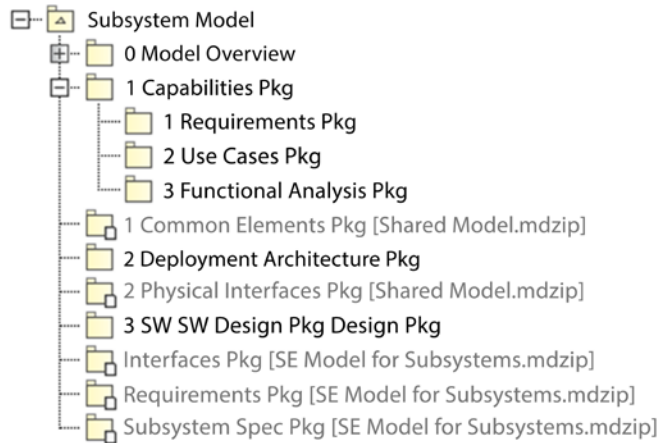


Figure 1.43: Subsystem model canonical structure

The **Common Stereotypes** and **Physical Interfaces** packages are referenced from the Shared Model, while the **Requirements** package is referenced from the Systems Model.

We would like to think that the requirements and use cases being handed down to the subsystem team are perfect; however, we would be wrong. First, there may be additional elaboration work necessary at the subsystem level to understand those subsystem requirements. Further, discipline-specific requirements must be derived from the subsystem requirement so that the electronics, mechanical, and software engineers clearly understand what they need to do. That work is held in the **Subsystem Spec Package**. If we create additional, more detailed use cases, they will be analyzed in the **Functional Analysis Package** in the same way that the system use cases are analyzed in the Systems Model.

The **Deployment Architecture Package** is where the identification of the facets and the allocation of responsibilities to the engineering disciplines takes place. To be clear, this package does not detail the internal structure of the facets; the electronics architecture, for example, is not depicted in this package, but the electronics facet as a black box entity is. Further, the interfaces between the electronics, software, and mechanical facets are detailed here as well.

Lastly, the **SW Design Package** is where the design and implementation of the software will be done. It is expected that the software will continue to work in the model but that the other facets will not. Alternatively, the software engineers can create their own separate model that references the subsystem model. For electronics and mechanical design, we expect that they will use their own separate tools and this model will serve only as a specification of what needs to be done in those facets. It is possible that the electronics design could be done here, using SysML or UML and then generating SystemC, for example, but that is fairly rare. SysML and UML are poorly suited to capture mechanical designs, as they don't have any underlying metamodel for representing or visualizing geometry.

Copy the subsystem specification

The Subsystem Package is *copied* (rather than referenced) from the Systems Model. This model holds the subsystem details such as subsystem functions and use cases. Of course, the name of this package in *Figure 1.43* is misleading; if the name of the subsystem was **Avionics Subsystem**, then the name of this package would be **Avionics Subsystem Package** or something similar.

In a practical sense, I prefer to copy the subsystem package from the Systems Model, because that isolates the Subsystem Model from subsequent changes to that package in the Systems Model that may take place in later iterations. The subsystem team may then explicitly re-import that changed Subsystem Model at times of its own choosing. If the subsystem package is added by reference, then whenever the systems team modifies it, the changes are reflected in the referencing Subsystem Model. This can also be handled by other means, such as referencing versions of the system model in the configuration management set, but I find this conceptually easier. However, if you prefer to have a reference rather than a copy, that's an acceptable variation point in the recipe.

Add a reference to the subsystem requirements

The **Requirements** package in the Systems Model also holds the derived subsystem requirements (see *Chapter 2, System Specification: Functional, Safety, and Security Analysis*). Thus, referencing the **Requirements** package from the system model allows easy access to those requirements.

Add a reference to the physical interfaces and data schema

Logical interfaces serve the needs of systems engineering well, for the most part. However, since the subsystem team is developing the physical subsystem, they need to know the actual bit-level details of how to communicate with the actors and other subsystems, so they must reference the physical interfaces from the Shared Model.

Add model overview diagram

As before, each model should have a **Model Overview** diagram to serve as a table of contents and introduction to the model.

How it works

At the highest level, there is a federated set of models defined here. The Systems Model holds the engineering data for the entire set, and this information is almost exclusively at the logical level of abstraction. This means that the important logical properties of the data are represented – such as the extent and precision of the data – but their physical properties are not. Thus, we might represent an element such as a *Radar Track* in the Systems Model as having a value property of range with an extent of 10 meters to 300 kilometers and a precision of ± 2 meters. Those are logical properties. But the physical schema might represent the value as a scaled integer with 100^* the value for transmission over the 1553 avionics bus. Thus, a range of 123 kilometers would be transmitted as an integer value of 12,300. This representation is a part of the physical data schema that realizes the logical properties.

Beyond the Systems Model, the Shared Model provides a common repository for information shared by multiple subsystems. During the systems engineering work, this is limited to the physical interfaces and associated physical data schema. Later in downstream engineering, other shared subsystem design elements might be shared, but that is beyond our scope of concern.

Lastly, each subsystem has its own Subsystem Model. This is a model used by the interdisciplinary subsystem team. For all team members, this model serves as a specification of the system and the related engineering facets. Remember that a facet is defined to be the contribution to a design specific to a single engineering discipline, such as software, electronics, or mechanical design. Software work is expected to continue in the model but the other disciplines will likely design their own specific tools.

Example

Figure 1.44 shows the initial organization of the *Pegasus System Model*, with Architecture 0 already added (see recipe *Architecture 0*). I've filled in a few other details, such as adding references to diagrams to illustrate how they might be shown here:

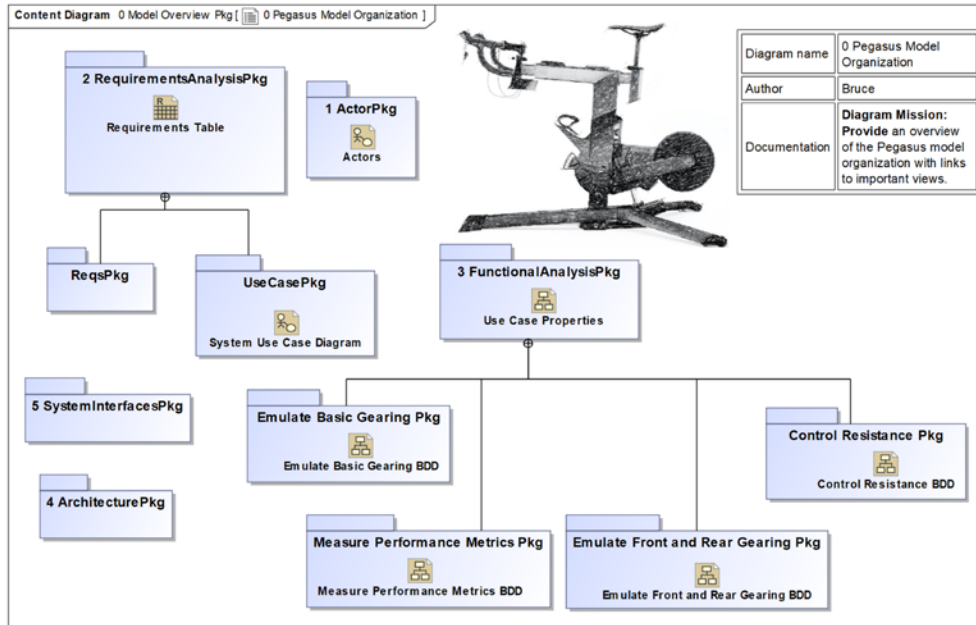


Figure 1.44: Pegasus system model – overview diagram

The shared model and subsystem model are *not* created early in the MBSE process but rather late, after the Pegasus architecture work is completed for the current iteration and the work developing the hand-off to downstream engineering is underway. We will discuss their creation in *Chapter 4, Handoff to Downstream Engineering*.

Managing change

In real engineering environments, work products are not the result of a single person working in isolation. Rather, they represent the integration of the efforts of many people. In model-based systems engineering, the model is central to the project and this means that many people must work simultaneously on the model. Without getting in each other's way. Or losing work.

The heart of managing such a shared engineering work product is **change management** and its close cousin **configuration management**. The former is the process and procedures that govern how changes made to a work product are controlled, including the controlled identification of changes, the implementation of those changes, and the verification of the changes. Configuration management is a systems engineering process for establishing and maintaining consistency of a product's performance, functional, and physical attributes with its requirements, design, and operational information throughout its life.

See https://en.wikipedia.org/wiki/Configuration_management for more information. Both of these are deep and broad topics and will be treated somewhat superficially here. It is nevertheless an important topic, so we will present a simple workflow for change and configuration management of models in this recipe.

Central to this recipe is the notion that we change a model for a purpose. In traditional engineering processes, this is managed via a **change request**, sometimes called an **Engineering Change Order (ECO)**. In agile processes, this is due to a **work item** in the iteration backlog. In either case, the change is made to the model for a reason and to achieve some goal. The problem is that many people need to change other elements in the model to achieve other goals. Frequently, different changes must be made by different people to the same model elements.

The concept of a **package** was introduced early in the development of UML. A package is a model element that contains other model elements, including use cases, classes, signals, behaviors, diagrams, and so on. As such, it is the fundamental unit of organization for models. It was also intended to be the fundamental **Configuration Item (CI)**, the atomic piece for configuration management. It was envisioned that a modeler would manage the configuration of a package as a unit, including all its internal contained elements together. Modern tools like Cameo and Rhapsody support this but also allow finer-grained configuration management down to the individual element level, if desired. Nevertheless, the package remains the most common CI.

The terms **version** and **revision** are common terms in the industry around this topic. A model *version* is a changed model. Such changes are typically considered minor. A revision is a controlled version and is generally considered a major change. It is common that revisions are numbered as whole numbers and versions are fractional numbers. Version 19.2 indicates revision 19 version 2. Revisions are considered baselines and permanent, and versions are minor changes and temporary. Revisions generally go through a more robust verification and validation process, often at the end of a sprint. Versions may be created at each test-driven development cycle (known in the Harmony process as the **nanocycle** – see the *Test-Driven Modeling* pattern in *Chapter 5, Demonstration of Meeting Needs*) and so come and go rapidly.

All this is relevant to the two basic workflows for change and configuration management: Lock and Release and Branch and Merge. Both primarily work at the package level.

Purpose

The purpose of this recipe is to provide a workflow for robust change control over model contents in the presence of multiple and possibly simultaneous changes to the model made by different engineers.

Inputs and preconditions

This recipe starts after a model is **baselined**. A baseline means that the model is stored under a configuration and change control environment, usually after it has achieved some basic level of maturity. This means that there is a current source of truth for the model contents, and all changes are managed by the configuration and change control environment. The second precondition is the presence of a change to be made to the model. This recipe does not concern itself with how the decision is made to proceed with the change, only that the decision has been made.

Outputs and postconditions

The output is the updated model put back under configuration and change control either as a version or as a revision, depending on the scope of the change.

How to do it

Figure 1.45 shows the workflow for this recipe. Two alternative flows are shown. On the left is the more common and simpler Lock and Release sub-workflow, and on the right is the Branch and Merge sub-workflow. Both sub-workflows reference the **Make Change** activity, which is shown as a nested diagram on the right of the figure:

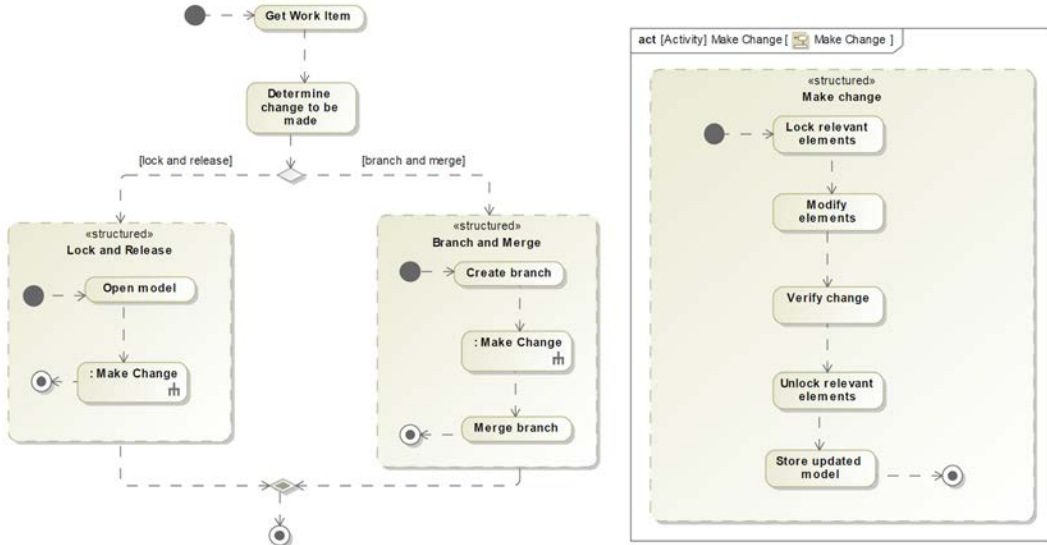


Figure 1.45: Managing change

Get work item

The start of the workflow is to get a request to modify the baseline model. This can be the addition of a model structure, behavior, functionality, or some other property. It can also be to repair an identified defect or pay off some technical debt.

Determine the change to be made

Once the request for change has been received, the engineer generally develops a plan for the modification to be made. The work may involve a single element, an isolated set of elements in the same package, or broad, sweep changes across the model.

Lock and release

The Lock and Release workflow is by far the most common approach to modifying a model under configuration control. The detailed sub-tasks follow. Its use means that other engineers are prohibited from making changes to the elements checked out to the first engineer, until the locks owned by the latter are released.

Open the model

This step opens the model in the configuration management repository.

Make a change

This makes and verifies the change in the model. It has a number of sub-tasks, which will be described shortly.

Branch and merge

This alternative sub-workflow is useful when there are many people working on the model simultaneously. People will create a separate branch in which to work so that they will not interfere with anyone else's work. The downside is that it is possible that they will make incompatible changes, making the merge back into the main trunk more difficult. This sub-workflow has three steps.

Create a branch

In this step, the engineer creates a branch of the main project; this latter model is known as the trunk. A branch is a separate copy of the project completely independent of the original, but containing, at least at the start, exactly the same elements.

Merge a branch

This step takes the changes model and merges those changes back into the main project or trunk. There are usually changes that are merely additions but other changes may be in conflict with the current version of the trunk, so this step may require thoughtful intervention.

Make a change activity

This activity is referenced in both the Lock and Release and Branch and Merge sub-workflows. It contains several included steps.

Lock relevant elements

Whenever you load a model that is under configuration control, you must explicitly lock the elements you wish to modify.

Modify elements

This step is where the actual changes to the model are performed.

Verify change

Before checking in a changed model, it is *highly recommended* that you verify the correctness of the changes. This is often done by performing tests and a review/inspection of the model elements. Recipes in *Chapter 5, Demonstration of Meeting Needs*, discuss how to perform such verification.

Unlock relevant elements

Once the changes are ready to be put back into the configuration management repository, they must be unlocked so that other engineers can access them.

Store updated model

In this final step of the *Make Change Activity*, store the unlocked model back into the configuration management repository.

Example

Since we are using Cameo in this book, we will be using Cameo Teamwork Cloud as the configuration management environment. Other tools, such as Rhapsody from IBM, can use various configuration management tools but with Cameo, either Teamwork Cloud or the older Teamwork Server is required.

Because the more common Lock and Release workflow is a degenerative case of the more elaborate Branch and Merge sub-workflow, we will just give an example of the latter.

The example here shows a context for an exercise bike. The context block is named **Pain Cave**, and it contains a **Rider** part that represents the system user, and an **Exercise Bike** part that represents the system under design. The **Rider** block provides **cadence** and **power** as inputs to the system as flow properties and receives **resistance** back:

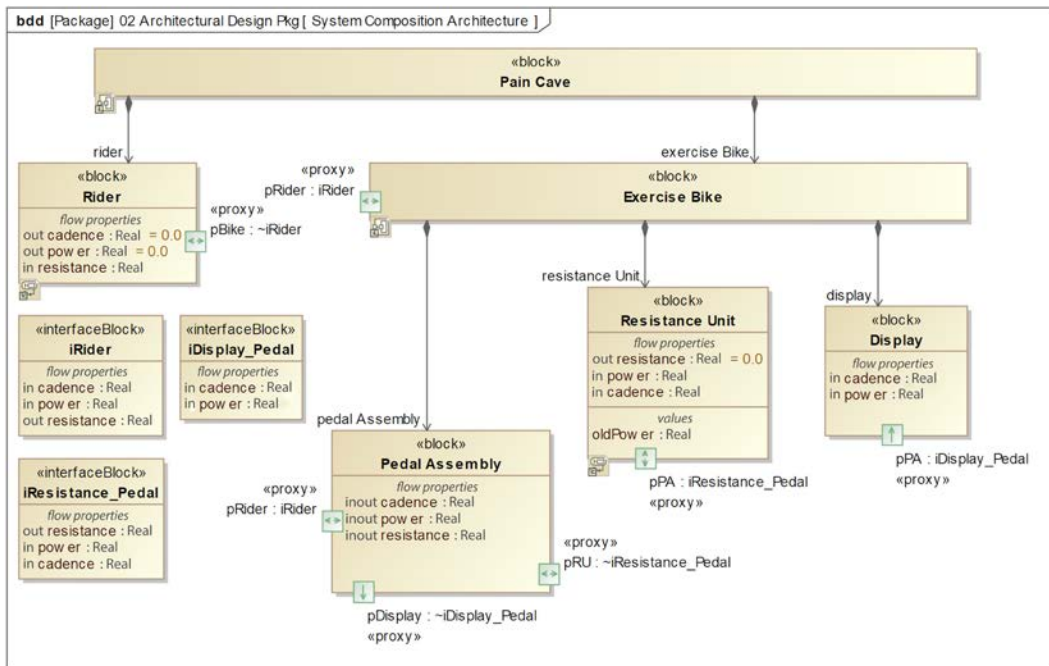


Figure 1.46: Pain cave model composition architecture

Figure 1.47 shows the internal block diagram connecting the **Rider** and **Exercise Bike** parts in the context of the **Pain Cave**, while Figure 1.48 shows the internal structure of the **Exercise Bike** itself:

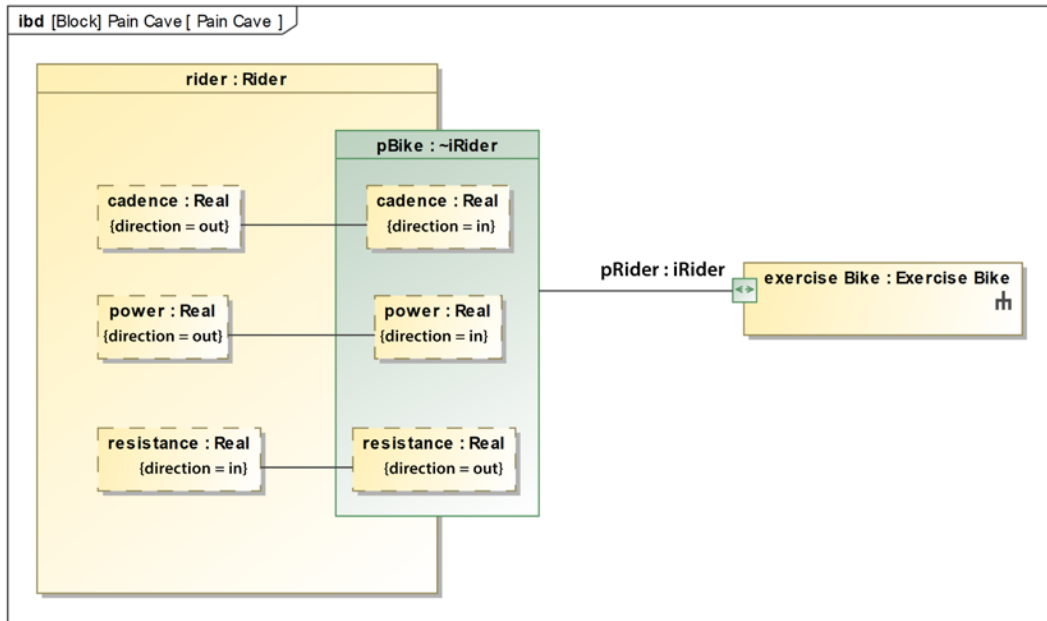


Figure 1.47: Pain cave connected context

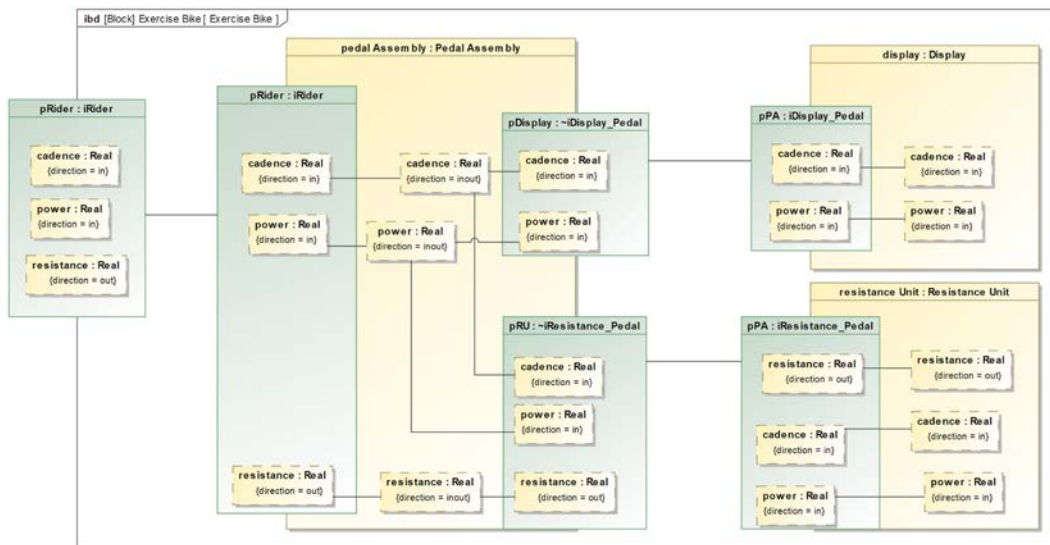


Figure 1.48: Exercise bike connected architecture

Finally, *Figure 1-49* shows the behaviors of two elements, **Rider** and **Resistance Unit**. The concept here is that the **Resistance Unit** produces resistance based on the power and cadence it receives from the **Rider**. Of course, this behavior performed is not in any way realistic but just a simple example of behavior:

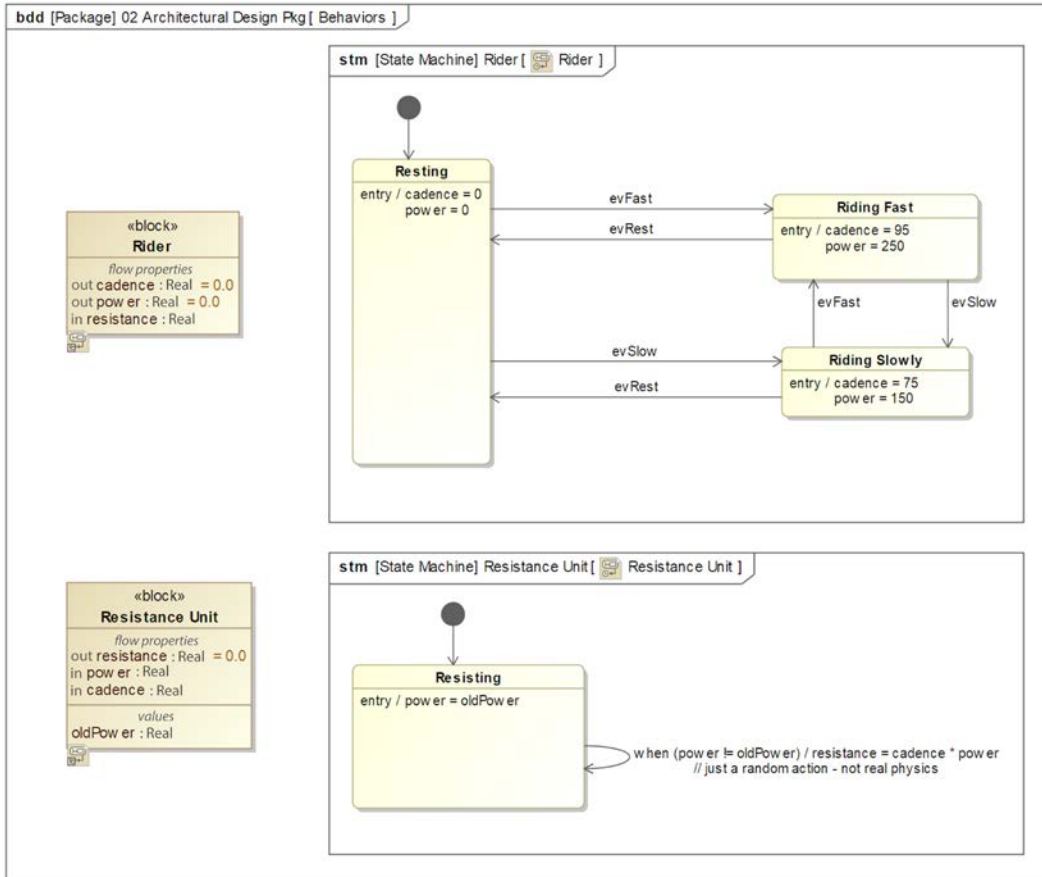


Figure 1.49: Behaviors

This system does execute, as can be seen in *Figure 1.50*, which shows the current states of the running simulation using Cameo's Simulation Toolkit, and the current values of the flow properties in *Figure 1.51*. In Cameo, you can visualize a running system in a number of ways, including the creation of diagrams that contain other diagrams; for activity and state diagrams, Cameo highlights the current state or action as it executes the model. The Simulation Toolkit also exposes the current instances and values held in value properties during the simulation:

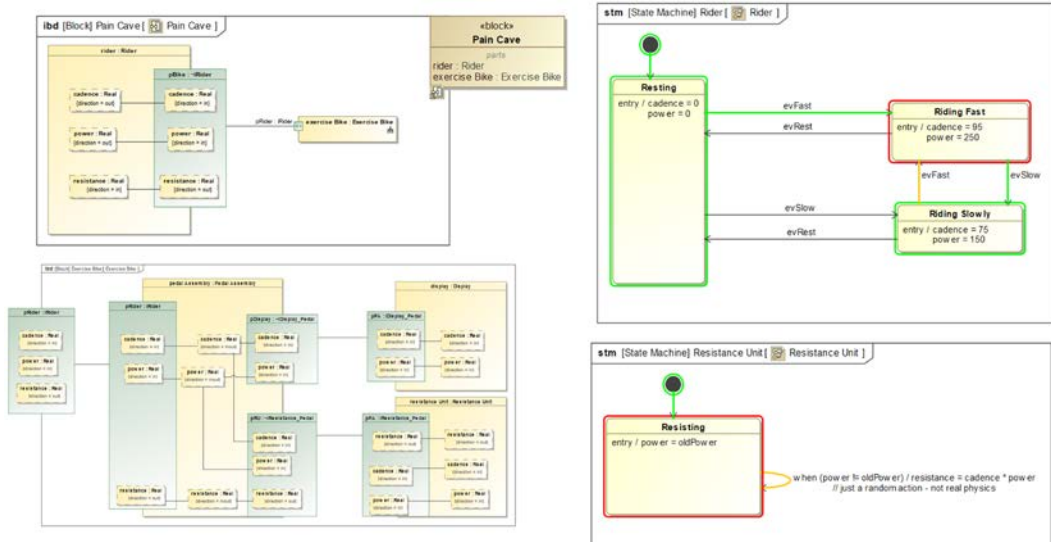


Figure 1.50: Pain cave simulation view

Name	Value
Pain Cave	Pain Cave@7324d61a
exercise Bike : Exercise Bike	Exercise Bike@33372062
display : Display	Display@55cde876
cadence : Real	95.0000
power : Real	250.0000
pedal Assembly : Pedal Assembly	Pedal Assembly@704749c1
cadence : Real	95.0000
power : Real	250.0000
resistance : Real	18750.0000
resistance Unit : Resistance Unit [Resisting]	Resistance Unit@355a566b
cadence : Real	95.0000
oldPower : Real	
power : Real	
resistance : Real	18750.0000
rider : Rider [Riding Fast]	Rider@211ea37
cadence : Real	95.0000
power : Real	250.0000
resistance : Real	18750.0000

Figure 1.51: Pain cave values during simulation

This is our baseline model.

Get work item

For this example, we will work on two user story work items together:

1. As a rider, I want to be able to see the pedal cadence and power as I ride so that I can set my workout levels appropriately.
2. As a Rider, I want to be able to control the amount of resistance, given cadence, and power so I have finer control over my workout efforts.

Determine the change to be made

In this simple example, the changes are pretty clear. The **Display** block must be able to get input from the Rider to change the scaling factor for the resistance, and then send this information to the **Resistance Unit**. The **Resistance Unit** must incorporate the scaling factor in its output. Additionally, behavior must be added to the **Display** block to display cadence and power.

Create branch

In the Cameo tool, after logging in to the Teamwork Cloud environment, select your project using **Collaborate > Projects** menu to select your project. At the right of the project from which you want to create a baseline, click on the ellipsis to open the **Select Branch** dialog, then select **Edit Branches**, which opens the **Edit Branches** dialog. Here, select the version from which you wish to branch and select **Create Branch**. These dialogs are all shown in *Figure 1.52*:

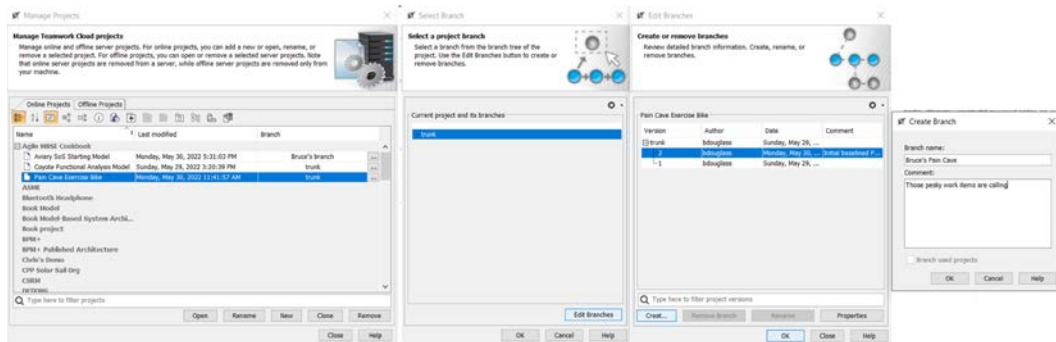


Figure 1.52 Creating a branch

Once created, you can open the branch in the **Manage Projects** dialog.

Make Change::Lock relevant elements

To edit the elements you must lock them. In Cameo, this is a right-click menu option for elements in the containment tree. We will work exclusively in the **O2 Architectural Design Pkg** package, so we will lock that package and all its contents. In Cameo, you must select **Lock Elements for Edit Recursively** to lock the package and the elements it contains. If you lock without recursion, you only have edit rights to the package itself and not its content.

Make Change::Modify elements

Let's look at the changes made for each of the work items:

1. As a rider, I want to be able to see the pedal cadence and power as I ride so that I can set my workout levels appropriately.

For this change, we'll modify the **Display** block to have a state behavior driven by a change event, whose specification is when either the power or the cadence changes. This requires the addition of a couple of value properties. The **Display** state machine is shown in *Figure 1.53*. For our purposes here, we'll just use Groovy print statements to print out the values. In the simulation, these print statements output text to the console of the Simulation Toolkit:

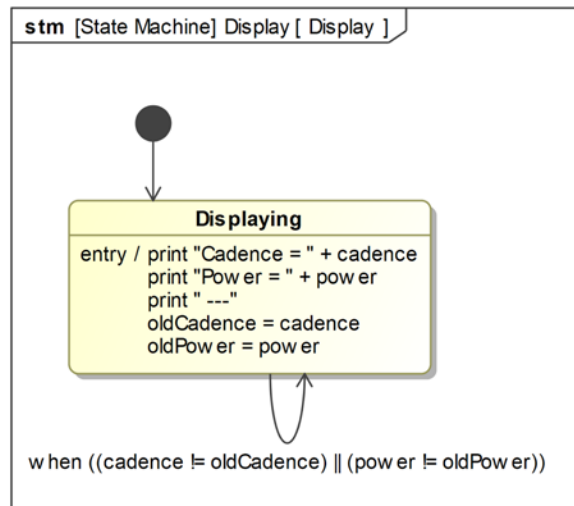


Figure 1.53: Display state machine

2. As a Rider, I want to be able to control the amount of resistance, given cadence, and power.

For this work item, we'll need to add a pair of ports between the **Rider** and the **Resistance Unit** blocks so the **Rider** can send the **evAugmentGear** and **evDecrementGear** events. The **Rider** state machine will be extended to be able to send these events (under the command of the simulation user), and the **Resistance Unit** state machine will accept these events and use them to update the **scaling factor** used to compute resistance. We will also need to add said **scaling factor** and update the computation.

Figure 1.54 shows the updated **Rider** state machine and Figure 1.55 shows the updated **Resistance Unit** state machine. The latter state machine was refactored a bit to account for the issue that you also want to recompute the resistance when the gearing is changed, as well as when cadence or power changes:

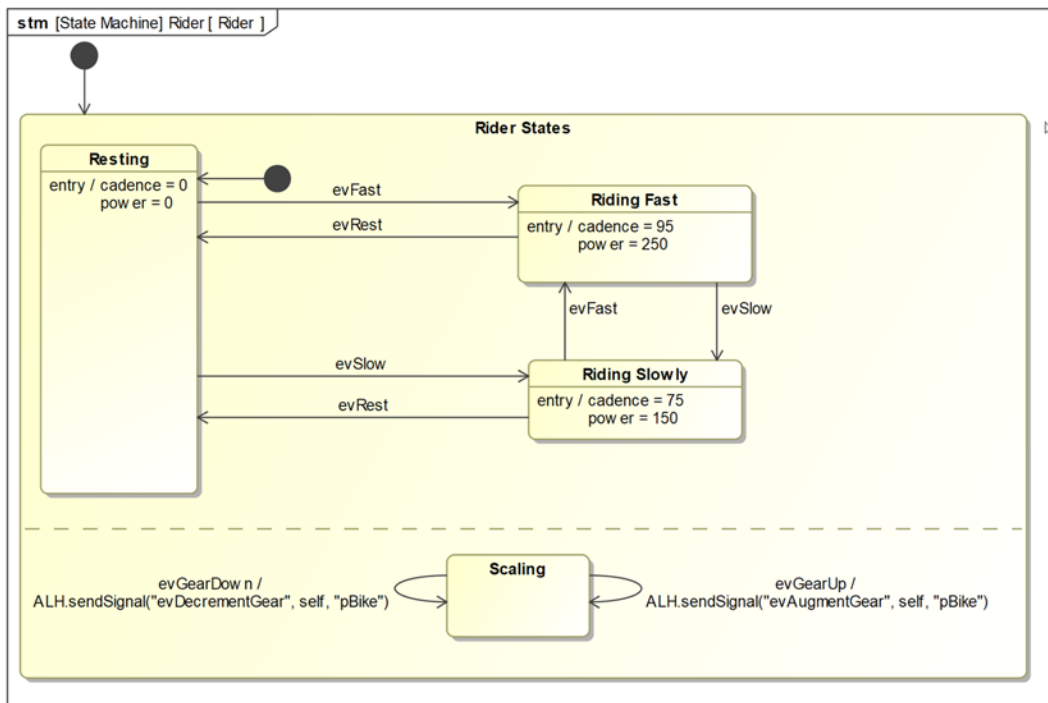


Figure 1.54: Updated rider state machine

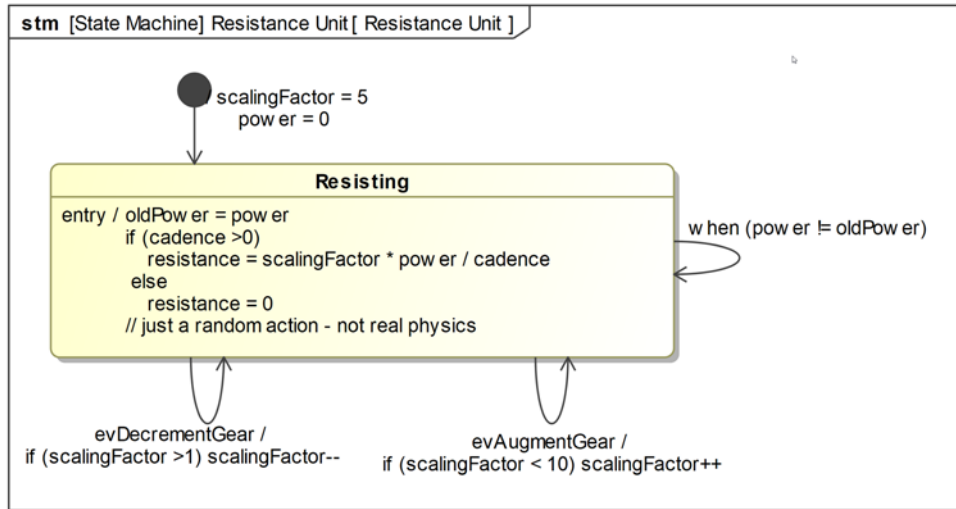


Figure 1.55: Updated resistance unit state machine

The updated block definition diagram is shown in Figure 1.56. Note the changes to the interface block `iRider`, the addition of a new port, and the value property for the `Resistance Unit`:

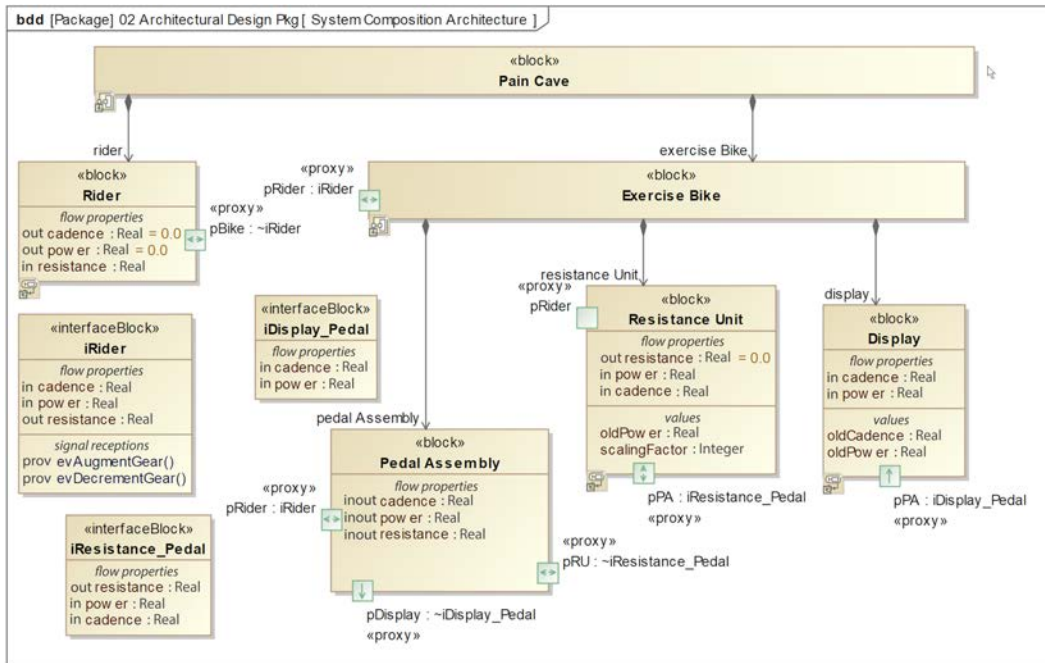


Figure 1.56: Updated pain cave BDD

Lastly, *Figure 1.57* shows the updated internal block diagram for the **Exercise Bike**. Note that the **Exercise Bike.pRider** port is now also connected to the **pRider** port of the **Resistance Unit**. This allows the latter to receive the gearing events:

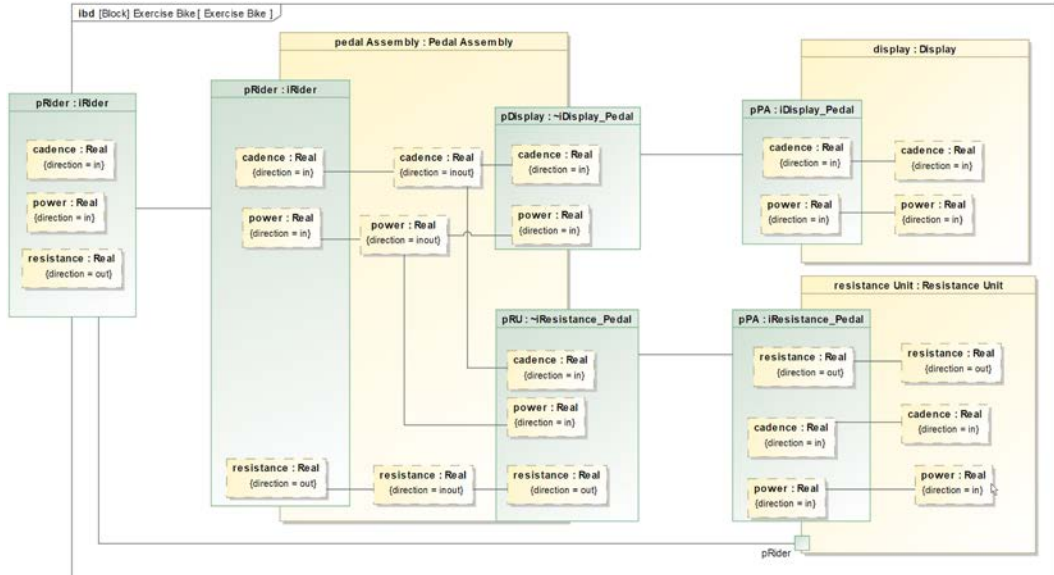


Figure 1.57: Updated exercise bike IBD

Make Change::Verify change

To make sure we made the changes correctly, we can run the simulation. As a simulation operator, you can use the simulation toolkit to start the operation sending the **evFast** signal to the Rider, and then send the **evGearUp** and **evGearDown** events to see the effect on the resulting resistance. Also, you can check the output console to ensure the cadence and power are being displayed.

Cameo makes it simple to create a simulation view that allows visualization of the behaviors (see *Figure 1.58*). I've superimposed the console window in the figure so you can see the results of the **Display** block behavior:

In Cameo, open the trunk (target) model, then select **Collaborate > Merge From**; this opens the **Select Server Project** dialog. Here, select the branch version you just created:

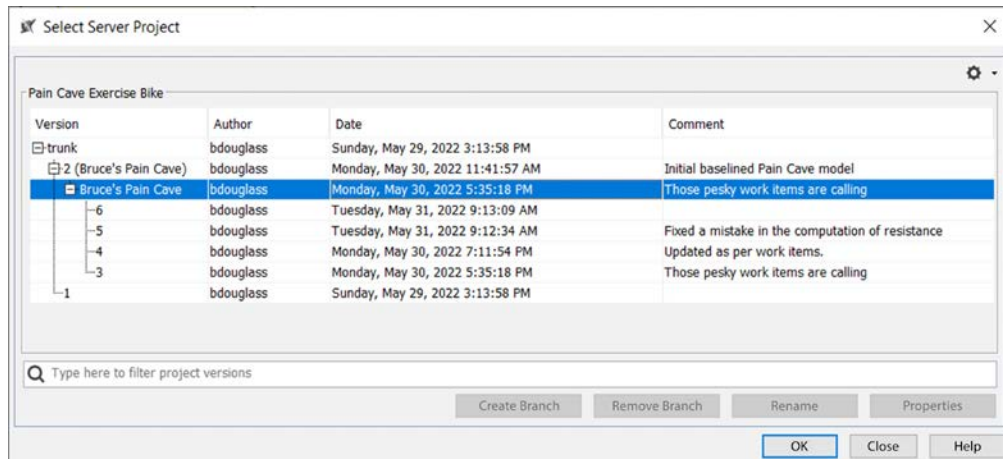


Figure 1.59: Selecting the branch for merge

Then a dialog pops up if you want to merge with the trunk locked or not. Always merge with the trunk locked.

Once you click on **Continue**, the **Merge** dialog opens. There are many options for seeing what the changes are, accepting some changes, and rejecting others. Color coding identifies the different kinds of changes: addition, deletions, and modifications:

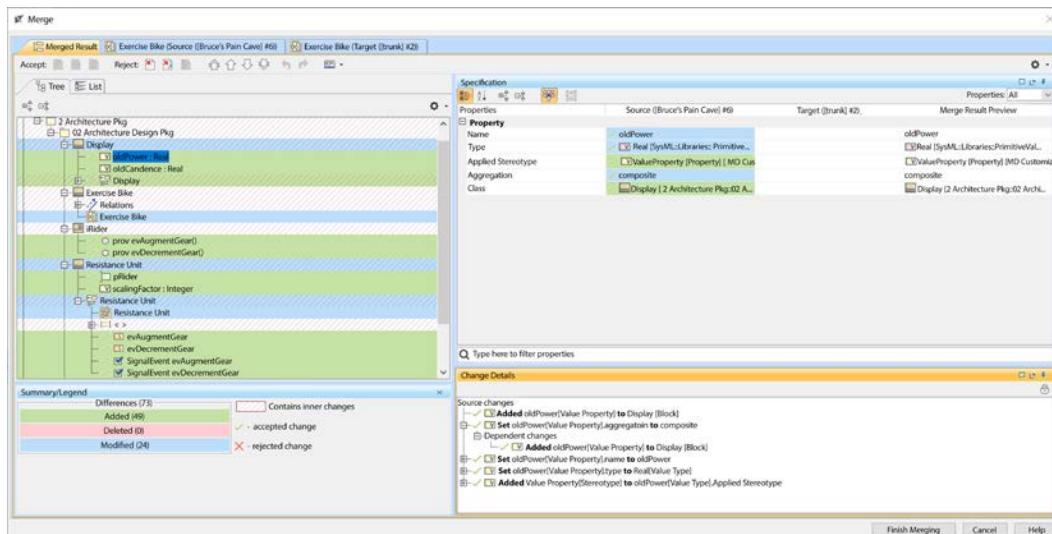


Figure 1.60: Review merge changes

Any conflicting changes are highlighted and can be resolved manually, by right-clicking and accepting either the source (updated branch) change or keeping the target (trunk) version of the element. Cameo has a nice feature that allows you to explore diagrammatic changes graphically. The scroll bar at the bottom of the window allows you to switch between the unchanged and changed views of the diagram.

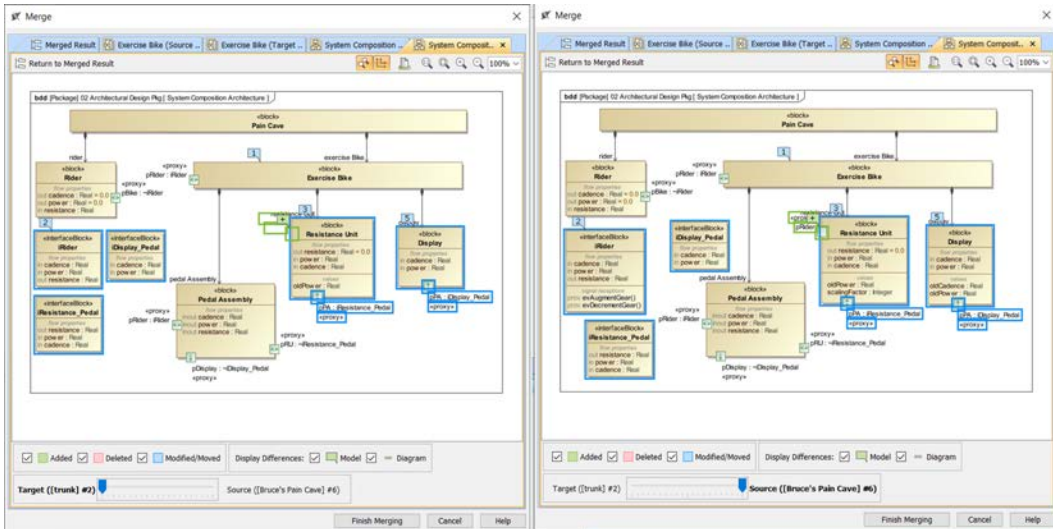


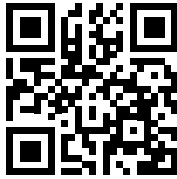
Figure 1.61: Reviewing diagrammatic changes

Click **Finish Merging** when you're ready to complete the merge process and click **Collaborate > Commit Changes** to save the updated merged model to the trunk. Remember that the Branch and Merge approach is the lesser-used workflow, and the simpler Lock and Release approach is more common. Just remember to lock and unlock elements recursively and be sure to unlock when you're done making your changes.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>



2

System Specification

Recipes in this chapter

- Functional Analysis with Scenarios
- Functional Analysis with Activities
- Functional Analysis with State Machines
- Functional Analysis with User Stories
- Model-Based Safety Analysis
- Model-Based Threat Analysis
- Specifying Logical System Interfaces
- Creating the Logical Data Schema

This chapter contains recipes to do with the capturing and analysis of requirements. The first four recipes are alternative ways to achieve essentially the same thing. Functional analysis generates high-quality requirements, use cases, and user stories – all means to understand what the system must be.

By “high-quality requirements,” I mean requirements focused on a use case that are demonstrably:

- Complete
- Accurate
- Correct
- Consistent
- Verifiable

The problem is that natural language is ambiguous, imprecise, and only weakly verifiable. Keeping the human-readable text is very useful, especially for non-technical stakeholders, but insufficient to ensure we are building the right system. Let's look at why that is.

Why aren't textual requirements enough?

There are many reasons why textual requirements by themselves fail to result in usable, high-quality systems.

First, it is difficult to ensure all the functionality is present:

- All normal (sunny day) functionality?
- All edge-cases?
- All variations of inputs, sequences, and timings?
- All exception, error, and fault cases?
- Qualities of service such as performance, range, precision, timing, safety, security, and reliability?
- All stakeholders appropriately represented?

Getting that much detail is a daunting task indeed. But even beyond that, there is an “air gap” between realizing a possibly huge set of “shall” statements and actually meeting the stakeholder needs. The stakeholder believes that if the system performs a specific function, then in practice, their needs will be met. Experience has shown that this is not always true. Customers often ask for features that don't address their true needs. Further, requirements are volatile and interact often in subtle but potentially catastrophic ways.

We address this issue by capturing requirements both in textual and formal means via modeling. The textual requirements are important because they are human-readable by anyone even without modeling training and often appear in binding legal contracts. The model representation of the requirement is more formal and lends itself to more rigorous thought and analysis, including simulation. In general, both forms are necessary.

Definitions

Before we get into the recipes, let's agree on common terms.

Requirement	<p>A stakeholder requirement is a statement of what a stakeholder needs. A system requirement is a statement of what the system must do to satisfy a stakeholder's need. We will focus on system requirements in this chapter. Normally, requirements are written in an active voice using the <i>shall</i> keyword to indicate a normative requirement, as in</p> <p><i>The system shall move the robot arm to comply with the user directive.</i></p>
Actor	<p>An actor is an element outside the scope of the system we are specifying that has interactions with the system that we care about. Actors may be human users, but they can also be other systems, software applications, or environments.</p>
Use case	<p>A use case is a collection of scenarios and/or user stories around a common usage of a system. One may alternatively think of a use case as a collection of requirements around a usage-centered capability of the system. Still another way to think about use cases is that they are a sequenced set of system functions that execute in a coherent set of system-actor interactions. These all come down to basically the same thing. In practice, a use case is a named usage of a system that traces to anywhere from 10-100 requirements and 3-25 scenarios or user stories.</p>
Activity	<p>An activity in SysML is a composite behavior of some portion of a system. Activities are defined in terms of sequences of <i>actions</i> that, in this context, correspond to either</p> <ul style="list-style-type: none"> • A system function • An input • An output <p>Activities can model the behavior of use cases. Activities are said to be <i>fully constructive</i> in the sense that they model all possible behavior of the use case.</p>
State machine	<p>A state machine in SysML is a classifier behavior of a system element, such as a block or use case. In this context, a state machine is a <i>fully constructive</i> behavior focusing on the conditions of the system (states) and how the system changes from state to state, executing system functions along the way.</p>
Scenario	<p>A scenario is an interaction of a set of elements in a particular case or flow. In this usage, a scenario represents a partially complete behavior showing the interaction of the actors with the system as it executes a use case. The reason that it is partially complete is that a given scenario only shows one or a very small number of possible flows within a use case. Scenarios are roughly equivalent to user stories. In SysML, scenarios are generally captured using sequence diagrams.</p>

User Story	<p>A user story is a statement about system usage from a user or actor’s point of view that achieves a user goal. User stories describe singular interactions and so are similar in scope to scenarios. User stories use a canonical textual formulation such as</p> <p>“As a” <user> “I want” <feature> “so that” <output or outcome>.</p> <p>For example,</p> <p><i>As a pilot, I want to control the rudder of the aircraft using foot pedals so that I can set the yaw of the aircraft.</i></p> <p>User stories tend to be most beneficial for simpler interactions, as complex interactions are difficult to write out in an understandable text. Scenarios are generally preferred for complex interactions or when there is a lot of precise detail that must be specified. Consider this somewhat unwieldy user story:</p> <p><i>As a navigation system, I want to measure the position of the aircraft in 3 dimensions with an accuracy of +/- 1 m every 0.5s so that I can fly to the destination.</i></p> <p>And that’s still a rather simple scenario. We will talk about user stories more later in this chapter in the <i>Functional Analysis with User Stories</i> recipe.</p>
------------	--

Table 2.1: Some important specification terms

Now that we have defined the most important terms used in system specification, let’s look at different recipes that allow us to understand what we want the system to do.

Functional Analysis with Scenarios

As stated in the chapter introduction, functional analysis is a means to both capture and improve requirements through analysis. In this case, we’ll begin with scenarios as a way to elicit the scenarios from the stakeholder and create the requirements from those identified interactions. We then develop an executable model of the requirements that allows us to verify that the requirements interact how we expect them to, identify missing requirements, and perform “what-if” analyses for additional interactions.

Purpose

The purpose of this recipe is to create a high-quality set of requirements by working with the stakeholders to identify and characterize interactions of the system with its actors. This is particularly effective when the main focus of the use case is the interaction between the actors and the system, or when trying to gather requirements from non-technical stakeholders.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view.

Outputs and postconditions

There are several outcomes, the most important of which is a set of requirements accurately and appropriately specifying the behavior of the system for the use case. Additional outputs include an executable use case model, logical system interfaces to support the use case behavior along with a supporting logical data schema, and a set of scenarios that can be used later as specifications of test cases.

How to do it

Figure 2.1 shows the workflow for this recipe. There are many steps in common with the next two recipes:

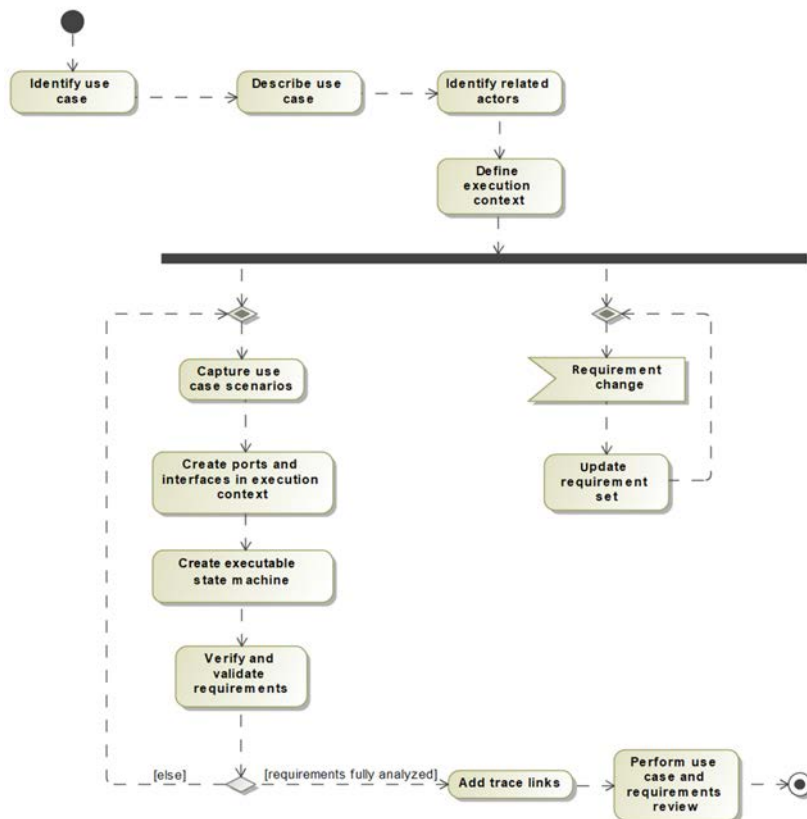


Figure 2.1: Functional analysis with scenarios

Identify use case

This first step is to identify the generic usage of which the scenarios of interest, user stories, and requirements are aspects.

Describe use case

The description of the use case should include its purpose, and a general description of the flows, preconditions, postconditions, and invariants (assumptions). Some modelers add the specific actors involved, user stories, and scenarios, but I prefer to use the model itself to contain those relations.

Identify related actors

The related actors are those people or systems outside our scope that interact with the system while it executes the current use case. These actors can send messages to the system, receive messages from the system, or both.

Define execution context

The execution context is a kind of modeling “sandbox” that contains an executable component consisting of elements representing the use case and related actors. The recommended way to achieve this is to create separate blocks representing the use case and the actors, connected via ports. Having an isolated simulation sandbox allows different system engineers to progress independently on different use case analyses.

Capture use case scenarios

Scenarios are singular interactions between the system and the actors during the execution of the use case. When working with non-technical stakeholders, it is an effective way to understand the desired interactions of the use case. We recommend starting with normal, “sunny day” scenarios before progressing to edge case and exceptional “rainy day” scenarios. It is important to understand that every message identifies or represents one or more requirements.

Create ports and interface in execution context

Once we have a set of scenarios, we’ve identified the flow from the use case to the actors and from the actors to the system. By inference, this identifies ports relating the actors and the system, and the specific flows within the interfaces that define them.

Create executable state machine

This step creates what I call the “normative state machine.” Executing this state machine can recreate each of the scenarios we drew in the *Capture use case scenarios* step. All states, transitions, and actions represent requirements. Any state elements added only to assist in the execution that do not represent requirements should be stereotyped as «non-normative» to clearly identify this fact. It is also common to create state behavior for the actors in a step known as “instrumenting the actor” to support the execution of the use case in the execution context.

Verify and validate requirements

Running the execution context for the use case allows us to demonstrate that our normative state machine in fact represents the flows identified by working with the stakeholder. It also allows us to identify flows and requirements that are missing, incomplete, or incorrect. These result in *Requirements_change* change requests to fix the identified requirements defects.

Requirements_change

Parallel to the development and execution of the use case model, we maintain the textual requirements. This workflow event indicates the need to fix an identified requirements defect.

Update requirement set

In response to an identified requirements defect, we fix the textual requirements by adding, deleting, or modifying requirements. This will then be reflected in the updated model.

Add trace links

Once the use case model and requirements stabilize, we add trace links using the «trace» relation or something similar. These relations allow the backtrace to stakeholder requirements as well as forward links to any architectural elements that might already exist. For relations forward to design elements, we generally prefer «satisfy».

Perform use case and requirements review

Once the work has stabilized, a review for correctness and compliance with standards may be done. This allows subject matter experts and stakeholders to review the requirements, use cases, states, and scenarios for correctness, and for quality assurance staff to ensure compliance with modeling and requirements standards.

Example

Identify use case

This example will examine the **Emulate Basic Gearing** use case. The use case is shown in *Figure 2.2*:

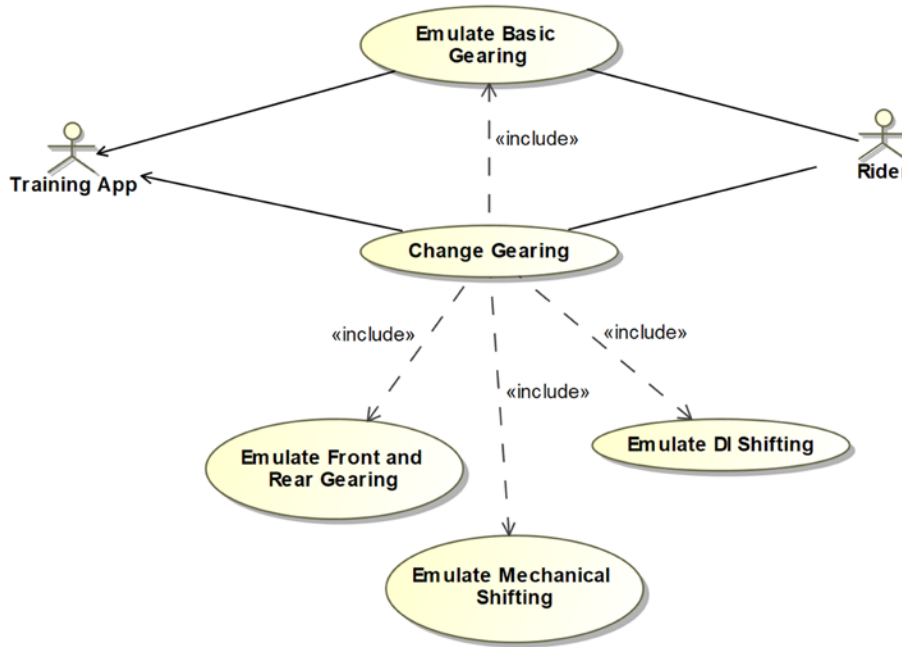


Figure 2.2: Emulate Basic Gearing Use Case

Describe use case

All model elements deserve a useful description. In the case of a use case, we typically use the format shown here:

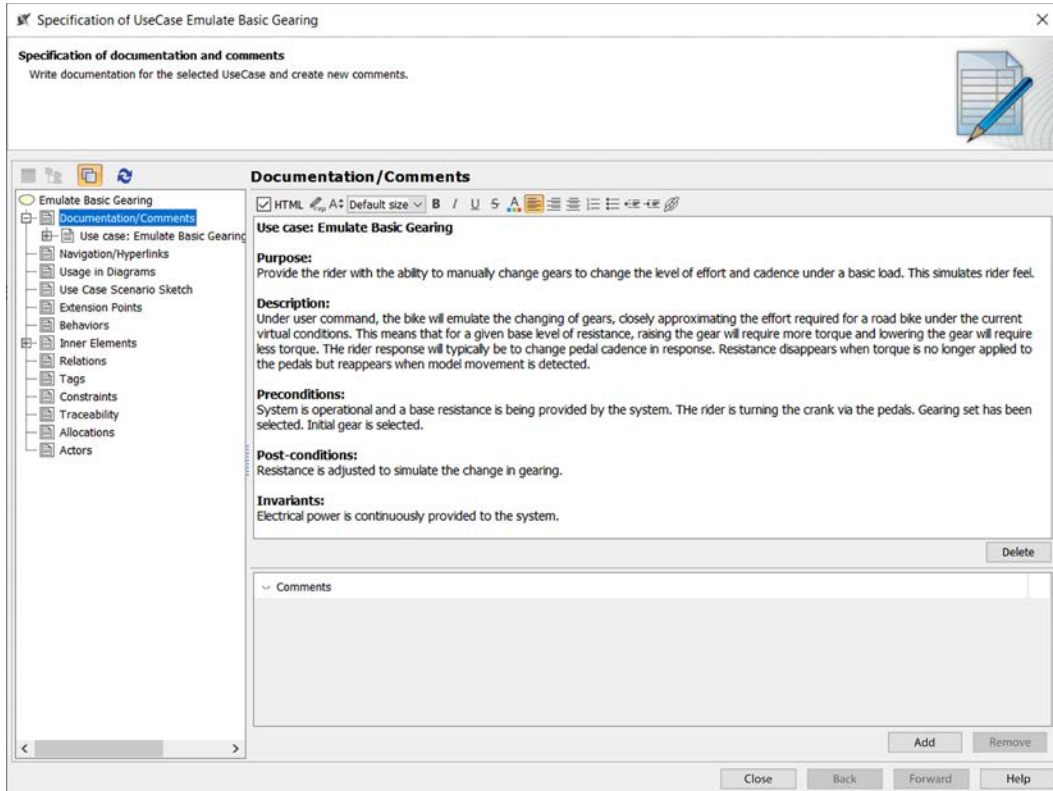


Figure 2.3: Use Case Description

Identify related actors

The related actors in this example are the **Rider** and the **Training App**. The rider signals the system to change the gearing via the gears control and receives a response in terms of changing resistance. The training app, when connected, is notified of the current gearing so that it can be displayed. The relation of the actors to the use case is shown in *Figure 2.2*.

Define execution context

The execution context creates blocks that represent the actors and the use case for the purpose of the analysis. In this example, the following naming conventions are observed:

- The block representing the use case has the use case name (with white space removed) preceded by **uc_**. Thus, for this example, the use case block is named **uc_EmulateBasic-Gearing**.

- Blocks representing the actors are given the actor name preceded with **a** and an abbreviation of the use case. For this use case, the prefix is **aEBG_** so the actor blocks are named **aEBG_Rider** and **aEBG_TrainingApp**.
- The interface blocks are named **i + <use case block>_<actor block>**. The names of the two interface blocks are **iEmulateBasicGearing_Rider** and **iEmulateBasicGearing_TrainingApp**. The normal form of the interface block is associated with the proxy port on the use case block; the conjugated form is associated with the corresponding proxy port on the actor block.

All these elements are shown on the block definition diagram in *Figure 2.4*. This diagram uses a Cameo feature to display the contents of a diagram – in this case, the IBD for the **Emulate Basic Gearing Context** block – on another:

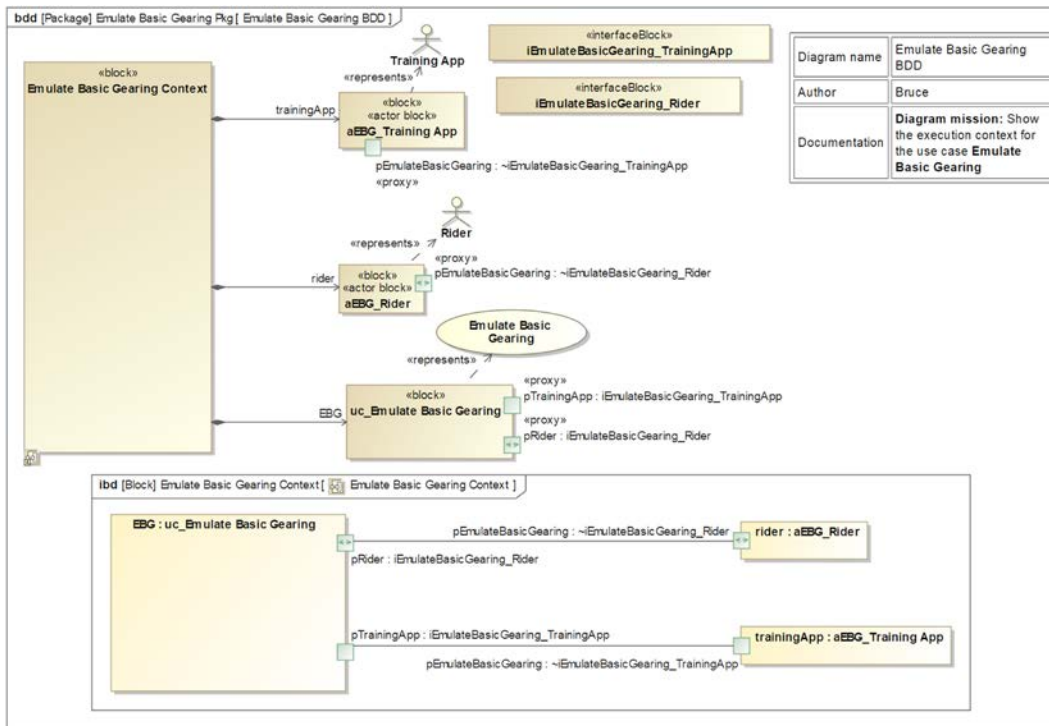


Figure 2.4: Emulate Basic Gearing Execution Context

Capture use case scenarios

Creating continuous flows in Cameo sequence diagrams isn't obvious. You'll need to do a few preparatory steps:

1. First, create the «cont inuous» stereotype. Add this into a profile in the project and use the default **Element** for the base metaclass.
2. Add flow properties to the blocks that you want to flow. In this first scenario, we want **resistance** to flow from the **aEBG_Rider** block to the **uc_Emulate Basic Gearing** block and **appliedTorque** to flow in the opposite direction.
3. Add value types **AppliedTorque** and **Resistance** (as subtypes of **Real**) and use these to type the above flow properties.
4. On the context internal block diagram, add item flows of the created types to the connector between the **rider** and **EBG** parts. Make sure the directions are correct!
5. On the sequence diagram, add a message that you want to convey the flow. Add the item flow to that message with the **Item Flow Manager**.
6. Add the «cont inuous» stereotype to the message, as it is a continuous rather than discrete message.
7. I use a **critical region** interaction operator to provide the context over which the continuous flow applies. Note that ordinary sequence diagram rules of ordering do not apply to continuous flows. I also apply the «cont inuous» stereotype to the interaction operator as well. In this case, the use of the «cont inuous» stereotype declares an intent but doesn't really change the execution semantics of the model.



Cameo note: The **Item Flow Manager** is best found by selecting the relation or message and selecting the item flow manager from the hovering quick tools menu that appears.

This results in the flow being defined as shown in *Figure 2.5*. You can see the flow properties in the actor and use case blocks, the types that define them, the item flows between the blocks, and the item flows on the connector relating the parts on the included internal block diagram.

All of this is to support adding the item flows in the sequence diagram (Figure 2.6):

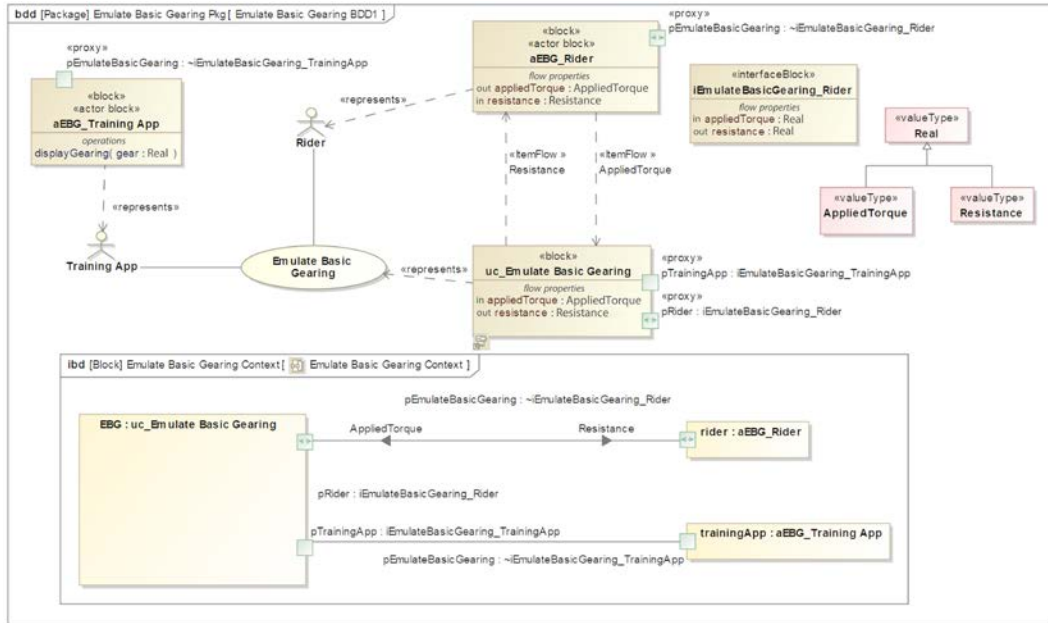


Figure 2.5: Blocks with value types and item flows added

Scenarios here are captured to show the interaction of the system with the actors using this use case. Note that continuous flows are shown as flows with the «continuous» stereotype. This resistance at a specific level is applied continuously until the level of resistance is changed. As is usual in use case analysis, messages between the actors are modeled as events, and invocation of system functions on the use case lifeline are modeled as operations.

The first scenario (Figure 2.6) shows normal gear changes from the rider. Note that the **messages to self** on the use case block lifeline indicate system functions identified during the scenario development. The state invariants show the state of the instance fulfilling the part role in the scenario at various points in the interaction. States, of course, are conditions of the instance, and may also be used to imply pre and postconditions:

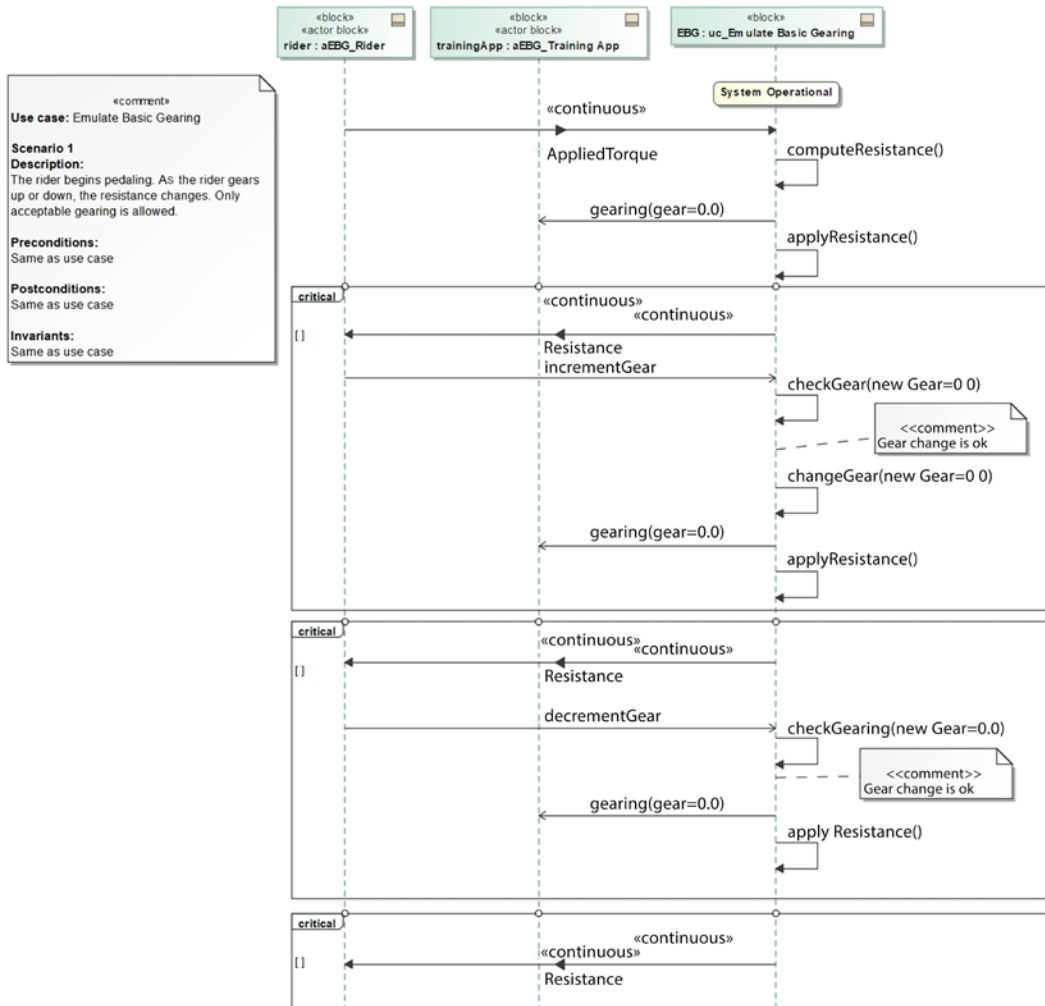


Figure 2.6: Emulate basic gearing scenario 1

The next scenario shows what happens when the rider tries to increment the gearing beyond the maximum gearing allowed by the current configuration.

It is shown in *Figure 2.7*:

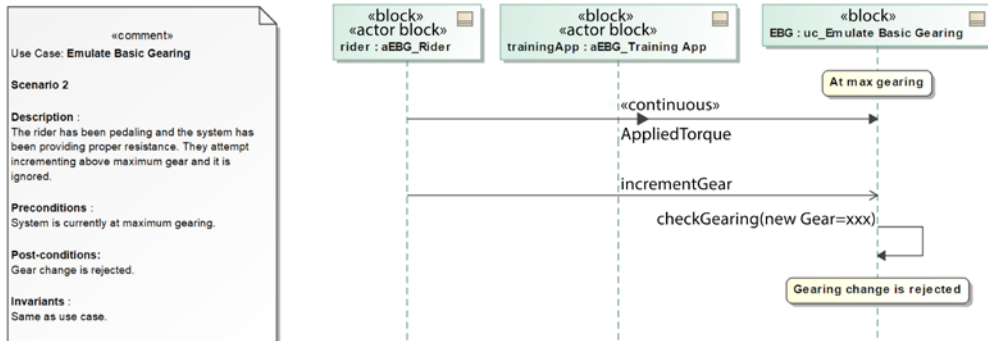


Figure 2.7: Emulate basic gearing scenario 2

The last scenario for this use case, shown in *Figure 2.8*, shows the rejection of a requested gear change below the provided gearing:

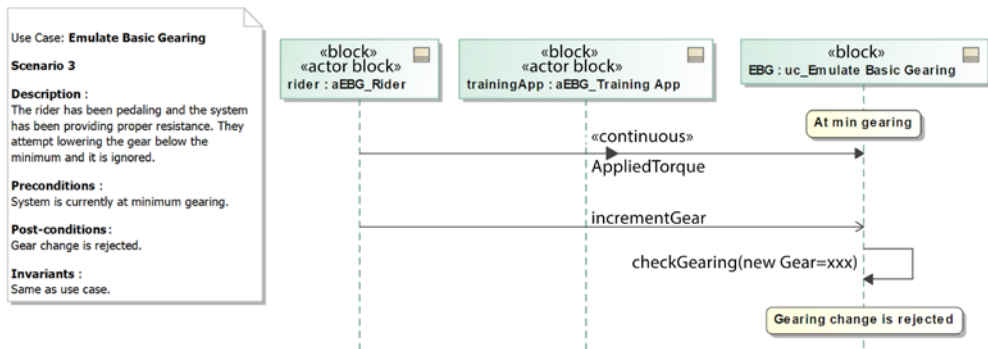


Figure 2.8: Emulate basic gearing scenario 3

Based on these sequences, we identify the following requirements:

- The system shall respond to applied pedal torque with resistance calculated from the base level of resistance, current gearing, and applied torque to simulate pedal resistance during road riding.
- The system shall send the current gearing to the training app when the current gearing changes.
- The system shall respond to a rider-initiated increase in gear by applying the new level of gearing provided that it does not exceed the maximum gearing of the gearing configuration.

- The system shall respond to a ride-initiated decrease in gear by applying the new level of gearing provided that it does not exceed the minimum gearing of the gearing configuration.

Create ports and interfaces in execution context

It is a simple matter to update the ports and interface blocks to contain the messages going between the actors and the use case. The sequence diagrams identify the messages between the use case and actor blocks, so the interface blocks must support those specific flows (*Figure 2.9*):

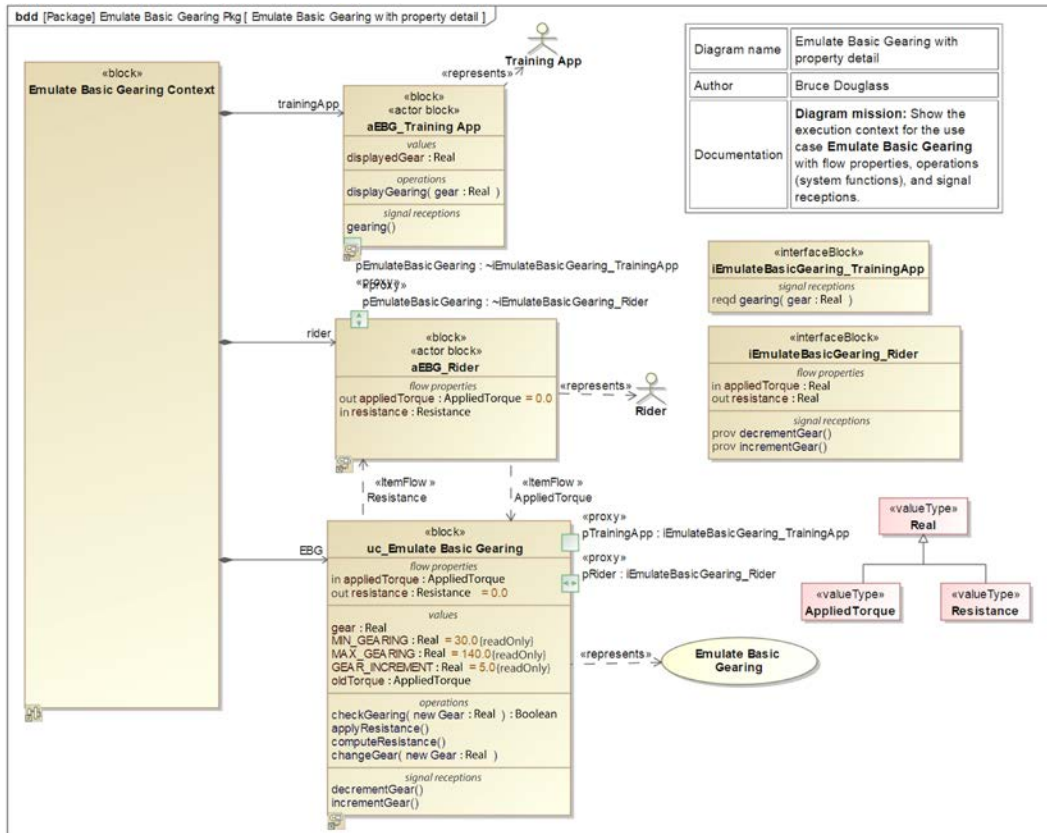


Figure 2.9: Emulate basic gearing with properties and interfaces

After updating the block and interface block definitions, we need to connect the flow properties to the port on the **Emulate Basic Gearing Context** block. This is done by showing the flow properties in both the parts and the ports and connecting them with connectors. In Cameo, this is done by right-clicking on the part or port and selecting **Display > Flow Properties** in the pop-up menu.

See Figure 2.10:

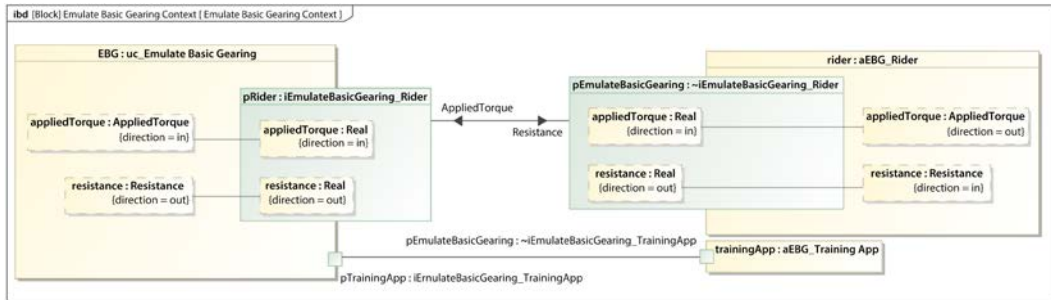


Figure 2.10: Internal block diagram with connected flow properties

Create executable state model

This step constructs the normative state machine for the use case as well as instrumenting the actors with their own state machines. The state machine of the use case block is the most interesting because it represents the requirements. Figure 2.11 shows the state machine for the **Emulate Basic Gearing** use case. The figure also shows the simple activity `send gearing to training app` that is referenced on the transition inside the **Basic Gearing** and **Handing Gear Change** states:

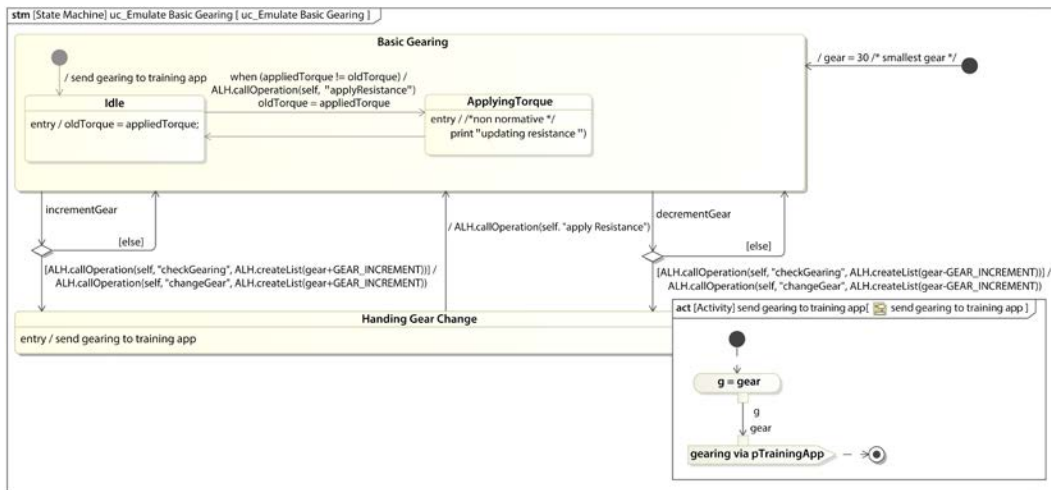


Figure 2.11: Emulate Basic Gearing State Machine

The **when ()** trigger used in the state machine is a SysML *change event*. The trigger is invoked when the expression value **appliedTorque != oldTorque** (i.e., then the value of **appliedTorque** changes) transitions from **false to true**.

To support the execution, the system functions must be elaborated enough to support the execution and simulation. These system functions include **applyResistance()**, **checkGearing()**, and **changeGear()** operations. Each of these is defined by a support opaque behavior method. *Table 2.2* shows their simple implementation.

Note that this is a “low-fidelity” simulation so the precise calculation relating resistance, torque, and gearing need not be implanted here. The designers will certainly need to do so, however:

Operation	Method	Opaque Behavior
applyResistance	applyResistance method	Resistance = gear * 100
changeGear(newGear: Real)	changeGear method (newGear: Real)	gear = newGear
checkGearing(newGear: Real): Boolean	checkGearing method(newGear:Real): Boolean	return (newGear <= MAX_GEARING) and (newGear >= MIN_GEARING)

Table 2.2: Emulate Basic Gearing operations and methods

Note that *all* opaque behaviors in this book are implemented in Cameo’s Groovy action language. To set this as the default action language for your projects, simply access the **Options > Project** menu, enter “language” in the dialog’s filter field, and set each language option to **Groovy**.

The system variable **gear** is represented as a **Real** (from the SysML value type library), representing the gear multiplier, in a fashion similar to gear inches, a commonly used measure in cycling. The flow properties **appliedTorque** and **resistance** are likewise implemented as **Reals**.

The state machines for the actor blocks are even simpler than that of the use case block. *Figure 2.12* shows the **Rider** state machine and *Figure 2.13* shows the **Training App** state machine and implementation of its **displayGearing()** function.

I have a convention that I commonly use for naming signals; external signals generated by the user running the simulation have an “ex” prefix (for external event) while internal signals generated within the context have an “ev” (for event) prefix:

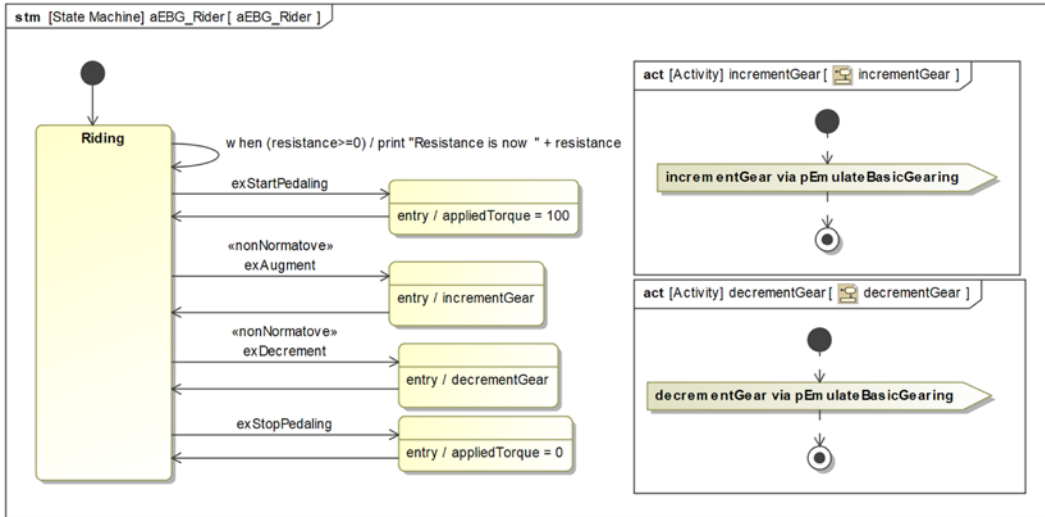


Figure 2.12: Rider Actor Block state machine

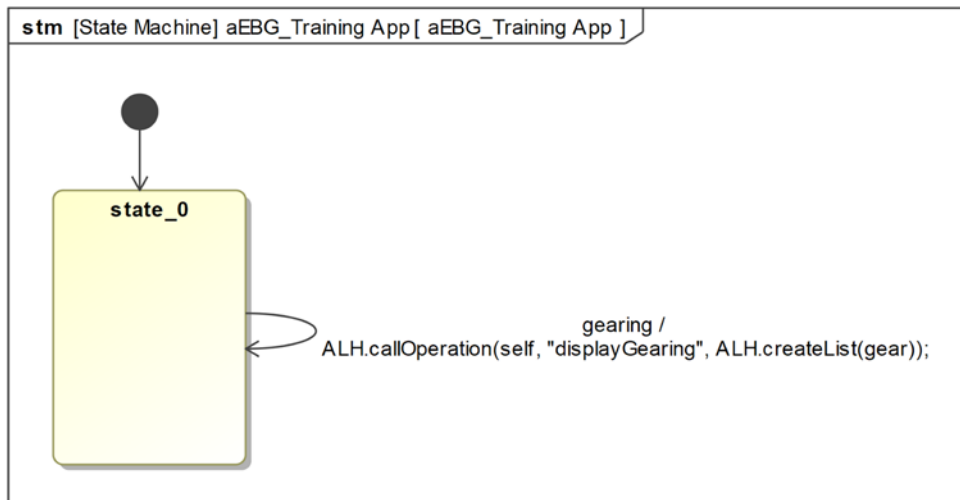


Figure 2.13: Training App Actor Block state machine

The **displayGearing** operation is realized by a **displayGearing** method with a simple implementation:

```
displayedGear = gear
print "Training App: Gear = " + displayedGear
```

Lastly, some constants are defined. **DEFAULT_GEARING** is set to the same value as **MIN_GEARING**; in this case, 30 gear inches. **MAX_GEARING** is set to about the same as a 53x10 gearing, 140. The **GEAR_INCREMENT** is used for incrementing or decrementing the gearing and is set to 5 gear inches for the purpose of simulation.

Verify and validate requirements

To facilitate control of the execution, a user interface display can be created. To do this in Cameo, the basic steps are:

1. Create a **Sim Support Pkg** package to hold it all (optional).
2. Add a Configuration diagram with a single configuration. Name this configuration **Emulate Basic Gearing Sim Config**.
3. Drag the **Emulate Basic Gearing Context** block and drop it on the simulation configuration to make that block the simulation target.
4. If you want to automatically create a sequence diagram every time you run a simulation, add a **Sequence Diagram Generator** to the diagram and then drag it onto the simulation configuration to add it as a *listener*.
5. Create a **User Interface Modeling** diagram (it's in the **expert view** of the menu).
6. Add a frame to this diagram:
 1. Drag the **Emulate Basic Gearing Context** block and drop it on this frame to set its **represents property**.
 2. In the specification for the frame, change the *title* property to **Emulate Basic Gearing UI Panel**. That will be the title of the dialog when it pops up during the simulation.
7. Add three panels:
 1. Drag the **rider** part (not the block, but the part) owned by the **Emulate Basic Gearing Context** block and drop it on the leftmost pane.
 2. Drag the **EBG** part owned by the **Emulate Basic Gearing Context** block and drop it on the middle pane.
 3. Drag the **trainingApp** part owned by the **Emulate Basic Gearing Context** block and drop it on the right pane.

8. In each pane, add **text labels** for fixed text fields, **text fields** for the values you want to display from those parts, and buttons for the events you want to insert into the running simulation.
9. Drag the relevant property from the blocks defining those parts onto the appropriate control.
10. Once you have defined the frame and its properties, drag the frame onto the simulation configuration you created in *Step 2* to add that as a UI element.

When that's done, you'll have a UI diagram that looks like *Figure 2.14* (note I added a view of the simulation configuration diagram as well):

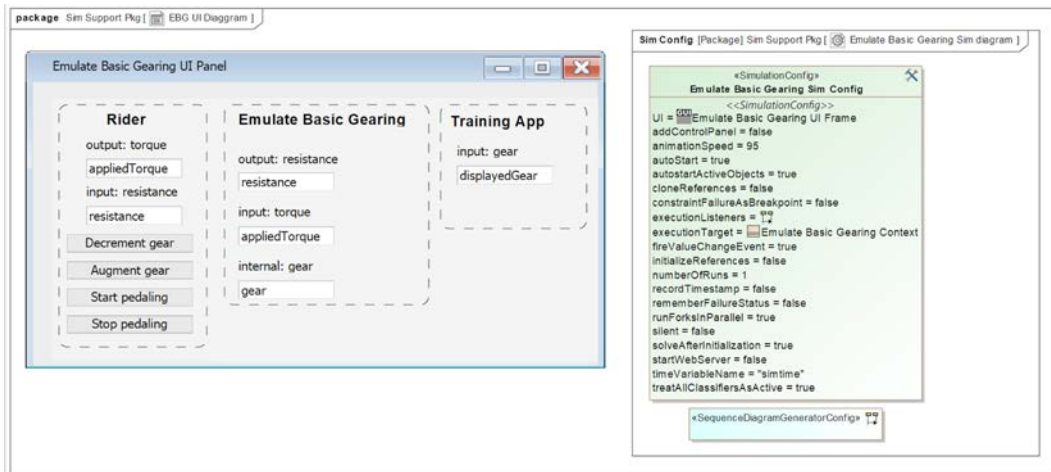


Figure 2.14: Emulate basic gearing simulation user interface

To run the simulation with Cameo's Simulation Toolkit, you'll need to select the configuration to run from the configuration list menu just below the **Analyze** menu. To run without the configuration, simply right-click the element you want to simulate and select **Simulation > Run**.

The control panel you created will pop up when you run the configuration. Then on the dialog, click **Start Pedaling**, **Augment Gear**, and **Decrement Gear**, waiting for things to stabilize between clicking.

The execution of the state model recreates the sequence diagrams. *Figure 2.15* shows the first part of the recreation of Scenario 1 (*Figure 2.6*) by the executing model. That is, the Simulation Toolkit creates the diagram from the running simulation. The user lifeline (created by Cameo during the simulation run) represents you, the simulation driver, as you insert events by clicking buttons or otherwise entering events.

The assignments in curly braces denote values being changed, whether through operation execution or the flow of values across the ports:

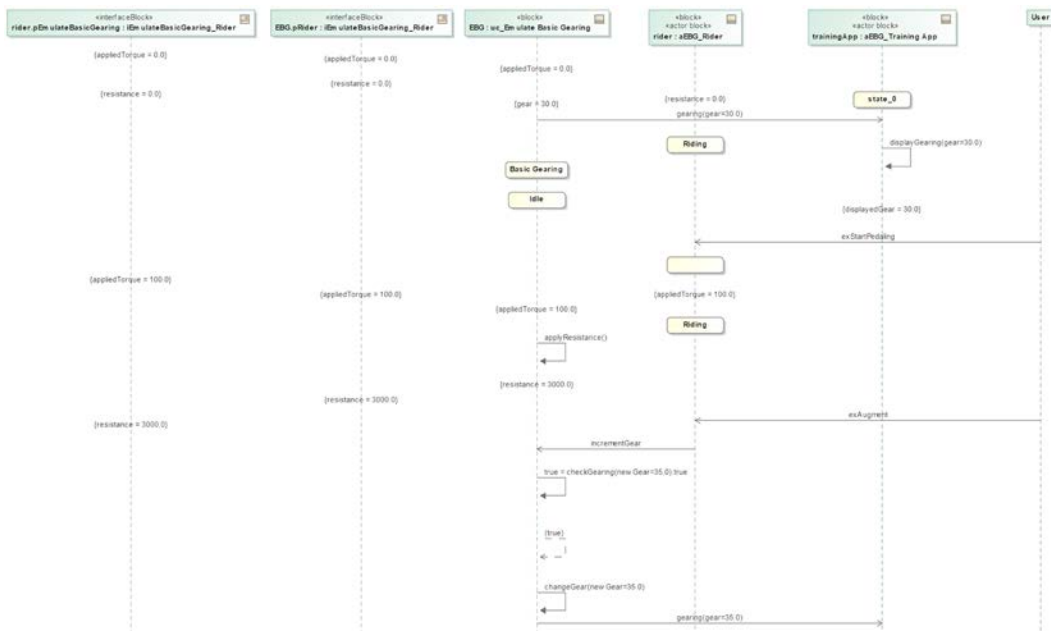


Figure 2.15: Animated sequence diagram from model simulation

While in review, the project lead notices that there is no requirement initial starting value for the gearing before a specific gear has been selected. Additionally, we see that the requirement to notify the training app was missing. These are identified missing requirements that must be added. In general, any message flowing into the use case indicates requirements to receive a request or flow, and then invoke an internal behavior (system function) to handle the request or process the flow. Any message flowing from the use case to an actor indicates requirements to send information, results, or flows to the actor. Messages from the use case to itself indicate requirements for a transformation of some kind.

Requirements_change

In this example, we notice that we omitted a requirement to update the rider display of the gearing. The change has already been made to the state machine.

Update requirement set

We add the following requirements to the requirements set:

- The system shall display the currently selected gear.

- The system shall default to the minimum gear during initialization.

Add trace links

In this case, we ensure there are trace links back to stakeholder requirements as well as from the use case to the requirements. This is shown in the use case diagram in *Figure 2.16*. The newly identified requirements are highlighted with a bold border:

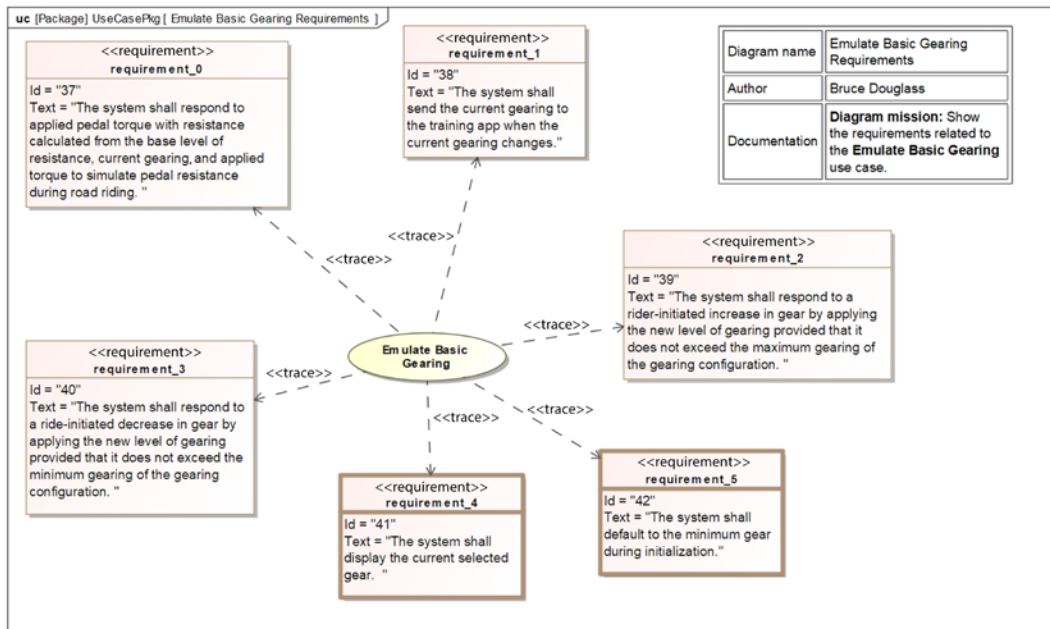


Figure 2.16: Emulate basic gearing requirements

Perform use case and requirements review

The requirements model can now be reviewed by relevant stakeholders. The work products that should be included in the review include all the diagrams shown in this section, the requirements, and the executing model. The use of the executing model allows for what-if examination of the requirements set to be easily done during the review. Such questions as “What happens to the gearing if the Rider turns the system off and back on?” or “What is the absolute maximum gearing to be allowed?” can be asked. Simulation of the model allows the questions to either be answered by running the simulation case or can be identified as an item that requires resolution.

Functional analysis with activities

Functional analysis can be performed in subtly different ways. In the previous recipe, we started with the sequence diagram to analyze the use case. That is particularly useful when the interesting parts of the use case are the interactions. The workflow in this recipe is slightly different, although it achieves exactly the same objectives. This workflow starts with the development of an activity model and generates scenarios from that. In this recipe, just as in the previous one, when the work is all complete, it is the state machine that forms the normative specification of the use case; the activity diagram is used as a stepping stone along the way. The objective of the workflow, as with the previous recipe, is to create an executable model to identify and fix defects in the requirements, such as missing requirements, or requirements that are incomplete, incorrect, or inaccurate. Overall, this is the most favored workflow among model-based systems engineers.

Purpose

The purpose of the recipe is to create a set of high-quality requirements by identifying and characterizing the key system functions performed by the system during the execution of the use case capability. This recipe is particularly effective when the main focus of the use case is the set of system functions and not the interaction of the system with the actors.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view.

Outputs and postconditions

There are several outcomes, the most important of which is a set of requirements accurately and appropriately specifying the behavior of the system for the use case. Additional outputs include an executable use case model, logical system interfaces to support the use case behavior and a supporting logical data schema, and a set of scenarios that can be used later as specifications of test cases.

How to do it

Figure 2.17 shows the workflow for this recipe. It is similar to the previous recipe. The primary difference is that rather than beginning the analysis by creating scenarios with the stakeholders, it begins by creating an activity model of the set of primary flows from which the scenarios will be derived:

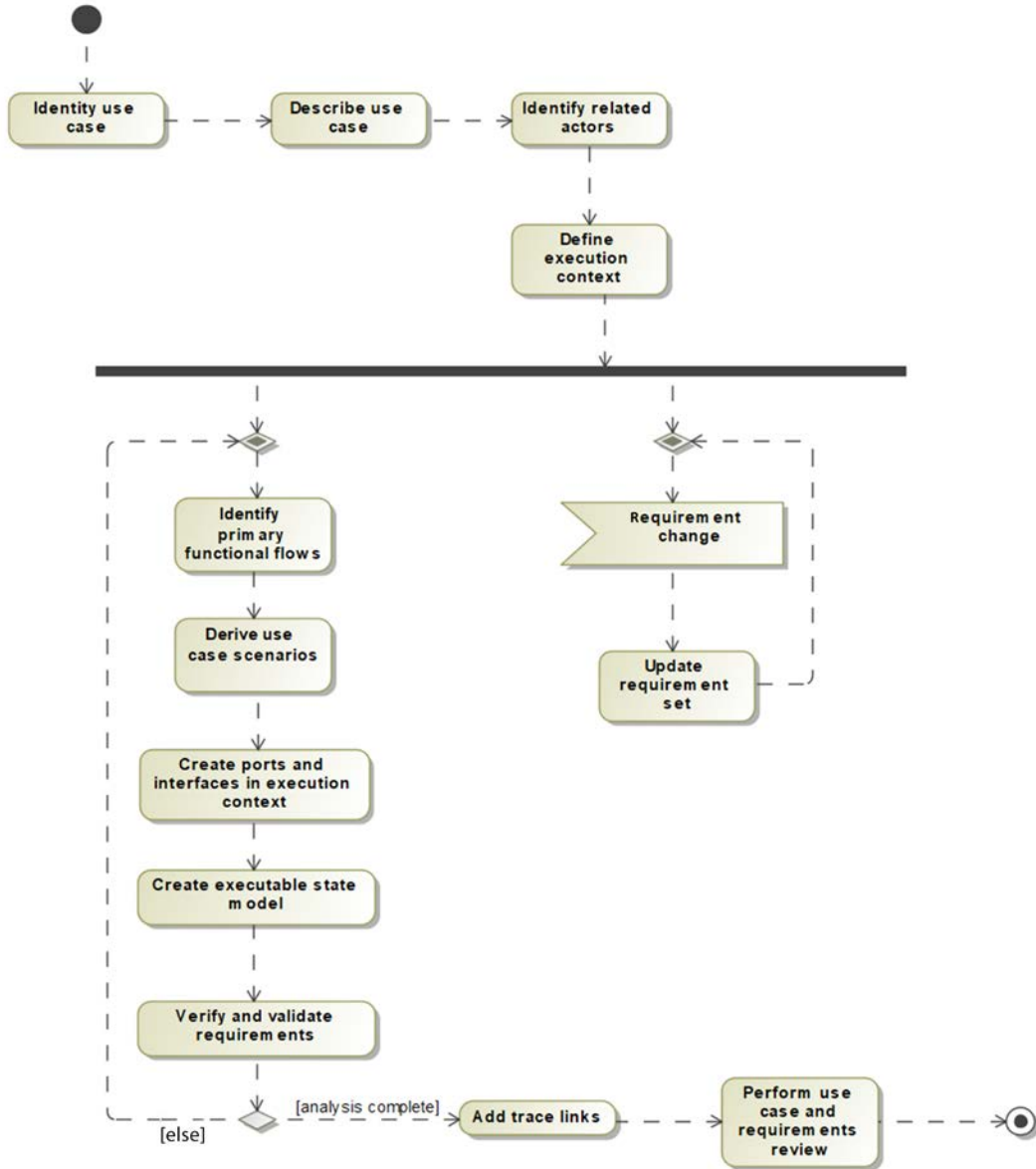


Figure 2.17: Functional Analysis with Activities

Identify use case

This first step is to identify the generic usage of which the scenarios of interest, user stories, and requirements are aspects.

Describe use case

The description of the use case should include its purpose, and a general description of the flows, preconditions, postconditions, and invariants (assumptions). Some modelers add the specific actors involved, user stories, and scenarios, but I prefer to use the model itself to contain those relations.

Identify related actors

The related actors are those people or systems outside our scope that interact with the system while it executes the current use case. These actors can send messages to the system, receive messages from the system, or both.

Define execution context

The execution context is a kind of modeling “sandbox” that contains an executable component consisting of elements representing the use case and related actors. The recommended way to achieve this is to create separate blocks representing the use case and the actors, connected via ports. Having an isolated simulation sandbox allows different system engineers to progress independently on different use case analyses.

Identify primary functional flows

The activity model identifies the functional flows of the system while it executes the use case capability. These consist of a sequenced set of actions, connected by control flows, with control nodes (notably, decision, merge, fork, and join nodes) where appropriate. In this specific recipe step, the focus is on the primary flows of the system – also known as “sunny day” flows – and less on the secondary and fault scenarios (known as “rainy day” scenarios). The actions are either system functions, reception of messages from the actors, sending a message to the actors, or waiting for timeouts.

This activity model is not complete in the sense that it will not include all possible flows within the use case. The activity diagram *can* be made complete, but it is usually easier to do that with a state machine. If you prefer to work entirely in the activity diagram, then evolve the activity model to be executable rather than develop a state machine for this purpose. The later recipe step *Create executable state model* will include all flows, which is why the state machine, rather than the activity model, is the normative specification of the use case. This activity model allows the systems engineer to begin reasoning about the necessary system behavior. Most systems engineers feel very comfortable with activity models and prefer to begin the analysis here rather than with the scenarios or with the state machine.

Derive use case scenarios

The activity model identifies multiple flows, as indicated by control nodes, such as decision nodes. A specific scenario takes a singular path through the activity flow so that a single activity model results in multiple scenarios. The scenarios are useful because they are easy to review with non-technical stakeholders and because they aid in the definition of the logical interfaces between the system and the actors.

Create ports and interface in execution context

Once we have a set of scenarios, we've identified the flow from the use case to the actors and from the actors to the system. By inference, this identifies ports relating the actors and the system, and the specific flows within the interfaces that define them.

Create executable state model

This step identifies what I call the “normative state machine.” Executing this state machine can recreate each of the scenarios we drew in the *Capture use case scenarios* step. All states, transitions, and actions represent requirements. Any state elements added only to assist in the execution that do not represent requirements should be stereotyped as «non-normative» to clearly identify this fact. It is also common to create state behavior for the actors in a step known as “instrumenting the actor” to support the execution of the use case in the execution context.

Verify and validate requirements

Running the execution context for the use case allows us to demonstrate that our normative state machine in fact represents the flows identified by working with the stakeholder. It also allows us to identify flows and requirements that are missing, incomplete, or incorrect. These result in *Requirements_change* change requests to fix the identified requirements defects.

Requirements_change

Parallel to the development and execution of the use case model, we maintain the textual requirements. This workflow event indicates the need to fix an identified requirements defect.

Update requirement set

In response to an identified requirements defect, we fix the textual requirements by adding, deleting, or modifying requirements. This will then be reflected in the updated model.

Add trace links

Once the use case model and requirements stabilize, we add trace links using the «trace» relation or something similar. These relations allow the backtrace to stakeholder requirements as well as forward links to any architectural elements that might already exist. For relations forward to design elements, we generally prefer «satisfy».

Perform use case and requirements review

Once the work has stabilized, a review for correctness and compliance with standards may be done. This allows subject matter experts and stakeholders to review the requirements, use cases, activities, states, and scenarios for correctness, and for quality assurance staff to ensure compliance with modeling and requirements standards.

Example

The example used for this recipe is the **Control Resistance** use case, shown in *Figure 2.18* along with some other use cases:

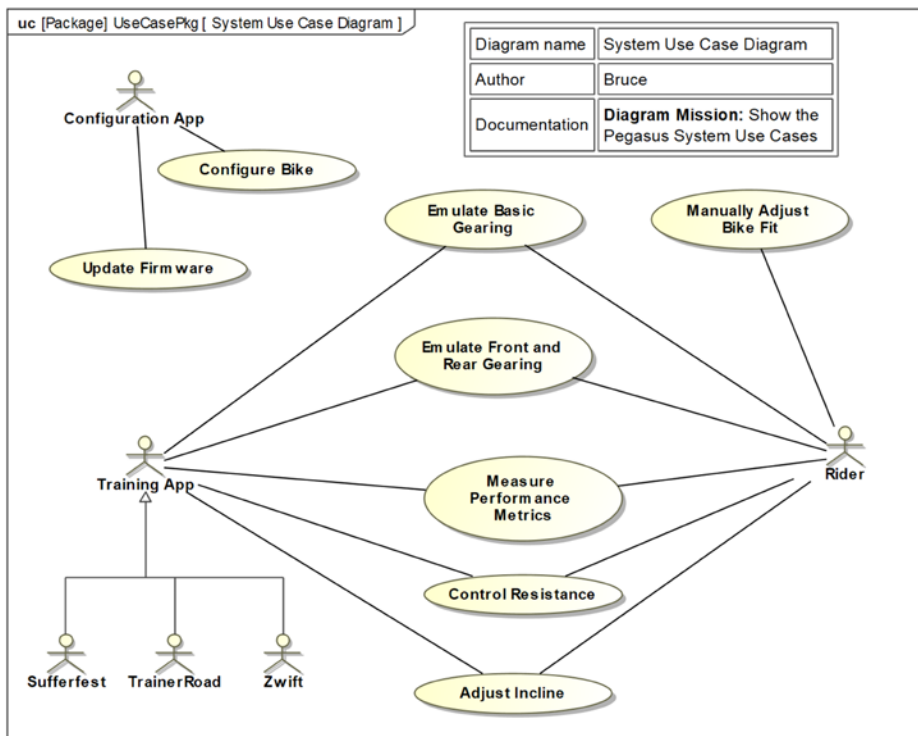


Figure 2.18: Use Cases for Analysis

Describe use case

All model elements deserve a useful description. In the case of a use case, we typically use a format as shown in *Figure 2.19*. To show an element's description on a diagram in Cameo, attach a note, right-click the note, and select **Text Display Mode > Show Documentation**:

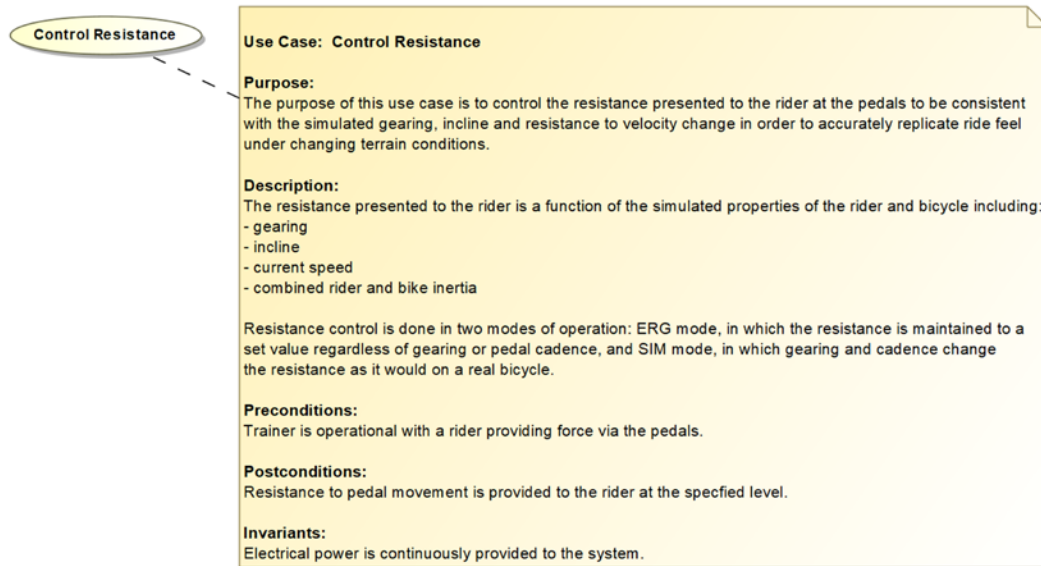


Figure 2.19: Control Resistance Use Case Description

Identify related actors

The related actors in this example are the **Rider** and the **Training App**. The rider signals the system to change the gearing via the gears control and receives a response in terms of changing resistance as well as setting resistance mode to ERG or SIM mode. The training app, when connected, is notified of the current gearing so that it can be displayed, provides a simulated input of incline, and can, optionally, change between SIM and ERG modes. The relation of the actors to the use case is shown in *Figure 2.18*.

Define execution context

The execution context creates blocks that represent the actors and the use case for the purpose of the analysis. In this example, the following naming conventions are observed:

- The block representing the use case has the use case name (with white space removed) preceded by **uc**. Thus, for this example, the use case block is named **ucControlResistance**.

- Blocks representing the actors are given the actor name preceded with “a” and an abbreviation of the use case. For this use case, the prefix is aCR_ so the actor blocks are named aCR_Rider and aCR_TrainingApp.
- The interface blocks are named i + <use case block>_<actor block>. The names of the two interface blocks are iControlResistance_Rider and iControlResistance_TrainingApp. The normal form of the interface block is associated with the proxy port on the use case block; the conjugated form is associated with the corresponding proxy port on the actor block.

All these elements are shown on the internal block diagram in *Figure 2.20*:

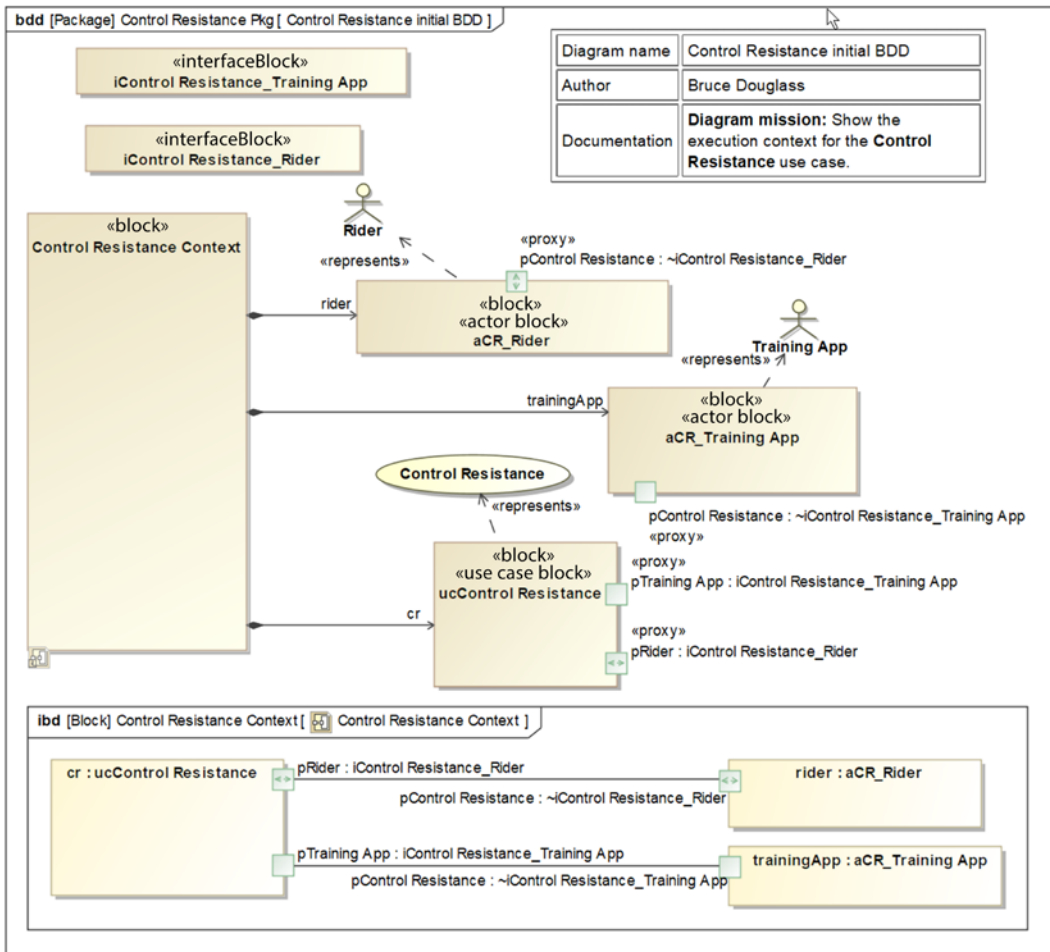


Figure 2.20: Control Resistance Execution Context

Identify primary functional flow

This step creates an activity model for the primary flows in the use case. We will create an activity diagram to get us started, but we will also create a state machine for the actual simulation. We do this because state machines are usually more appropriate for use case behaviors, but activity diagrams allow us to get started with the analysis more easily. Thus, the activity diagram is *notional* but the state machine is *normative*. This is a common way that system engineers do use case analysis. In Cameo, a block can own multiple behaviors. In our example here, the block will own both the activity diagram and the state machine but the **classifier behavior** of the block will be the state machine. This can be set in the block's **Specification Dialog**.

The flow consists of a set of steps sequenced by control flows and mediated by a set of control nodes. The actual system generates resistance in two different ways: **SIM mode**, in which the resistance is generated as a function of gearing, pedal cadence, force applied to the pedal, and simulated values of bike inertia, speed, acceleration, and drag; and **ERG mode**, in which a constant power is generated by the user by the system adjusting the resistance to pedal cadence.

The high-level flow is shown in *Figure 2.21*. Note that some actions – such as *Read Self* – are in the **Any Actions** list in the Cameo activity diagram toolbar:

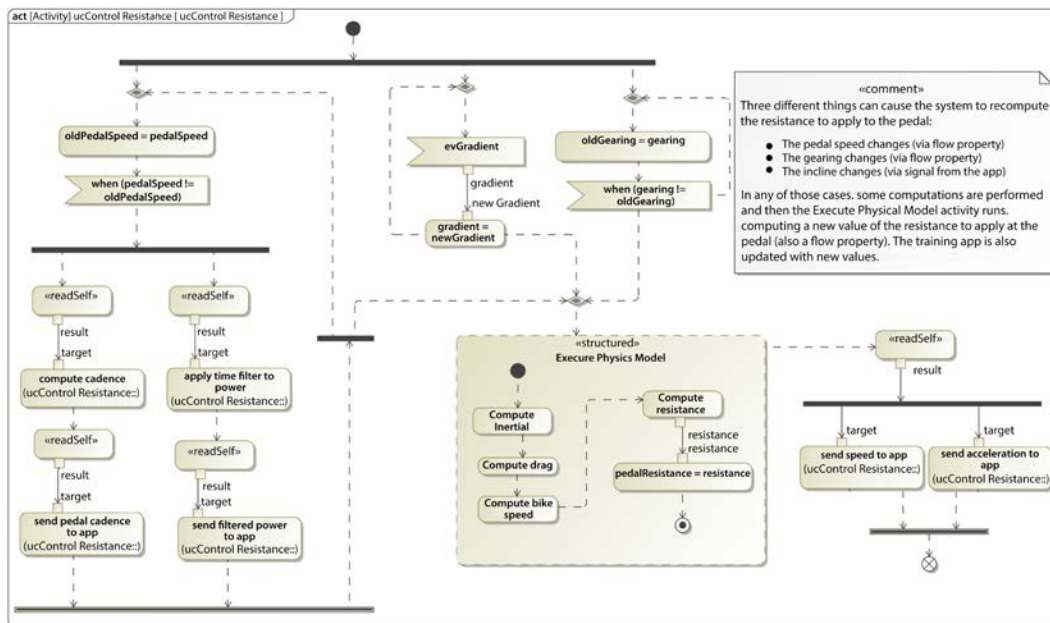


Figure 2.21: Activity for Compute Resistance

Directly from the activity diagram, we can identify several requirements, shown in tabular form in *Figure 2.22*:














#	Name	Text
1	 51 CR_requirement_9	The system shall accept simulated incline from the connected training app throughout a training session.
2	 63 CR_requirement_11	The system shall accept road incline from the connected training app.
3	 56 CR_requirement_14	The system shall compute and apply resistance to pedal movement based on simulated inertia, speed, acceleration and rider power input.
4	 54 CR_requirement_12	The system shall compute simulated drag based on computed rider inertia and simulated road incline.
5	 55 CR_requirement_13	The system shall compute simulated road speed based on computed inertia, current simulated speed and acceleration, and rider-applied force to the pedal.
6	 49 CR_requirement_7	The system shall provide time-filtering of power, supporting 0, 1-second, 3-second, and 5-second power averaging, settable by the rider.
7	 48 CR_requirement_6	The system shall measure force applied by the rider to the pedals.
8	 47 CR_requirement_4	The system shall compute pedal cadence from pedal speed
9	 43 CR_requirement_0	The system shall send pedal cadence to the associated training app, when connected.
10	 50 CR_requirement_8	The system shall send filtered power to the training app, if connected.
11	 57 CR_requirement_15	The system shall send computed speed and acceleration to the connected training app, if any.
12	 52 CR_requirement_10	The system shall store get the gearing from the rider controls.
13	 44 CR_requirement_1	The system shall store the rider weight for the computation of inertia

Figure 2.22: Control Resistance initial requirements

Derive use case scenarios

The activity flow in *Figure 2.21* can be used to create scenarios in sequence diagrams. It is typical to create a set of scenarios such that each control flow is shown at least once.

This is called the *minimal-spanning set* of scenarios. In this case, because of the nature of parallelism, a high-level scenario (Figure 2.23) is developed with more detailed flows put on reference scenarios:

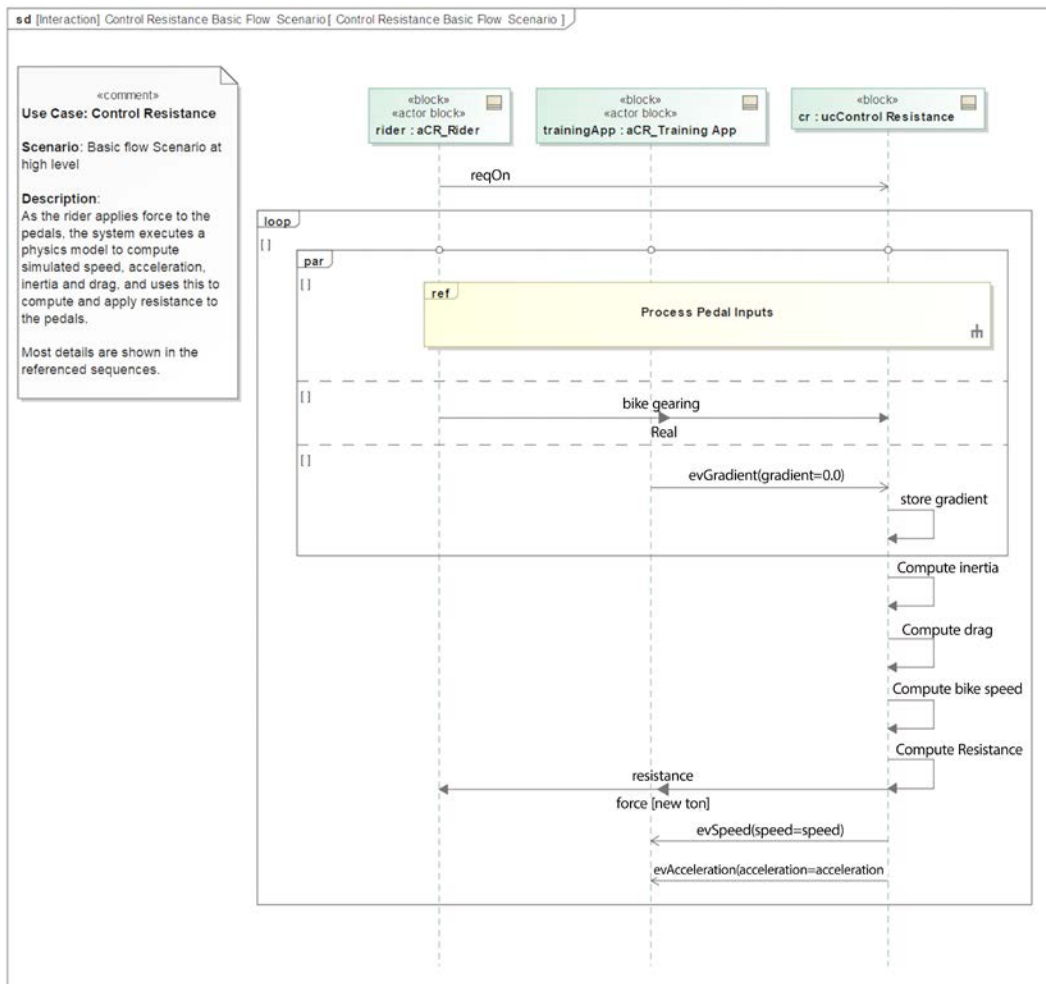


Figure 2.23: Compute Resistance Main Scenario

The first reference scenario (Figure 2.24) reflects the inputs, gathered via system sensors, of the pedal status. This part of the overall scenario flow provides the necessary data for the computation of resistance:

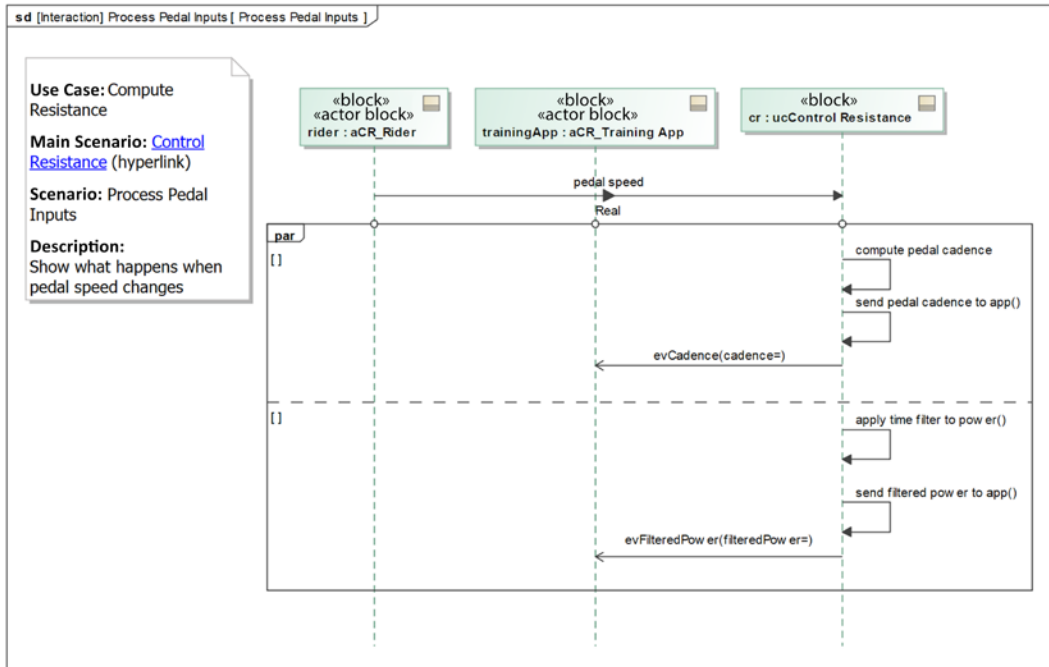


Figure 2.24: Process Pedal Inputs Scenario

Create ports and interfaces in execution context

Now that we have defined some interactions between the system and the actors, we can elaborate on the interfaces to support those message exchanges. This shows the flow properties supported by actor and use case blocks and the interfaces. The interfaces are defined so that the use case block is the unconjugated side and shows not only the flow properties that pass through the ports but also the signal receptions from the sequence diagrams.



In the example shown, several flow properties are typed by elements in the ISO-80000 SI standard units library. To get this library, select “Use Basic Units Library” in the *New Project* dialog when you create your model. RPM is not in the standard units library, but can be easily added as a specialization of the Real type from the SysML primitive types library.

This is shown in the BDD in *Figure 2.25*:

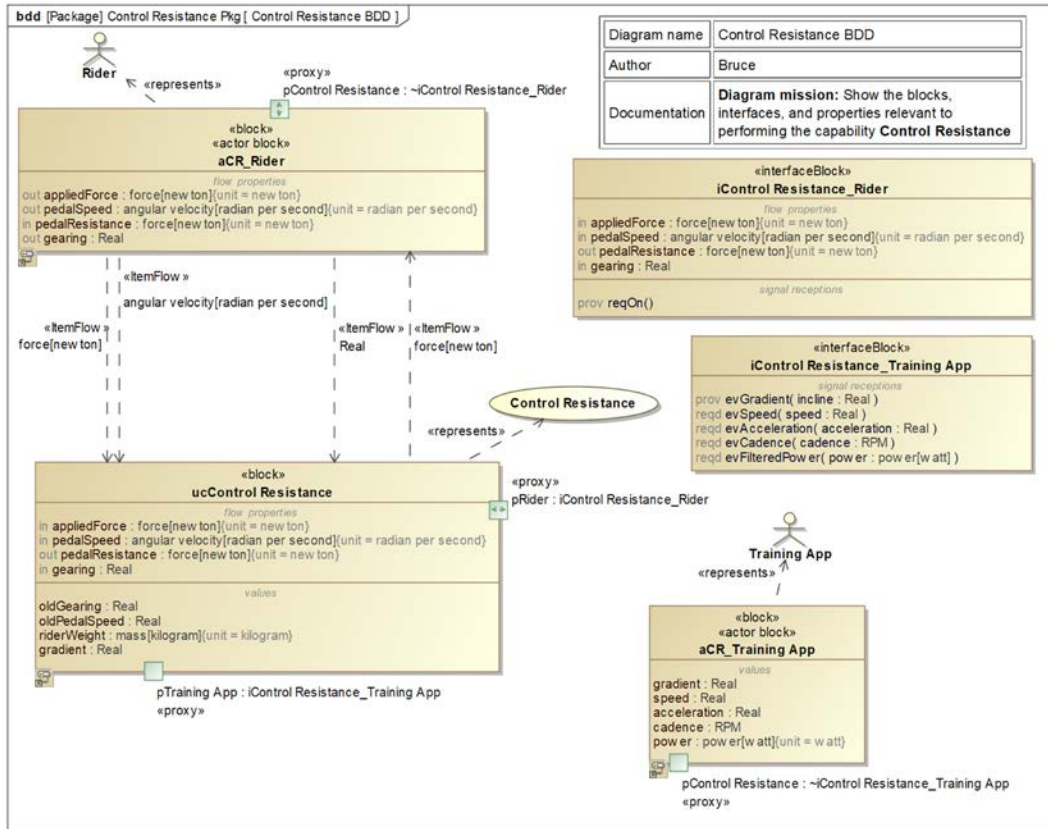


Figure 2.25: Compute resistance interfaces

Create executable state model

Because we added the activity diagram first, Cameo made that the **classifier behavior** for the use case block. We will need to change the **classifier behavior** property of this block to the state machine in the **specification dialog** for the use case block for the execution to proceed.

Figure 2.26 shows the state machine for the use case **Control Resistance**:

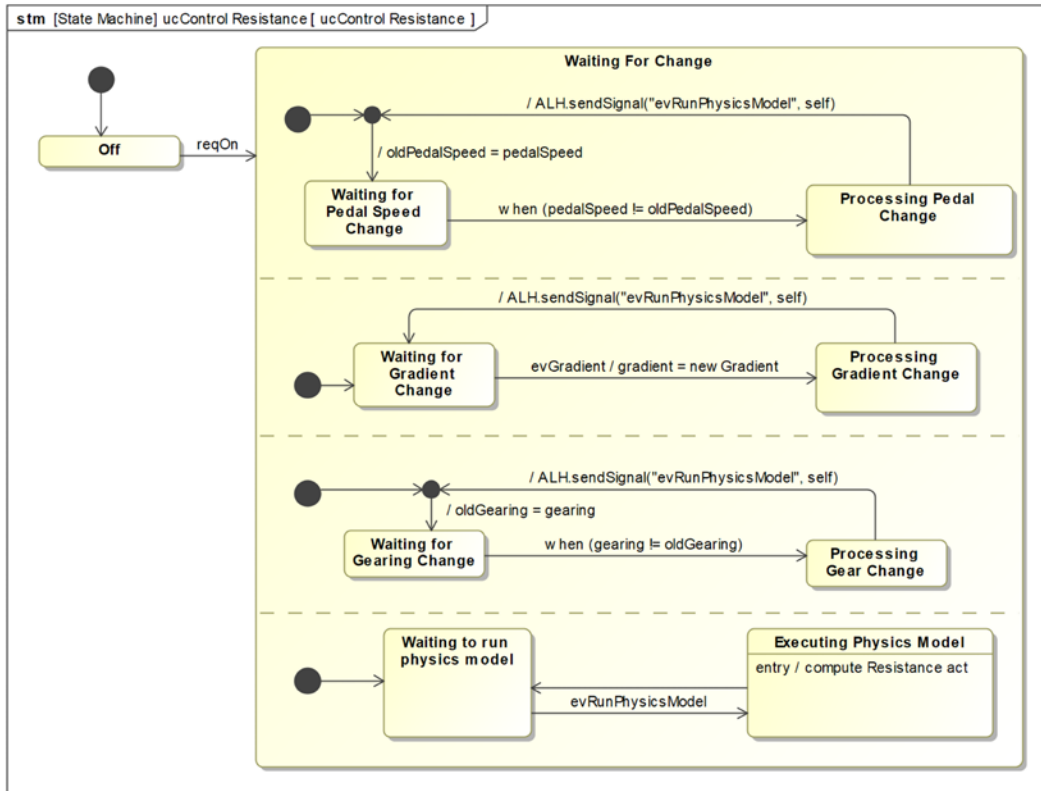


Figure 2.26: Control resistance use case state machine

This state machine uses orthogonal regions. Two of these regions use **change events**, as indicated by **when (<condition>)**. Change events are triggered when the condition changes from false to true. In both uses, the old value of a flow property is stored, and the change event transition fires when the current value of the flow property changes from its old value. In this way, the transitions are triggered when either the gearing is changed or when the pedal speed changes. Another and-state has a transition triggered when the signal **evGradient** is received from the **Training App**, indicating that the gradient from the virtual terrain has changed.

In all these cases, a signal is sent that causes the physics model to be reevaluated, resulting in a changed resistance at the pedal. *Figure 2.27* shows a very simplified behavior to update the pedal resistance, speed, acceleration, cadence, and power. Several of these values are passed to the **Training App** via signals. Note that this physics model is not correct, but getting the physics right is not relevant to the purpose of this simulation. Certainly, the designers must create a proper physics model.

We are doing a low-fidelity simulation for the purpose of ensuring that the right system functions are being called at the right time and the correct interactions between the system and the actors. This model is adequate for that purpose:

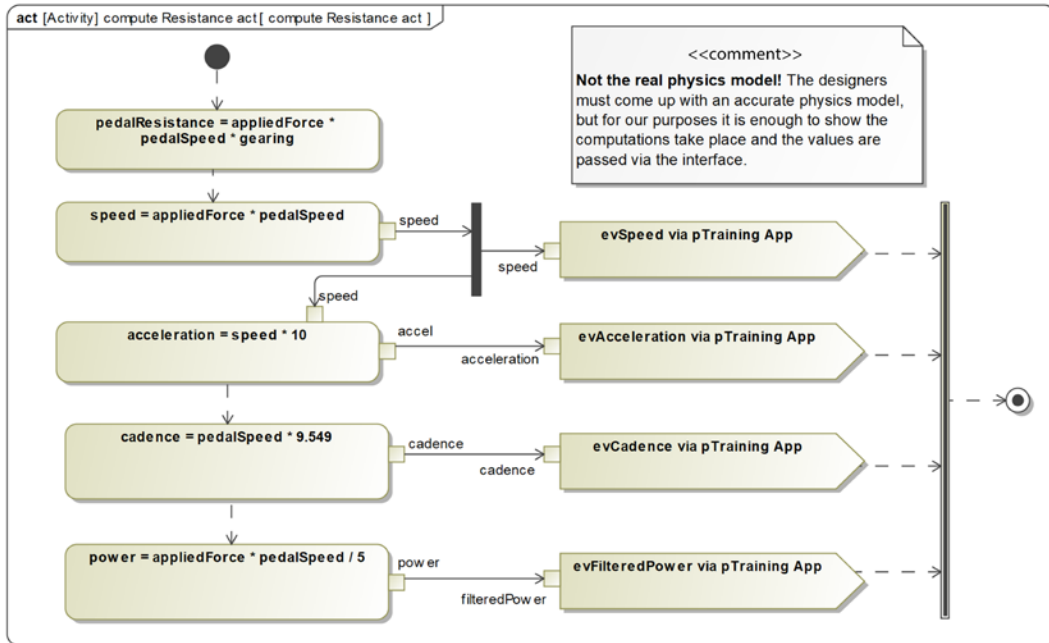


Figure 2.27: Compute resistance activity

We also need to “instrument the actors” for simulation support. A simple state behavioral model for the **Rider** is shown in Figure 2.28 and the state machine for the **Training App** is shown in Figure 2.29:

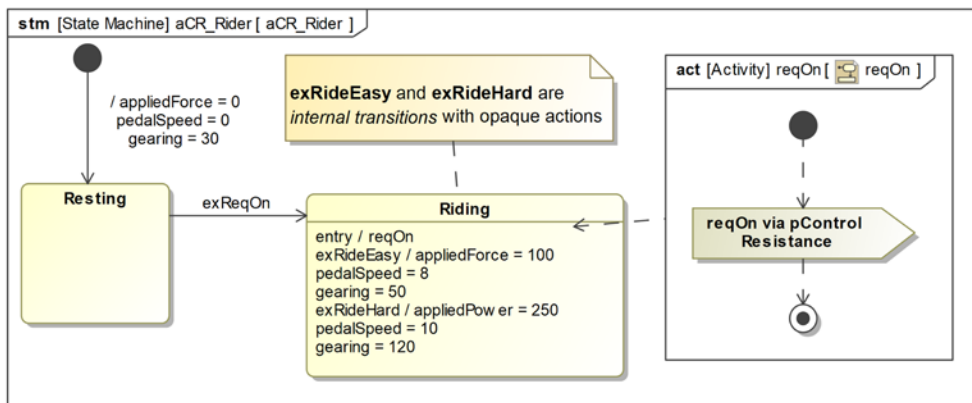


Figure 2.28: Rider state machine

For the purposes of the simulation, two user events were added: **exRideEasy** and **exRideHard**. These just set the flow properties **appliedPower**, **pedalSpeed**, and **gearing** to values representative of those conditions. You can always set the flow property values individually during simulation if desired. The **aCR_Rider** block processes those events as **internal transitions** in the **Riding** state:

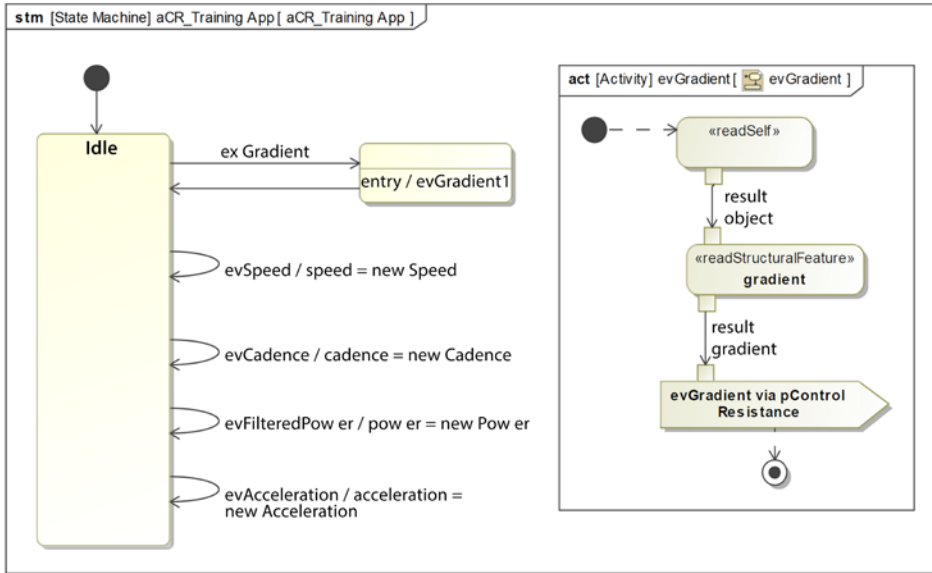


Figure 2.29: Training app state machine for the control resistance use case

Before we run the simulation, we also need to connect the flow properties on the **Control Resistance Context** internal block diagram. This is done by adding connectors between the flow properties in the ports (defined by the interface blocks) and the flow properties in the blocks. See Figure 2.30:

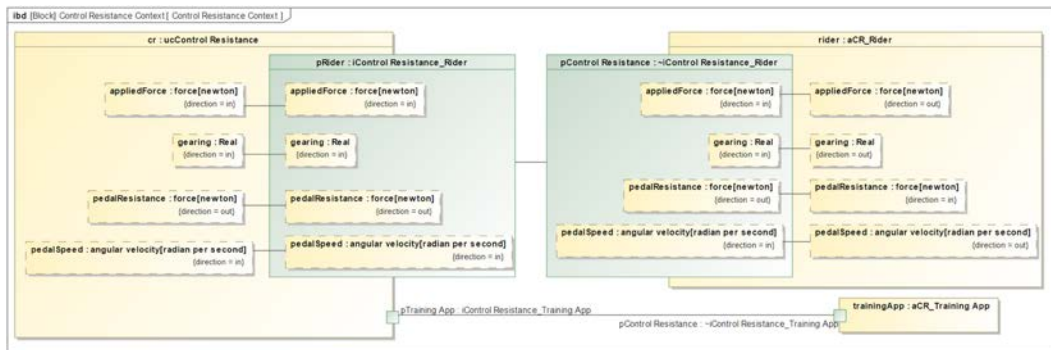


Figure 2.30: Connecting the flow properties on the IBD

Verify and validate requirements

The simulation is not meant to be a high-fidelity physics simulation of all the forces and values involved, but really to give a medium-fidelity simulation to help validate the set of requirements and to identify missing or incorrect ones. A control panel was created to visualize the behavior and provide input of values (*Figure 2.31*) using the same techniques as in the previous recipe. This is done in a Cameo “User Interface Modeling Diagram,” which is just a repurposed package diagram with specialized controls. The text controls are bound to the flow and value properties in the different blocks. The buttons send external (simulation user) events to the elements to make things happen:

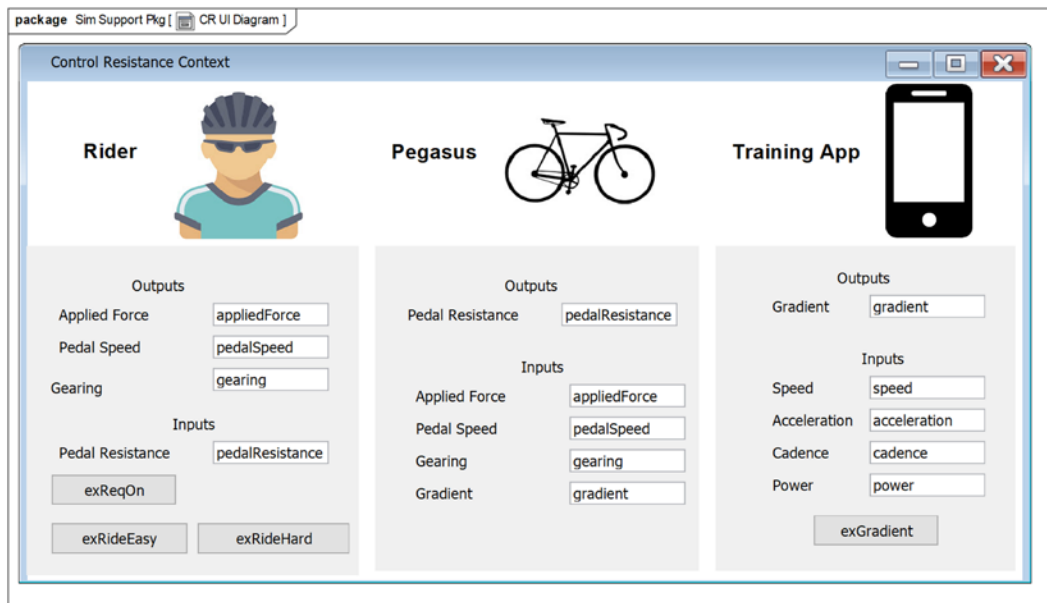


Figure 2.31: Control Resistance simulation user interface

This frame is used as the UI for a simulation configuration named **Control Resistance Sim Config**. I’ve also added a sequence diagram listener so that when the configuration is run, an animated sequence diagram will be created. You can also see in *Figure 2.32* that the **Control Resistance Context** is set as the **execution target**:

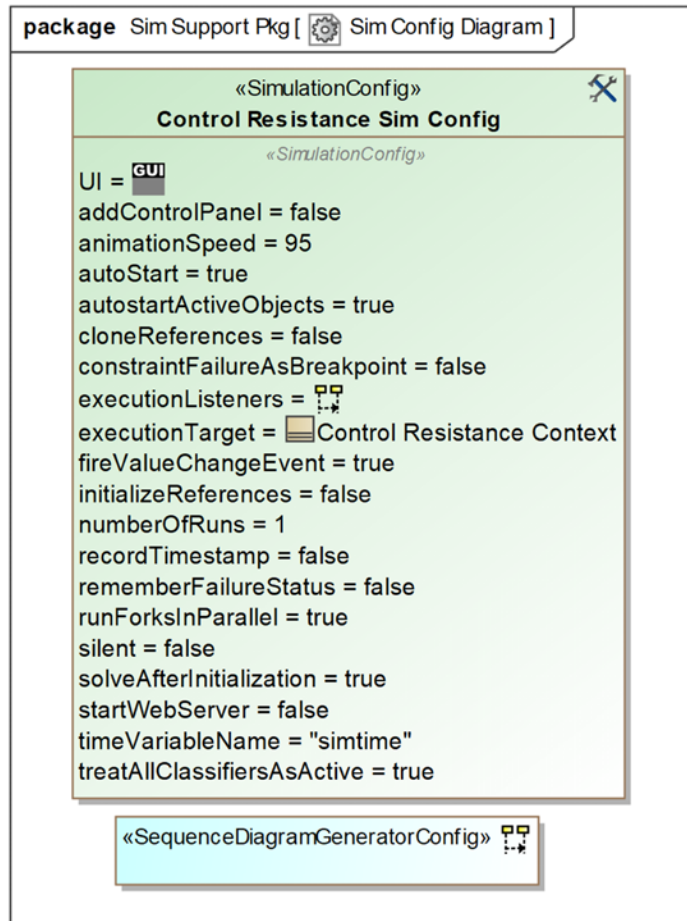


Figure 2.32: Control resistance simulation configuration

Simulation of difference scenarios results in many sequence diagrams capturing the behavior, such as the (partial) one shown in *Figure 2.33*.

The User lifeline is created by Cameo to represent you, as a simulation operator, and show when you insert events into the running simulation:

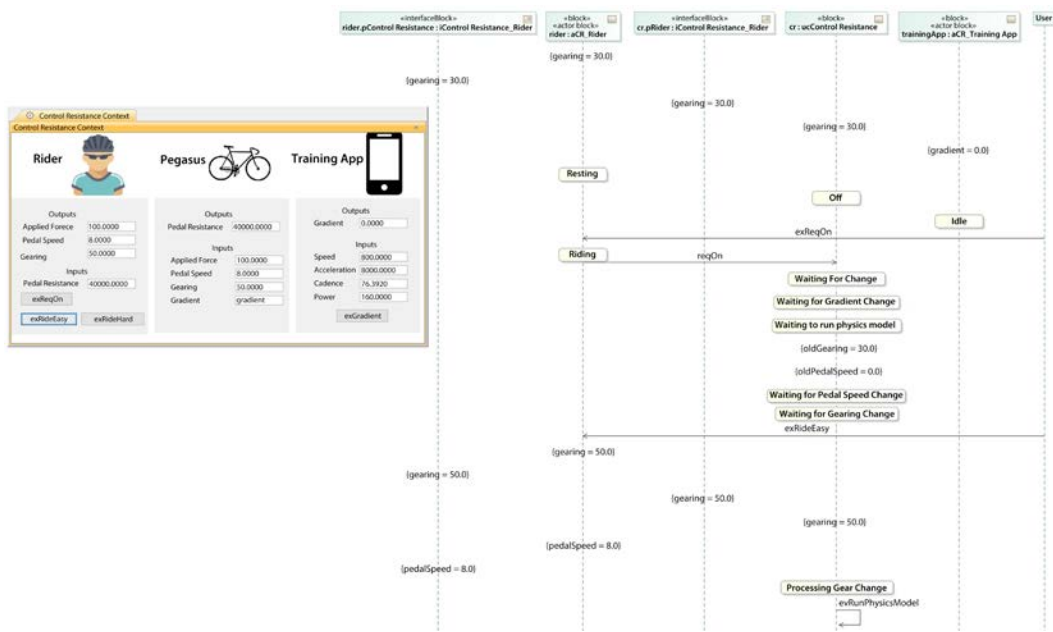


Figure 2.33: (Partial) animated sequence diagram example of control resistance use case

Requirements_change

A number of minor requirements defects are identified and flagged to be added to the requirements set.

Update requirements set

The creation and execution of the use case simulation uncovers a couple of new requirements related to timing:

- The system shall update the physics model frequently enough to provide the rider with a smooth and road-line experience with respect to resistance.
- The system shall update the training app with pedal cadence at least every 1.0 seconds.
- The system shall update the training app with rider-filtered power output at least every 0.5 seconds.
- The system shall update the training app with simulated bike speed at least every 1.0 seconds.

Also, we discover a missing data transmission to the training app:

- The system shall send current watts per kilogram to the training app for the current power output at least every 1.0 seconds.

Add trace links

Trace links are updated in the model. The trace links for this and the previous example are shown in matrix form in *Figure 2.34*:

UseCasePkg	Req requirement_0	Req requirement_1	Req requirement_2	Req requirement_3	Req requirement_4	Req requirement_5	Additional	CR CR_req	CR CR_req	CR CR_req	CR CR_req	CR CR_req	CR CR_req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	Initial CR Req	
Control Resistance	1	1	1	1	1	1	15	5	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Emulate Basic Gearing	6	→	→	→	→	→			→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→	→

Figure 2.34: Control Requirements use case requirements trace matrix

Perform use case and requirements show

The requirements model can now be reviewed by relevant stakeholders. The work products that should be included in the review include all the diagrams shown in this section, the requirements, and the executing model. The use of the executing model allows for what-if examination of the requirements set to be easily done during the review. Such questions as “How quickly does the resistance control need to be updated to simulate the road riding experience?” or “What is the absolute maximum resistance supported to be allowed?”. Simulation of the model allows the questions to either be answered by running the simulation case or can be identified as an item that requires resolution.

Functional analysis with state machines

Sometimes beginning with the state machine is the best approach to do use case analysis. This is particularly true when the use case is obviously “modal” in nature with different operational modes. This approach generally requires systems engineers who are very comfortable with state machines. The recipe is much like the previous use case analyses and can be used instead; the output is basically the same for all three of these recipes.

The primary differences are that no activity diagram is created and the sequence diagrams are created from the executing use case state behavior.

Purpose

The purpose of the recipe is to create a set of high-quality requirements by identifying and characterizing the key system functions performed by the system during the execution of the use case capability. This recipe is particularly effective when the use case is clearly modal in nature and the system engineers are highly skilled in developing state machines.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view.

Outputs and postconditions

There are several outcomes, the most important of which is a set of requirements accurately and appropriately specifying the behavior of the system for the use case. Additional outputs include an executable use case model, logical system interfaces to support the use case behavior and a supporting logical data schema, and a set of scenarios that can be used later as specifications of test cases.

How to do it

Figure 2.35 shows the workflow for this recipe. It is similar to the previous recipe. The primary difference is that rather than beginning the analysis by creating scenarios with the stakeholders, it begins by creating a state machine model of the set of primary flows from which the scenarios will be derived:

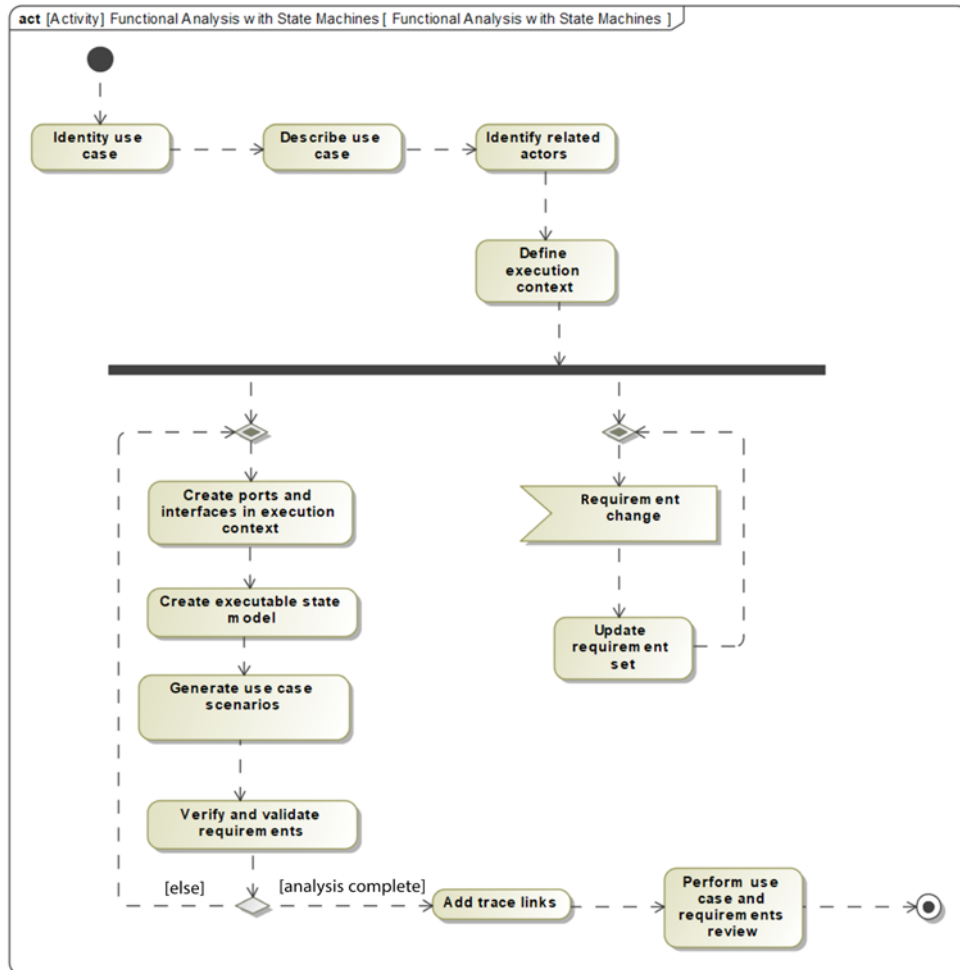


Figure 2.35: Functional analysis with states

Identify use case

The first step is to identify the generic usage of which the scenarios of interest, user stories, and requirements are aspects.

Describe use case

The description of the use case should include its purpose, and a general description of the flows, preconditions, postconditions, and invariants (assumptions). Some modelers add the specific actors involved, user stories, and scenarios, but I prefer to use the model itself to contain those relations.

Identify related actors

The related actors are those people or systems outside our scope that interact with the system while it executes the current use case. These actors can send messages to the system, receive messages from the system, or both.

Define execution context

The execution context is a kind of modeling “sandbox” that contains an executable component consisting of executable elements representing the use case and related actors. The recommended way to achieve this is to create separate blocks representing the use case and the actors, connected via ports. Having an isolated simulation sandbox allows different system engineers to progress independently on different use case analyses.

Create ports and interface in execution context

In this step, we create or update the ports and their defining interface blocks to specify the flows between the use case block and the actors.

Create executable state model

This step creates a state machine that represents the textual requirements in the formal language of state machines. Executing this state machine can recreate each of the scenarios we drew in the *Capture use case scenarios* step of the *Functional Analysis with Scenarios* recipe. Almost all states, transitions, and actions represent requirements. Any state elements added only to assist in the execution that do not represent requirements should be stereotyped as «non-normative» to clearly identify this fact. It is also common to create state behavior for the actors in a step known as “instrumenting the actor” to support the execution of the use case in the execution context.

Generate use case scenarios

The state model identifies multiple flows, driven by event receptions and transitions, executing actions along the way. A specific scenario takes a singular path through the state flow so that a single state machine model results in multiple scenarios. The scenarios are useful because they are easy to review with non-technical stakeholders and because they aid in the definition or refinement of the logical interfaces between the system and the actors. Because the state machine is executable, it can be automatically created from the execution of the state machine if you are using a supportive tool.

Verify and validate requirements

Running the execution context for the use case allows us to demonstrate that our normative state machine in fact represents the flows identified by working with the stakeholder. It also allows us to identify flows and requirements that are missing, incomplete, or incorrect. These result in *Requirements_change* change requests to fix the identified requirements defects.

Requirements_change

Parallel to the development and execution of the use case model, we maintain the textual requirements. This workflow event indicates the need to fix an identified requirements defect.

Update requirement set

In response to an identified requirements defect, we fix the textual requirements by adding, deleting, or modifying requirements. This will then be reflected in the updated model.

Add trace links

Once the use case model and requirements stabilize, we add trace links using the «trace» relation or something similar. These relations allow the backtrace to stakeholder requirements as well as forward links to any architectural elements that might already exist. For relations forward to design elements, we generally prefer «satisfy».

Perform use case and requirements review

Once the work has stabilized, a review for correctness and compliance with standards may be done. This allows subject matter experts and stakeholders to review the requirements, use cases, states, and scenarios for correctness, and for quality assurance staff to ensure compliance with modeling and requirements standards.

Example

The example used for this recipe is the **Emulate Front and Rear Gearing** use case. This use case is shown in *Figure 2.36* along with some closely related use cases:

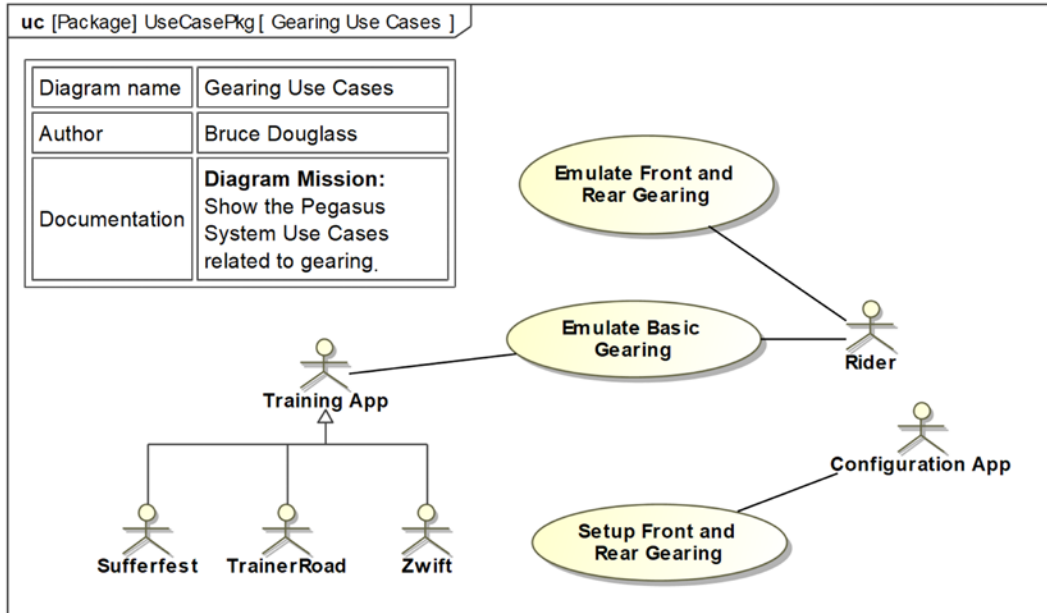


Figure 2.36: Emulate front and rear gearing use cases

Describe use case

All model elements deserve a useful description. In the case of a use case, we typically use a format as shown in *Figure 2.37*:

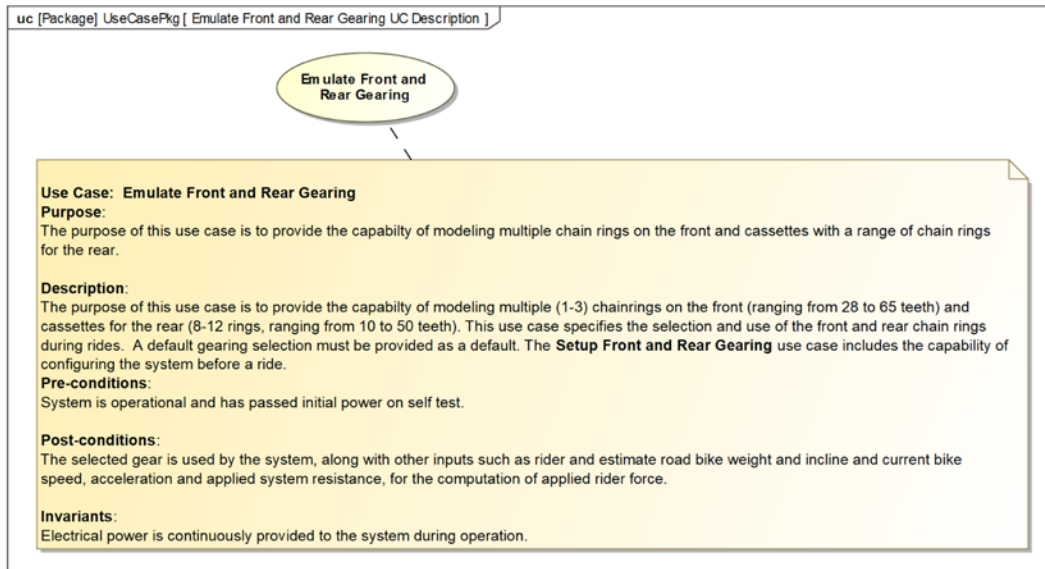


Figure 2.37: Emulate front and rear gearing use case description

Identify related actors

The related actors in this example are the **Rider** and the **Training App**. The rider signals the system to change the gearing via the gears control and receives a response in terms of changing resistance as well as setting resistance mode to ERG or SIM mode. The training app, when connected, is notified of the current gearing so that it can be displayed, provides a simulated input of incline, and can, optionally, change between SIM and ERG modes. The relation of the actors to the use case is shown in *Figure 2.36*.

Define execution context

The execution context creates blocks that represent the actors and the use case for the purpose of the analysis. In this example, the following naming conventions are observed:

- The block representing the use case has the use case name (with white space removed) preceded by **uc_**. Thus, for this example, the use case block is named **uc_EmulateFrontandRearGearing**.
- Blocks representing the actors are given the actor name preceded with “a” and an abbreviation of the use case. For this use case, the prefix is **aEFRG** so the actor block is named **aEFRG_Rider**.

- The interface blocks are named `i + <use case block>_<actor block>`. The name of the interface block is `iuc_EmulateFrontandRearGearing_aERFG_Rider`. The normal form of the interface block is associated with the proxy port on the use case block; the conjugated form is associated with the corresponding proxy port on the actor block.

All these elements are shown on the internal block diagram in *Figure 2.38*:

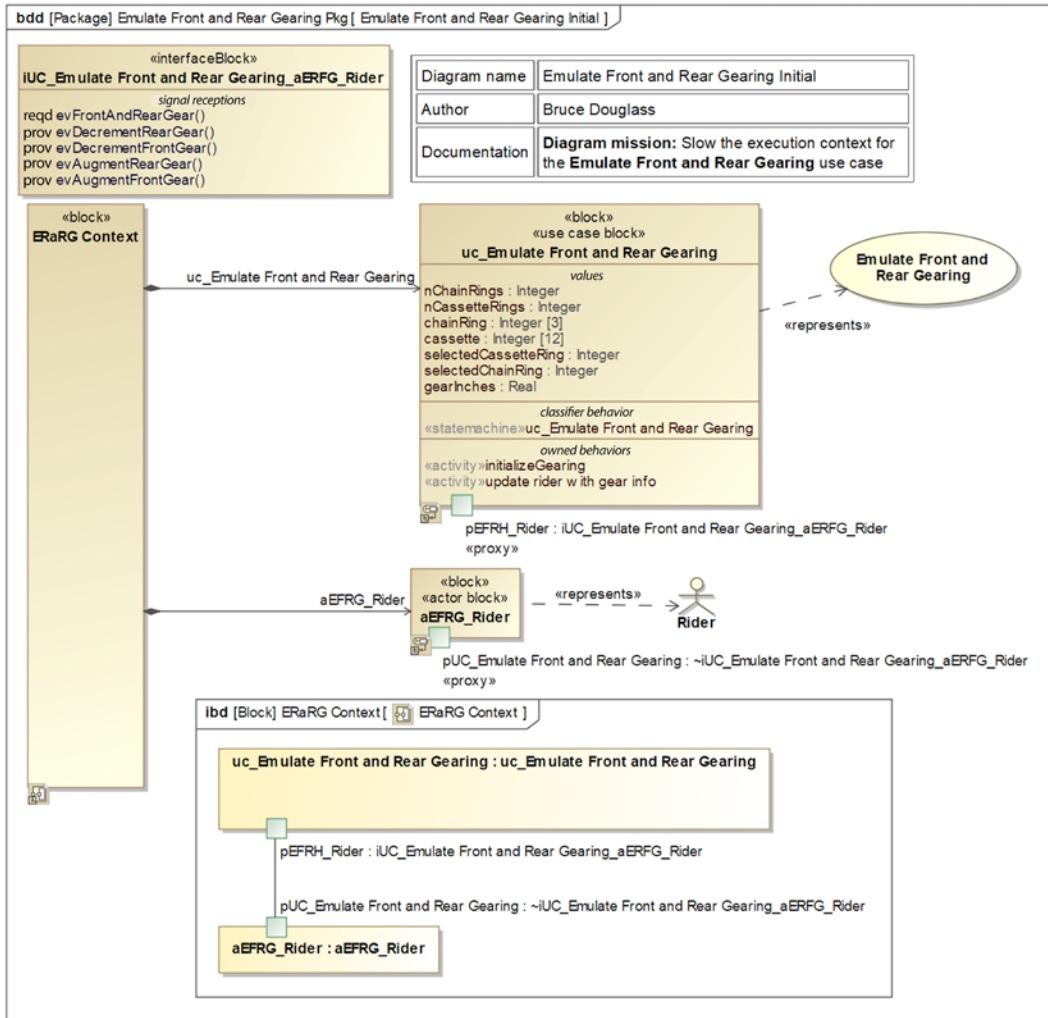


Figure 2.38: Emulate Front and Rear Gearing use case execution context

Create ports and interface in execution context

The (empty) ports and interfaces are added between the use case block and the actor blocks, as shown in *Figure 2.38*. These will be elaborated on as the development proceeds in the next step.

Create executable state model

Figure 2.39 shows the state machine for the use case **Emulate Front and Rear Gearing**. It is important to remember that the state machine is a restatement of textual requirements in a more formal language and *not* a declaration of design. The purpose of creating this state machine during this analysis is to identify requirement defects, not to design the system:

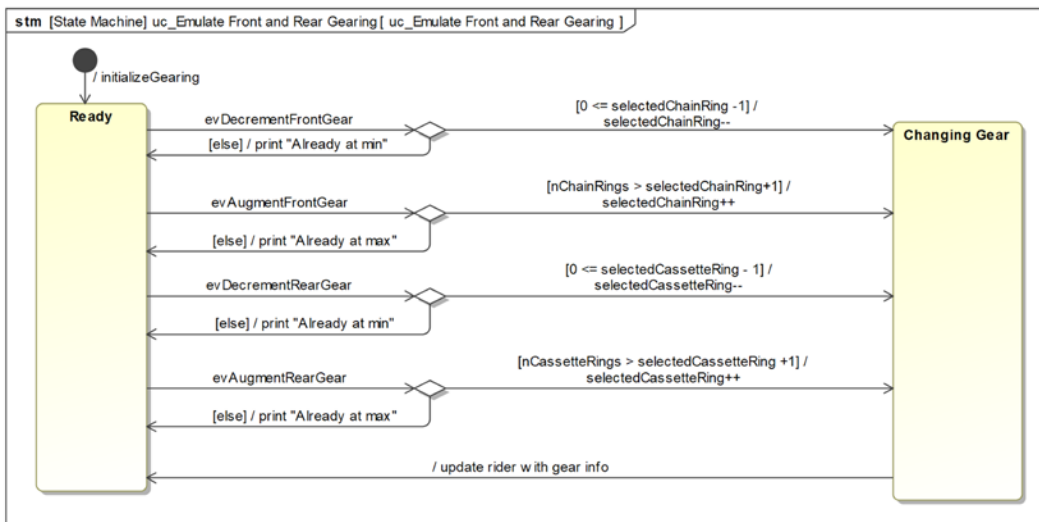


Figure 2.39: State machine for Emulate Front and Rear Gearing

The state machine starts off initializing the gearing and sending that gearing to the **Rider** (presumably via a display on the bike). This activity uses a loop node to initialize the set of 12 rear cassette cogs. Using the loop node in Cameo is a little tricky. One key is that the action in the test compartment returns a Boolean value named **result**, and this is set as the **Decider** property of the loop node.

See Figure 2.40:

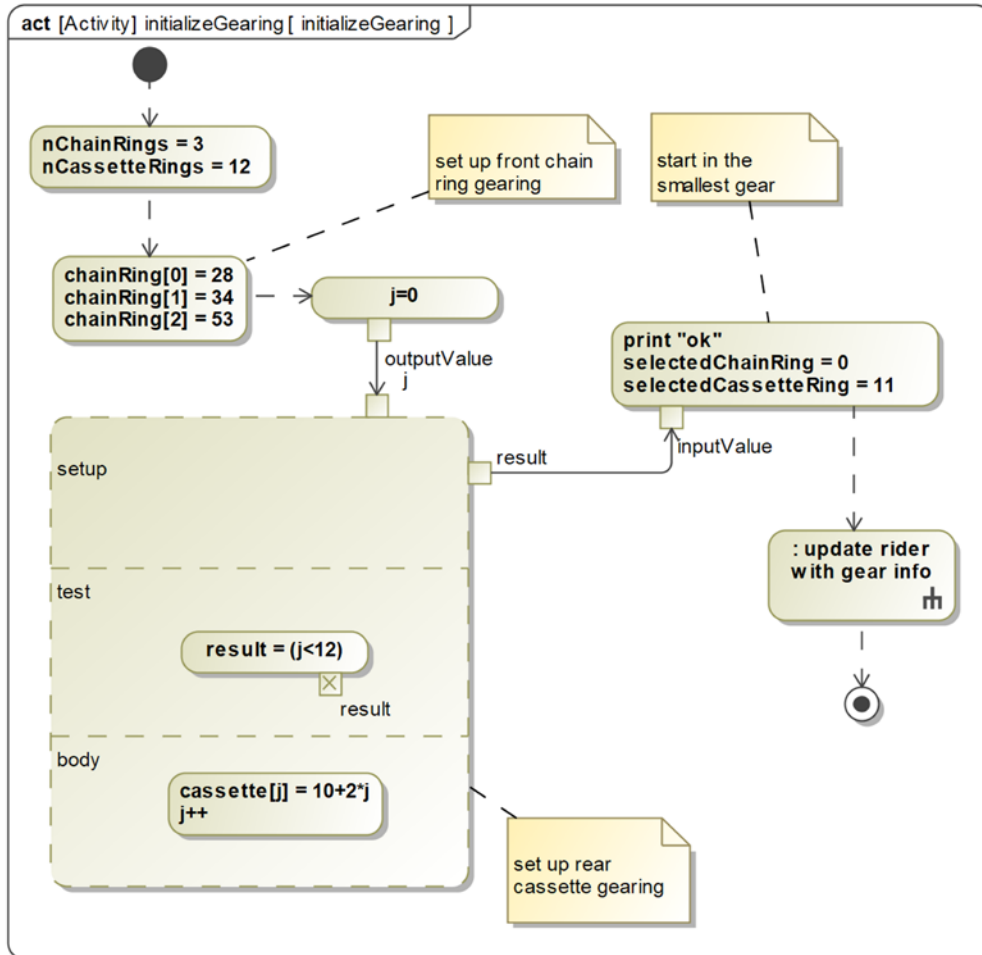


Figure 2.40: Initialize Gearing Activity

This activity invokes another via a **call behavior action** named **update rider with gear info**. This latter activity is also invoked from the main behavior state machine when gears are changed. It updates a value property known as **gear inches**; this is the distance the bike travels with one turn of the pedals. It also sends the gearing information to the rider. It is shown in *Figure 2.41*:

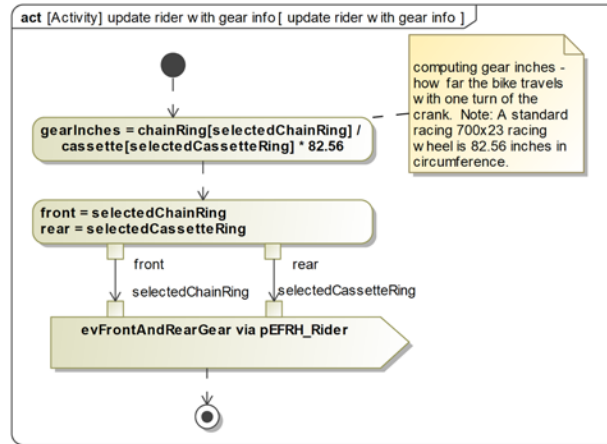


Figure 2.41: Update rider with gear info activity

The **Rider** state machine accepts the **evFrontAndRearGear** signal from the use case block to update it when the gears have changed. All this block does with this information is print the gearing to the console during the simulation. It also accepts the **ex** signals from the user running the simulation and sends these signals out of the port that is connected to the use case part. In this way, the **Rider** actor block can gear up and down for both the front chain ring and rear cassette:

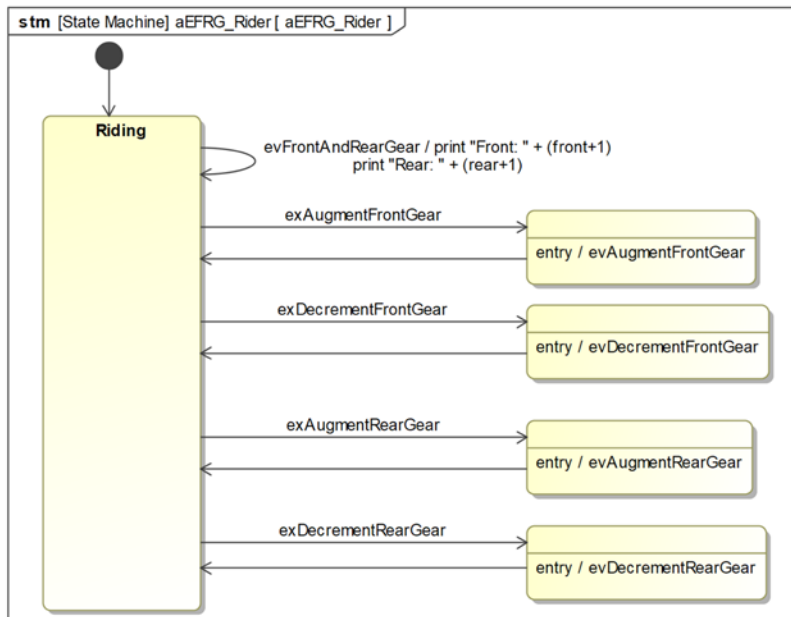


Figure 2.42: Rider actor block state machine

Similar to the example in the last recipe, I created a simple UI to allow easier control of the simulation and visualization of the result. This UI is shown in *Figure 2.43*. The UI uses text fields for scalar values and list control for the front chain ring and rear cassette arrays. It defines the UI for a simulation configuration:

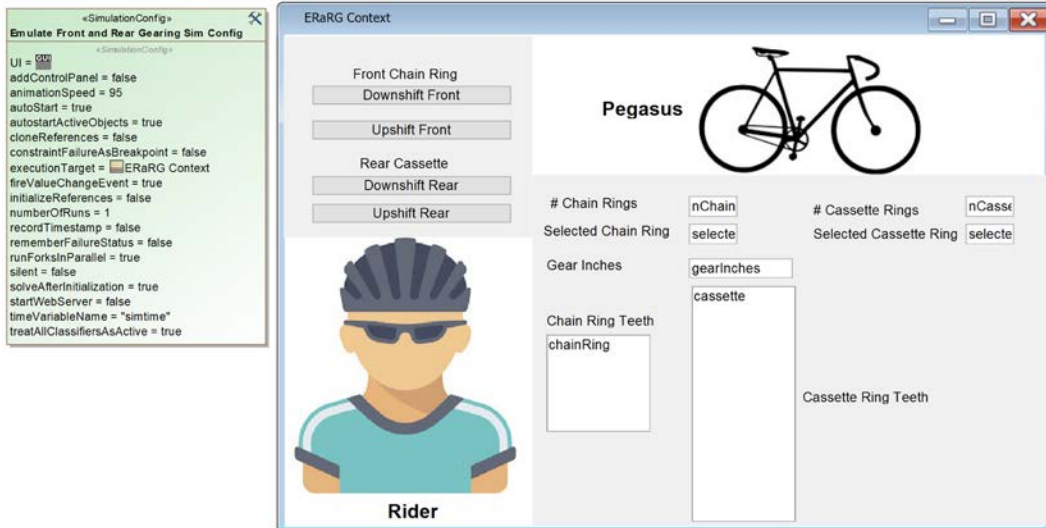


Figure 2.43: Emulate Front and Rear Gearing UI

Generate use case scenarios

Scenarios are specifically sequenced interaction sets that identify sequencing, timing, and values of different exemplar uses of a system. Sequence diagrams are generally easy to understand, even for non-technical stakeholders. In this recipe, sequences are created by exercising the use case state machine by changing the inputs to exercise different transition paths in the state machine. It is important to understand that there are usually an infinite set of possible scenarios, so we must constrain ourselves to consider a small representative set. The criteria we recommend is the **minimal-spanning set**; this is a set of scenarios such that each transition path and action is executed at least once. More scenarios of interest can be added, but the set of sequences should at least meet this basic criterion.

Let's consider a scenario showing the rider changing gears while riding (Figure 2.44). The figure skips over the initialization of the gearing and gets directly to the changing of the front chain ring and the rear cassette:

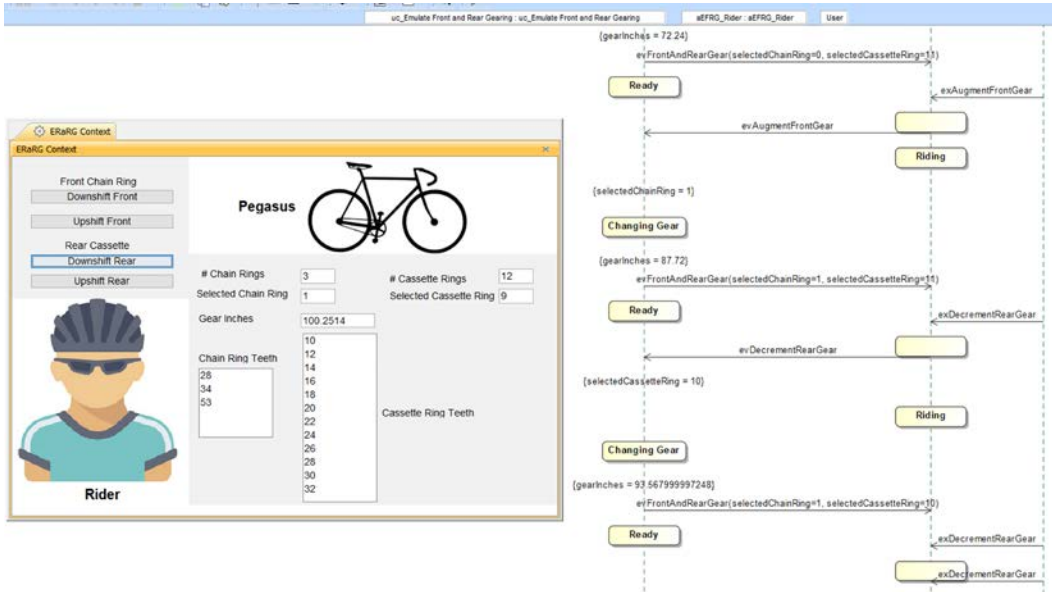


Figure 2.44: Scenario for gear changes while riding

Verify and validate requirements

The creation of the state machines in the previous section and their execution allows us to identify missing, incorrect, or incomplete requirements. The panel diagram in Figure 2.44 allows us to drive different scenarios and perform “what if” analyses to explore the requirements.

Requirements_change

Parallel to the development and execution of the use case model, we maintain the textual requirements. This workflow event indicates the need to fix an identified requirements defect.

Update requirement set

In this example, we'll show the requirements in a table in the modeling tool. *Figure 2.45* shows the newly added requirements:

#	△ Name	Text
1	12 efarg01	The system shall notify the rider of the current number of chain rings and cassette rings on start up.
2	13 efarg02	The system shall accept a rider command to enter a mode to configure the gearing.
3	14 efarg03	The system shall accept a rider command to set up from 1 to 3 front chain rings, inclusive.
4	15 efarg04	The default number of chain rings shall be 2.
5	16 efarg05	The rider shall be able to decrement the cassette ring from a higher (smaller number of teeth) to the next lower (larger number of teeth) gear until the largest cassette ring is reached.
6	17 efarg06	The system shall accept a rider command to set up from 10-12 cassette rings, inclusive.
7	18 efarg07	The default number of cassette rings shall be 12.
8	19 efarg08	The system shall accept a rider command to set any chain ring to have from 20 to 70 teeth.
9	20 efarg09	The system shall accept a rider command to set up any cassette ring to have from 10 to 50.
10	21 efarg10	The default number of teeth for 1 chain ring shall be 48.
11	22 efarg11	The default number of teeth for 2 chain rings shall be 34 and 53.
12	23 efarg12	The system shall inform the rider of the new gearing when the gear is changed.
13	24 efarg13	The default number of teeth for 3 chain rings shall be 28, 40, and 56.
14	25 efarg14	The system shall inform the training app of the new gearing when the gear is changed.
15	26 efarg15	The default number of teeth for 10 cassette rings shall be 11, 12, 14, 16, 18, 20, 22, 25, 28, and 32.
16	27 efarg16	The default number of teeth for 11 cassette rings shall be 11, 12, 13, 14, 15, 16, 17, 19, 21, 23, and 25.
17	28 efarg17	The default number of teeth for 12 cassette rings shall be 11, 13, 15, 17, 19, 21, 24, 28, 32, 36, 42, and 50.
18	29 efarg18	The rider shall be able to command the system to leave configuration mode.
19	30 efarg19	The default starting gear shall be chain ring 1 and cassette ring 1 when starting a ride.
20	31 efarg20	The rider shall be able to augment the front chain ring from a lower to the next higher gear until the largest chain ring is reached.
21	32 efarg21	The rider shall be able to decrement the front chain ring from a higher to the next lower gear until the smallest chain ring is reached.
22	33 efarg22	The rider shall be able to augment the cassette ring from a lower (larger number of teeth) to the next higher (smaller number of teeth) gear until the smallest cassette ring is reached.

Figure 2.45: Emulate Front and Rear Gearing requirements

Add trace links

Now that we've identified the requirements, we can add them to the model and add trace links to the Emulate Front and Rear Gearing use case. This is shown in the table in *Figure 2.46*:

UseCasePkg	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	
Control Resistance																							
Emulate Basic Gearing																							
Emulate DI Shifting																							
Emulate DI2 Button Gear Selection																							
Emulate Front and Rear Gearing	15	6	6	7	7	22																	

Figure 2.46: Emulate Front and Rear Gearing requirements trace matrix

Perform use case and requirements review

With the analysis complete and the requirements added, a review can be conducted to evaluate the set of requirements. This review typically includes various subject matter experts in addition to the project team.

Functional Analysis with User Stories

The other functional analysis recipes in this chapter are fairly rigorous and use executable models to identify missing and incorrect requirements. User stories can be used for simple use cases that don't have complex behaviors. In the other functional analysis recipes, validation of the use case requirements can use a combination of subject matter expert review, testing, and even formal mathematical analysis prior to their application to the system design. User stories only permit validation via review and so are correspondingly harder to verify as complete, accurate, and correct.

A little bit about user stories

User stories are approximately equivalent to scenarios in that both scenarios and user stories describe a singular path through a use case. Both are “partially constructive” in the sense that individually, they only describe part of the overall use case. User stories do it with natural language text, while scenarios do it with a SysML sequence diagram. The difference between user stories and scenarios is summarized in *Figure 2.47*:

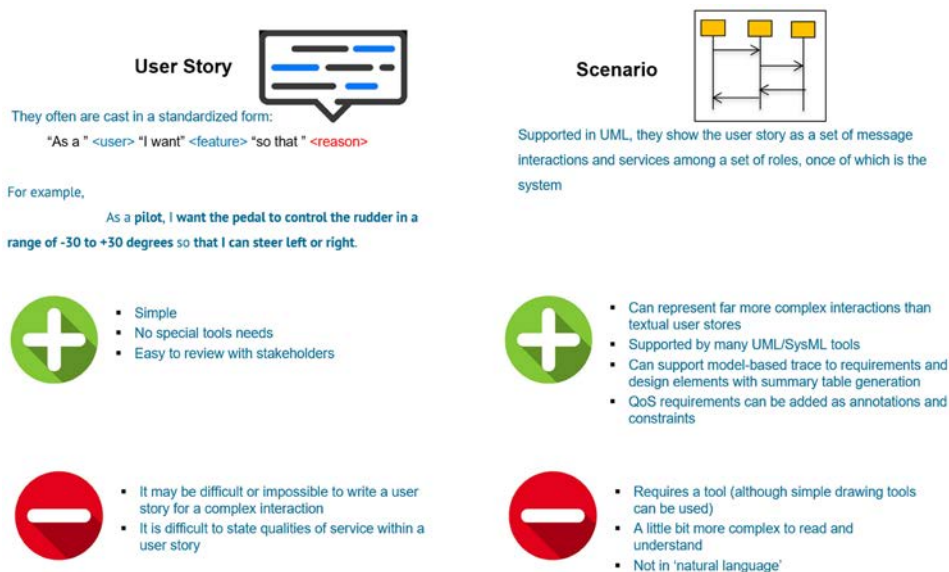


Figure 2.47: User story or scenarios

User stories have a canonical form:



“As a” <user> “I want” <feature> “so that” <reason>|<outcome>

A few examples of user stories are provided in *Chapter 1, Basics of Agile Systems Modeling*, in the recipe for *Estimating Effort*, such as:

User Story: **Set Resistance Under User Control:**



As a rider, I want to set the resistance level provided to the pedals to increase or decrease the effort for a given gearing, cadence, and incline so that the system simulates road riding effort.

Each user story represents a small set of requirements. A complete set of user stories includes all or most requirements traced to by the use case.

In SysML, we represent user stories as stereotypes of use cases and use the «include» relations to indicate the use case to which the user story applies. The stereotype adds the **acceptance_criteria** tag to the user story so that it is clear what it means to satisfy the user story. An example, relating a use case, user stories, and requirements, is shown in *Figure 2.48*:

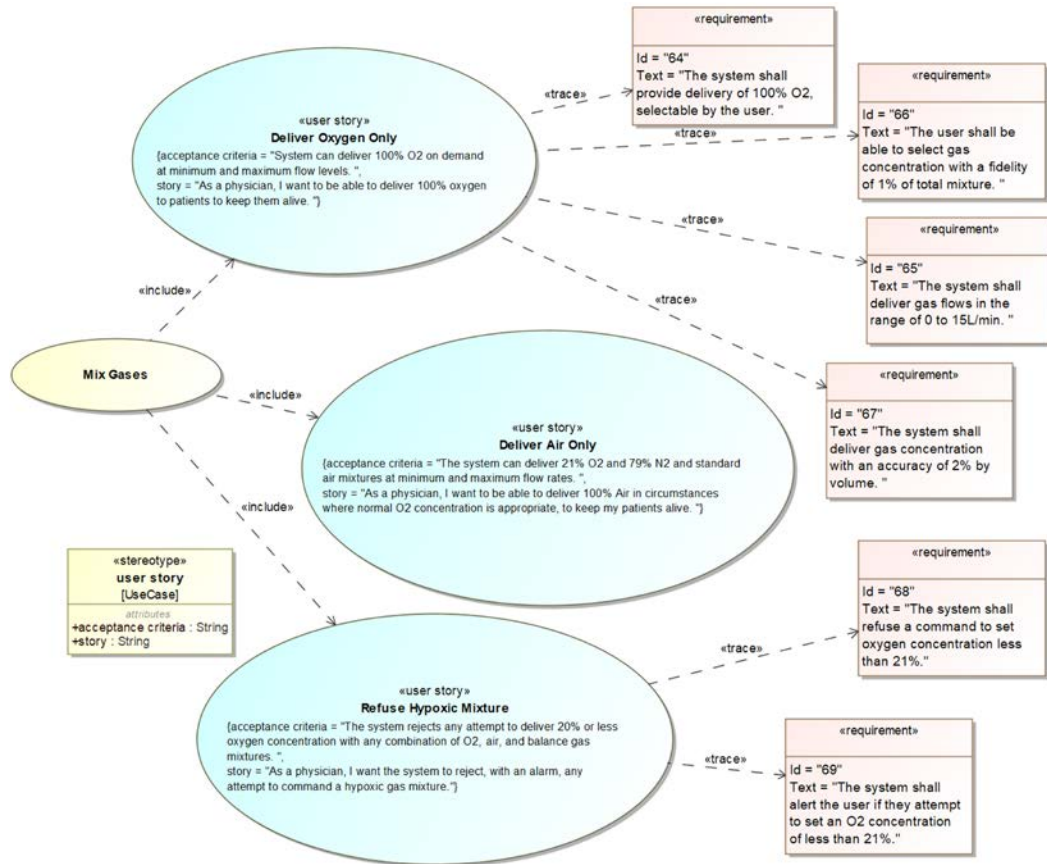


Figure 2.48: User Story as a stereotype of use case

User Story Guidelines

Here are some guidelines for developing good user stories:

1. Focus on the users.

Avoid discussing or referencing design, but instead, focus on the user-system interaction.

2. Use personae to discover the stories.

Most systems have many stakeholders with needs to be met. Each user story represents a single stakeholder role. Represent all the users with the set of user stories.

3. Develop user stories collaboratively.

User stories are a lightweight analytic technique and can foster good discussions among the product owner and stakeholders, resulting in the identification of specific requirements.

4. Keep the stories simple and precise.

Each story should be easy to understand; if it is complex, then try to break it up into multiple stories.

5. Start with epics or use cases.

User stories are small, finely-grained things, while epics and use cases provide a larger context.

6. Refine your stories.

As your understanding deepens and requirements are uncovered, the stories should be updated to reflect this deeper understanding.

7. Be sure to include acceptance criteria.

Acceptance criteria complete the narrative by providing a clear means by which the system design and implementation can be judged to appropriately satisfy the user's needs.

8. Stay within the scope of the owning epic or use case.

While it is true that in simple systems, user stories may not have an owner epic or use case, most will. When there is an owner epic or use case, the story must be a subset of that capability.

9. Cover all the stories.

The set of user stories should cover all variant interaction paths of the owning epic or use case.

10. Don't rely solely on user stories.

Because user stories are a natural language narrative, it isn't clear how they represent all the quality of service requirements. Be sure to include safety, reliability, security, performance, and precision requirements by tracing the user story to those requirements.

Purpose

User stories are a simple way to understand and organize requirements. Most commonly, these are stories within the larger capability context of an epic or use case. User stories are approximately equivalent to a scenario.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view.

Outputs and postconditions

The most important outcome is a set of requirements accurately and appropriately specifying the behavior of the system for the use case and acceptance criteria for what it means to satisfy them.

How to do it

Figure 2.49 shows the workflow for this recipe. It is a more informal approach than the preceding recipes but may be useful for simple use cases. Note that, unlike previous recipes, it does not include a behavioral specification in a formal language such as activities or state machines:

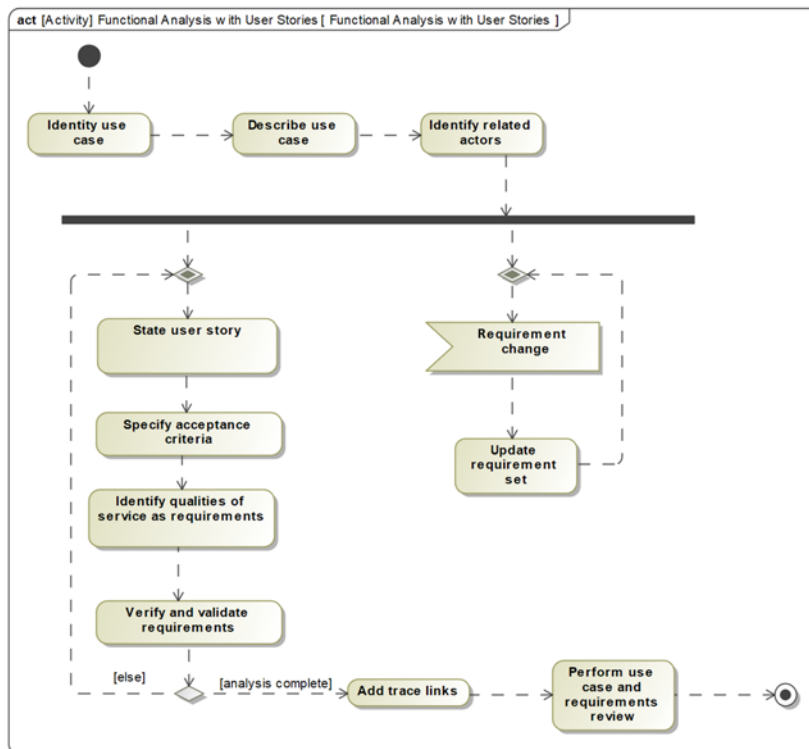


Figure 2.49: Functional analysis with user stories

Identify use case

This first step is to identify the generic usage of which the scenarios of interest, user stories, and requirements are aspects.

Describe use case

The description of the use case should include its purpose, and a general description of the flows, preconditions, postconditions, and invariants (assumptions). Some modelers add the specific actors involved, user stories, and scenarios, but I prefer to use the model itself to contain those relations.

Identify related actors

The related actors are those people or systems outside our scope that interact with the system while it executes the current use case. These actors can send messages to the system, receive messages from the system, or both.

State the user stories

This step includes more than creating the “As a <role> ...” statements. It also includes creating «include» relations from the owning use case and the addition of acceptance criteria for each user story. If this is the first time this is being done, you will also have to create a «user story» stereotype that applies to use cases to be able to create the model elements, as illustrated in *Figure 2.48*.

Specify acceptance criteria

Clearly state the criteria that will be applied to the system to demonstrate that the user story is acceptably implemented.

Identify quality of service as requirements

It is very common to forget to include various kinds of qualities of service. This step is an explicit reminder to specify *how well* the services are provided. Common qualities of service include safety, security, reliability, performance, precision, fidelity, and accuracy.

Verify and validate requirements

For this recipe, validating the requirements is done with a review with the relevant stakeholders. This should involve looking at the use, the set of user stories, the user stories themselves and their acceptance criteria, and the functional and quality of service requirements.

Requirements_change

Parallel to the development and execution of the use case model, we maintain the textual requirements. This workflow event indicates the need to fix an identified requirements defect.

Update requirement set

In response to an identified requirements defect, we fix the textual requirements by adding, deleting, or modifying requirements. This will then be reflected in the updated model.

Add trace links

Once the use case model and requirements stabilize, we add trace links using the «trace» relation or something similar. These relations allow the backtrace to stakeholder requirements as well as forward links to any architectural elements that might already exist. For relations forward to design elements, we generally prefer «satisfy».

Perform use case and requirements review

Once the work has stabilized, a review for correctness and compliance with standards may be done. This allows subject matter experts and stakeholders to review the requirements, use cases, and user stories for correctness, and for quality assurance staff to ensure compliance with modeling and requirements standards.

Example

Identify use case

For this recipe, we will analyze the **Emulate DI Shifting** use case. In many ways, this use case is an ideal candidate for user stories because the use case is simple and not overly burdened with quality-of-service requirements.



Interested readers can learn more about DI shifting here: https://en.wikipedia.org/wiki/Electronic_gear-shifting_system.

Describe use case

The use case description is shown in *Figure 2.50*:

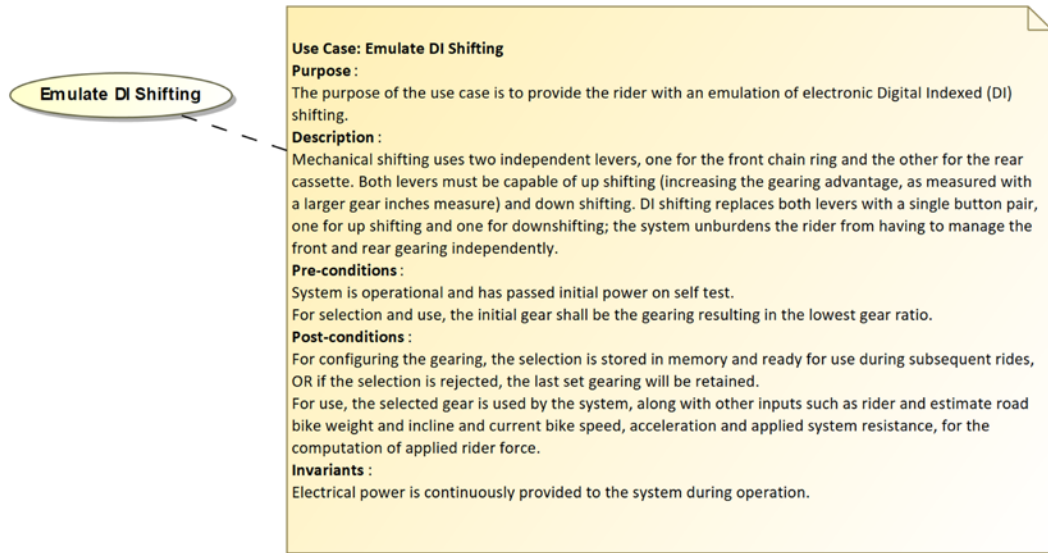


Figure 2.50: Description of the emulate DI shifting use case

Identify related actors

The only actor in this use case is the **Rider**, as shifting gears is one of the three key ways that the **Rider** interacts with the system (the other two being pedaling and applying brakes).

State the user stories

Figure 2.51 shows the three identified user stories for the use case: using buttons to shift gears, handling gearing crossover on upshifting, and handling gearing cross-over on downshifting:

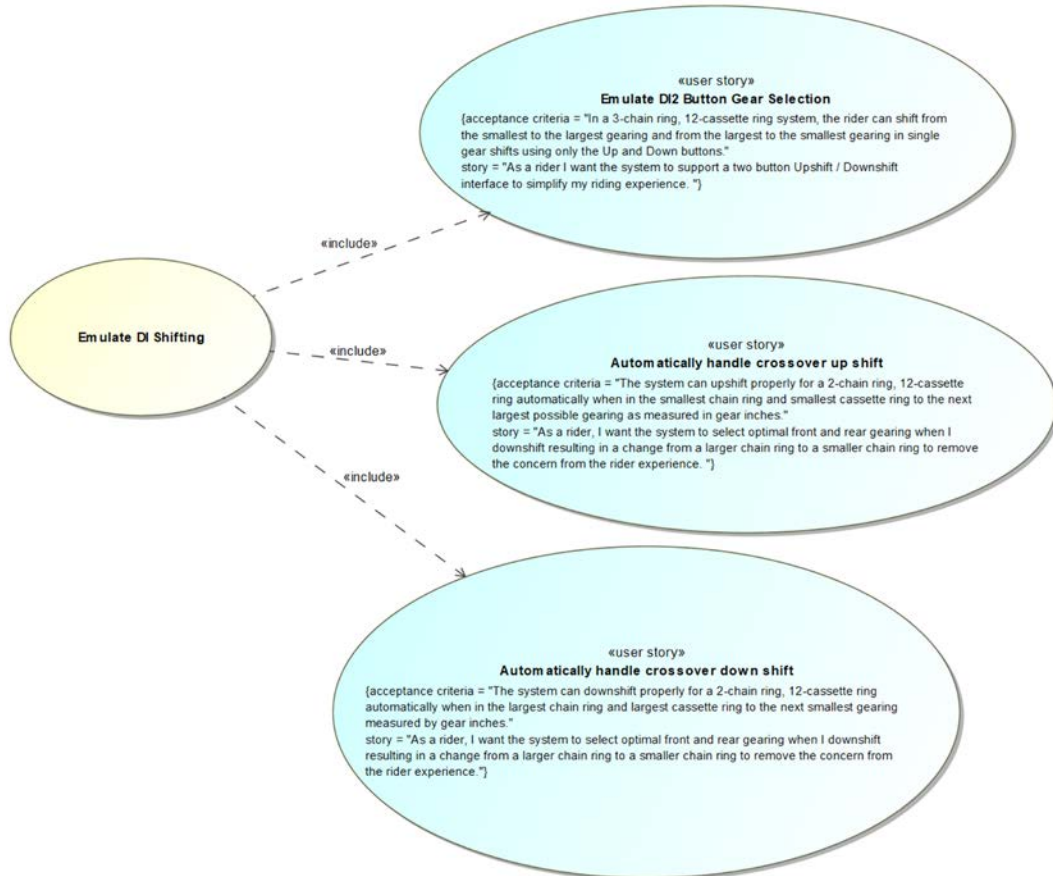


Figure 2.51: Emulate DI shifting user stories

Specify the related requirements

As these are simple user stories, there are a small number of functional requirements. See *Figure 2.52*:

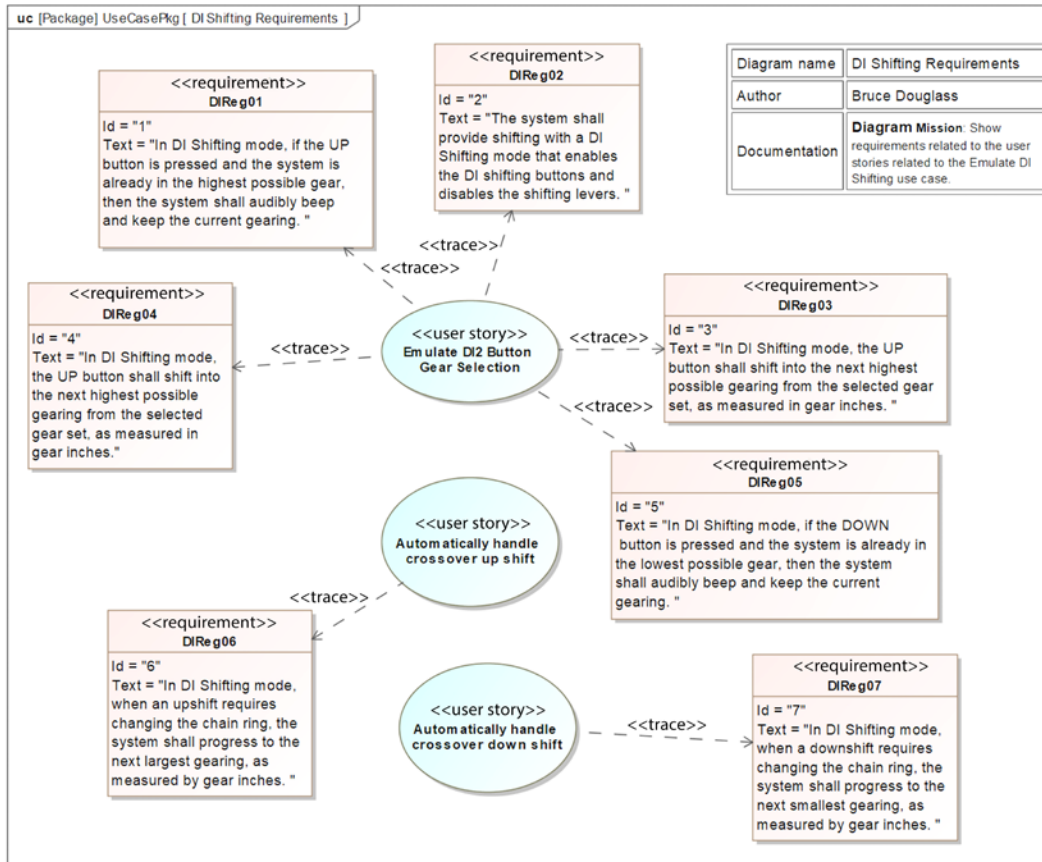


Figure 2.52: Emulate DI shifting functional requirements

Identify quality of service as requirements

The previous step specified a small number of requirements but didn't clarify how well these system functions are to be performed. Most notably, performance and reliability requirements are missing. These are added in *Figure 2.53*, shown this time in a requirements table:











#	Name	Text
1	 1 DIReg01	In DI Shifting mode, if the UP button is pressed and the system is already in the highest possible gear, then the system shall audibly beep and keep the current gearing.
2	 2 DIReg02	The system shall provide shifting with a DI Shifting mode that enables the DI shifting buttons and disables the shifting levers.
3	 3 DIReg03	In DI Shifting mode, the UP button shall shift into the next highest possible gearing from the selected gear set, as measured in gear inches.
4	 4 DIReg04	In DI Shifting mode, the UP button shall shift into the next highest possible gearing from the selected gear set, as measured in gear inches.
5	 5 DIReg05	In DI Shifting mode, if the DOWN button is pressed and the system is already in the lowest possible gear, then the system shall audibly beep and keep the current gearing.
6	 6 DIReg06	In DI Shifting mode, when an upshift requires changing the chain ring, the system shall progress to the next largest gearing, as measured by gear inches.
7	 7 DIReg07	In DI Shifting mode, when a downshift requires changing the chain ring, the system shall progress to the next smallest gearing, as measured by gear inches.
8	 34 DIQoSReq01	In DI Mode, shifts shall be executed in less than 400 ms or 1/4 pedal stroke, whichever is longer in time.
9	 35 DIQoSReq03	In DI Shifting mode, gear shifts may be taken regardless of current power load being applied to the gears, from 0W to the maximum load allowed.
10	 36 DIQoSReq02	The DI shifting button shall perform reliably for at least 100,000 presses.

Figure 2.53: Emulate DI Shifting QoS and Functional Requirements

Verify and validate requirements

The next step is to validate the user stories and related requirements with the stakeholders to ensure their correctness and look for missing, incorrect, or incomplete requirements.

Requirements_change

During this analysis, a stakeholder notes that nothing is said about how the system transitions between mechanical shifting and DI shifting. The following requirements are added:

- The system shall enter DI Shifting mode by selecting that option in the **Configuration App**.
- Once the DI Shifting mode is selected, this selection shall persist across resets, power resets, and software updates.
- Mechanical shifting shall be the default on initial startup or after a factory-settings reset.
- The system shall leave DI Shifting mode when the user selects the **Mechanical Shifting** option in the **Configuration App**.

Update requirements set

The requirements are updated to reflect the stakeholder input from above.

Add trace links

Trace links from both the use case and user stories to the requirements are added. These are shown in diagrammatic form in *Figure 2.54*.

Note: the figure does not show that use case **Emulate DI Shifting** traces to all these requirements just to simplify the diagram:

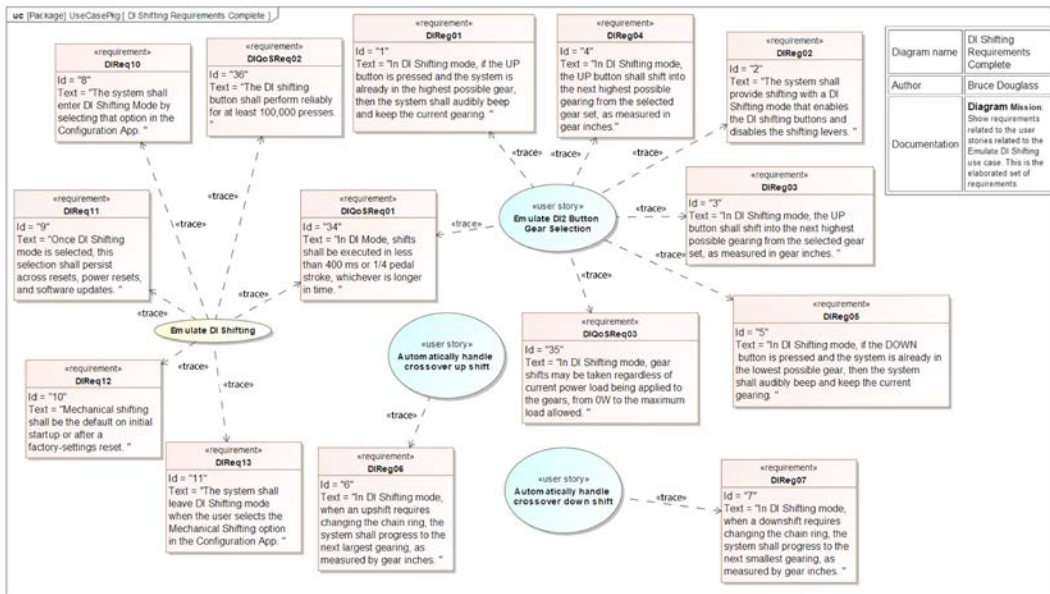


Figure 2.54: Emulate DI Shifting trace links

Perform use case, user story and requirements review

With the analysis complete and the requirements added, a review can be conducted to evaluate the set of requirements. This review typically includes various subject matter experts in addition to the project team.

Model-Based Safety Analysis

The term *safety* can be defined as *freedom from harm*. Safety is one of the three pillars of the more general concern of *system dependability*. Safety is generally considered with respect to the system causing or allowing physical harm to persons, up to and including death. Depending on the industry, different systems must conform to different safety standards, such as DO-178 (airborne software), ARP4761 (aerospace systems), IEC 61508 (electronic systems), ISO 26262 (automotive safety), IEC 63204 and IEC 60601 (medical), and EN50159 (railway), just to name a few. While there is some commonality among the standards, there are also a number of differences that you must take into account when developing systems to comply with those standards.

The recipe in this chapter provides a generic workflow applicable to all these standards but you may want to tailor it to your specific needs. Note that we recommend this analysis is done on a per-use case basis so that the analysis of each relevant use case includes safety requirements in addition to the functional and quality of services requirements.

A little bit about safety analysis

Some key terms for safety analysis are:

- **Accident** – A loss of some kind, such as injury, death, equipment damage, or financial. Also known as a *mishap* or *harm*.
- **Risk** – The product of the likelihood of an accident times its severity, so $R = L * S$.
- **Hazard** – A set of conditions and/or events that inevitably results in an accident.
- **Fault Tolerance Time** – The period of time a system can manifest a fault before an accident is likely to occur.
- **Safety control measure** – An action or mechanism that improves systems safety either by 1) reducing its likelihood or 2) reducing its severity.

The terms **faults**, **failures**, and **errors** are generally used in one of three ways, depending on the standard employed:

1. **Faults** lead to **Failures**, which lead to **Errors**:
 - a. **Fault** – An incorrect step, process, or data
 - b. **Failure** – The inability of a system or component to perform its required function
 - c. **Error** – A discrepancy between an actual value or action and the theoretically correct value or action
 - d. A fault at one level can lead to a failure one level up
2. **Faults** are actual behaviors that conflict with specified or desired behaviors:
 - a. **Fault** – Either a failure or an error
 - b. **Failure** – An event that occurs at a point in time when a system or component performs incorrectly:

Failures are *random* and may be characterized by a probability distribution
 - c. **Error** – A condition in which a system or component systematically fails to achieve its required function:

Errors are *systematic* and always exist, even if they are not manifest

Errors are the result of requirement, design, implementation, or deployment mistakes, such as a software bug

- d. **Manifest** – When a fault is visible. Faults may be *manifest* or *latent*
3. Faults are undesirable anomalies in systems or software (ARP-4761):
 - a. **Failure** – A loss of function or a malfunction of a system
 - b. **Error** –
 - An occurrence arising as a result of an incorrect action or decision by personnel operating or maintaining a system, or
 - A mistake in specification, design, or implementation

The most common form to perform the analysis is with a **Fault Tree Analysis (FTA)** diagram. This is a causality diagram that relates normal conditions and events, and abnormal conditions and events (such as faults and failures) with undesirable conditions (hazards). A **Hazard Analysis** is generally a summary of safety analysis from one or more FTAs.

An FTA diagram connects nodes with logic flows to aid understanding of the interactions of elements relevant to the safety concept. Nodes are either events, conditions, outcomes, or logical operators as shown in *Figure 2.55*. See <https://www.sae.org/standards/content/arp4761/> for a good discussion of FTA diagrams.

Some Profiles

Cameo provides the **Cameo Safety and Risk Analyzer** that supports table-based representations of information, mostly relying on **Failure Means and Effect (FMEA)** tables. Although this profile does not support diagrammatic representations such as FTAs, it is a viable approach for doing safety analysis. An introduction to this profile can be seen on the *No Magic* YouTube channel at <https://www.youtube.com/watch?v=NwuTV5-HA5s>. See also *Fault Tree Analysis and Simulation* at <https://www.youtube.com/watch?v=A86q00kF8Eo>.

I have long used the *Dependability Profile* I developed over a decade ago that supports FTA diagrams and safety analysis, along with support for reliability and security analysis. Recently, however, the **Risk Analysis and Assessment Modeling Language (RAAML)** has been released by OMG (see <https://www.omg.org/spec/RAAML/>). There is a prototype implementation of the RAAML standard available for Cameo at the time of this writing.



The Cameo prototype is available at <https://www.dropbox.com/s/g20itfxpdnyolzg/FTA%20RAAML%20Sample.mdzip?dl=0>. A Rhapsody RAAML prototype can be downloaded at <https://jazz.net/blog/index.php/2021/10/20/model-based-risk-analysis-with-raaml-in-ibm-engineering-systems-design-rhapsody/>.

Much of this Cameo profile is based on the internals of my Dependability Profile as well as RAAML, and I am completely in favor of the reuse of my profile in this context. RAAML is more expansive than the Dependability Profile in its safety analysis in that it supports some other analytic approaches; the two are similar in their support for reliability analysis, but the Cameo profile lacks any explicit security analysis. I hear from the team working on the RAAML standard that they are looking at a 1.1 revision to the RAAML standard that will address security analysis in the future. I will provide a simple profile for security analysis based on the Dependability Profile for the next recipe, *Model-Based Threat Analysis*.

I will use the Cameo prototype implementation in this book. Interested readers can download my Dependability Profile from my website at <https://www.bruce-douglass.com/models>, but be aware that it is only available for the Rhapsody tool.

Figure 2.55 shows the symbology in the Cameo RAAML prototype. It is slightly less complete than the Dependability Profile but is a serviceable set of elements.

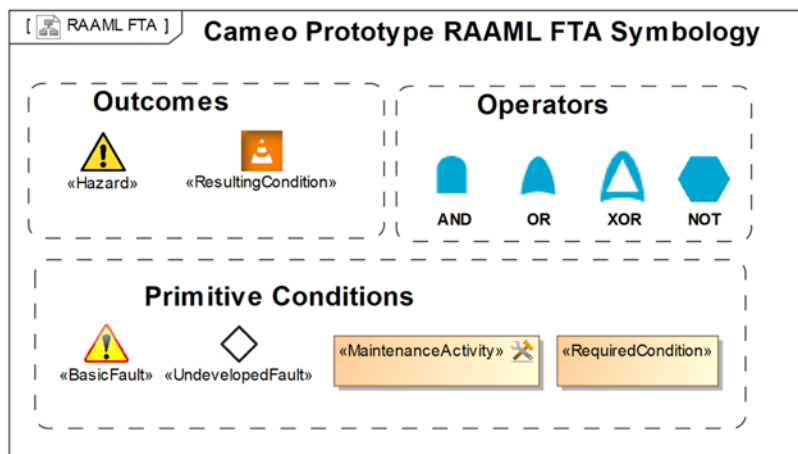


Figure 2.55: FTA symbology

At its core, an FTA is a causality diagram with conditions connected via logical operators showing a chain of causality. The logical operators take one or more inputs and produce a singular output. The AND operator, for example, produces a **TRUE** output if both its inputs are **TRUE**, while the OR operator returns **TRUE** if either of its inputs is **TRUE**.

Figure 2.56 shows an example FTA diagram. This diagram shows the safety concerns around an automotive braking system. The hazard under consideration is **Failure to Brake**. The diagram shows that this happens when the driver intends to brake and at least one of three conditions is present: a pedal input fault, an internal fault, or a wheel assembly fault:

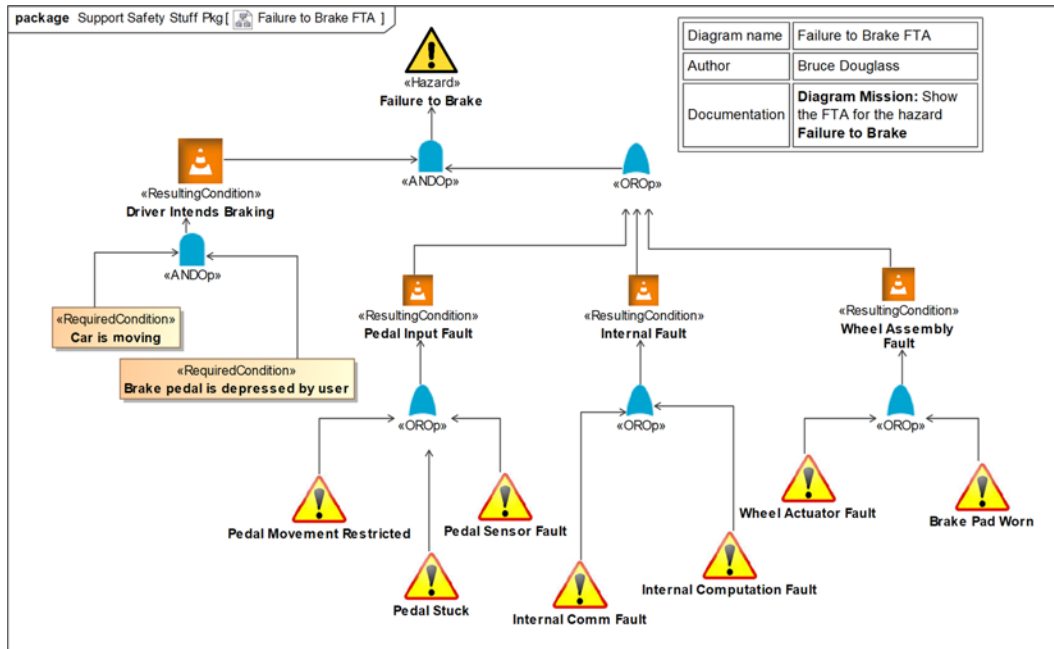


Figure 2.56: Example FTA diagram

Cut sets

A *cut* is a collection of faults that, taken together, can lead to a hazard. A *cut set* is the set of such collections such that all possible paths from the primitive conditions and events to the hazard have been accounted for. In general, if you consider n primitive conditions as binary (present or non-present), then there are 2^n cuts that must be examined. Consider the simple FTA in Figure 2.57. The primitive conditions are marked as a through e:

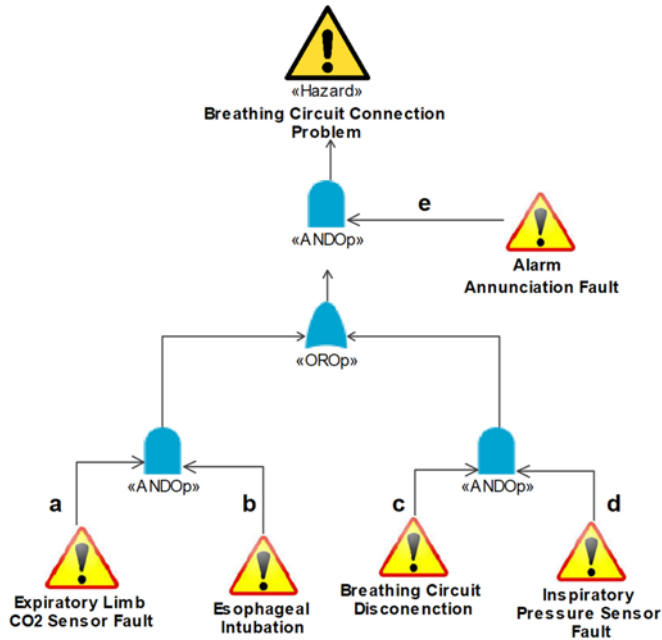


Figure 2.57: Cut set example

With 5 primitive conditions, there are 32 prospective cut sets that should be considered, of which only 3 can lead to the hazard manifestation, as shown in Figure 2.58. Only these three need to be subject to the addition of a safety measure:

Basic Fault/Condition	a	b	c	d	e	Hazard
1	T	T	F	F	T	T
2	F	F	T	T	T	T
3	T	T	T	T	T	T
4	T	F	F	F	T	F
5	F	T	F	F	T	F
6	F	F	T	F	T	F
7	F	F	F	F	T	F
8	T	T	T	T	F	F
9	F	T	T	T	F	F
10	F	F	T	T	F	F
(22 more...)						

Figure 2.58: Cut sets example (2)

Hazard analysis

There is normally one FTA diagram per identified hazard, although that FTA can be decomposed into multiple FTA diagrams (although the Cameo profile lacks the transfer operator from the FTA standard, there are other means to accomplish this decomposition). A system, however, normally has multiple hazards. These are summarized into a hazard analysis. A hazard analysis summarizes the hazard-relevant metadata, including the hazard name, description, severity, likelihood, risk, tolerance time, and possibly, related safety-relevant requirements, and design elements.

It should be noted that safety analysis is a rich and deep topic, the details of which are beyond the scope of this book. In this recipe, we will provide a simple FTA-based approach for performing safety analysis.

Purpose

The purpose of this recipe is to create a set of safety-relevant requirements for the system under development by analyzing safety needs.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view that has been identified, described, and for which relevant actors have been identified. Note: this recipe is normally performed in parallel to one of the functional analysis recipes from earlier in this chapter.

Outputs and postconditions

The most important outcome is a set of requirements specifying how the system will mitigate or manage the safety concerns of the system. Additionally, a safety concept is developed identifying the need for a set of safety control measures, which is summarized in a hazard analysis.

How to do it

Figure 2.59 shows the workflow for the recipe:

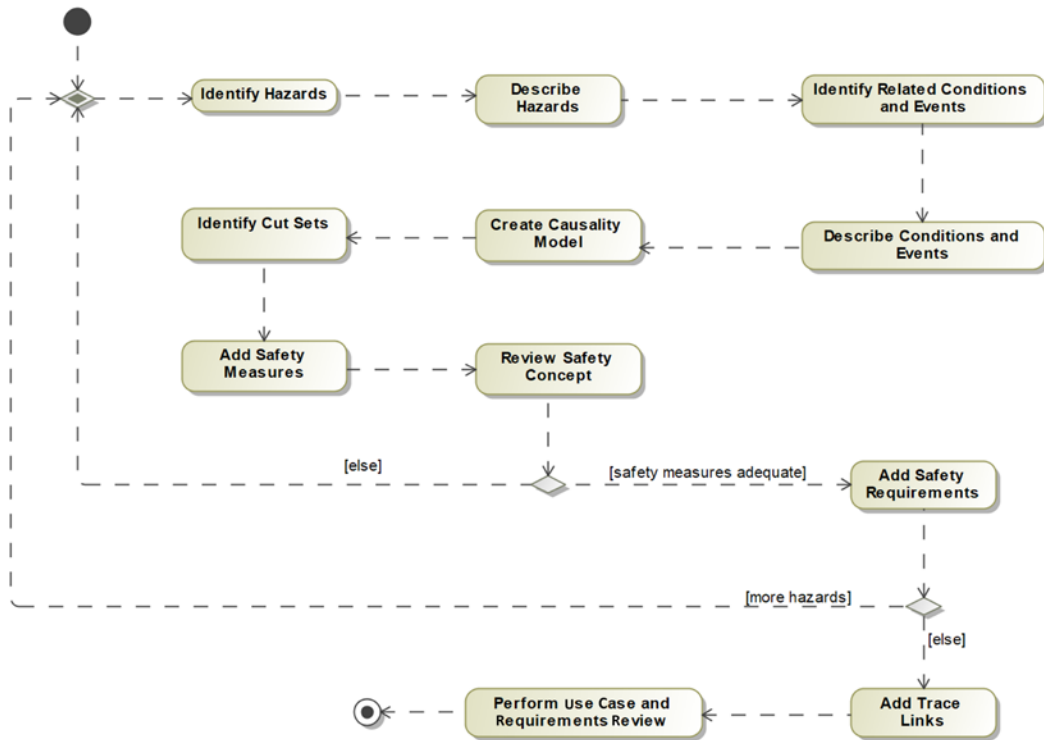


Figure 2.59: Model-based safety analysis workflow

Identify hazards

A hazard is a condition that leads to an accident or loss. This step identifies the hazards relevant to the use case under consideration that could arise from the system behavior in its operational context.

Describe hazards

Hazards are specified by their safety-relevant metadata. This generally includes the hazard name, description, likelihood, severity, risk, and safety integrity level, adopted from the relevant safety standard.

Identify related conditions and events

This step identified the conditions and events related to the hazard, including:

- Required conditions
- Normal events

- Hazardous events
- Fault conditions
- Resulting conditions

Describe conditions and events

Each condition and event should be described. A typical set of aspects of such a description includes:

- Overview
- Effect
- Cause
- Current controls
- Detection mechanisms
- Failure mode
- Likelihood, or Mean Time Between Failure (MTBF)
- Severity
- Recommended action
- Risk priority (product of likelihood and severity or MTBF/severity)

Create causality model

This step constructs an FTA connecting the various nodes with logic flows and logic operators flowing from primitive conditions up to resulting conditions and, ultimately, to the hazard.

Identify cut sets

Identify the relevant cuts from all possible cut sets to ensure that each is “safe enough” to meet the safety standard being employed. This typically requires the addition of safety measures, as discussed in the next step.

Add safety measures

Safety measures are technical means or usage procedures by which safety concerns are mitigated. All safety measures reduce either the likelihood or the severity of an accident. In this analysis, care should be taken to specify the effect of the measures rather than their implementation, as much as possible. Design-level hazard analysis will be conducted later to ensure the adequacy of the design realization of the safety measures specified here.

Review safety concept

This step reviews the analysis and the set of safety measures to ensure their adequacy.

Add safety requirements

The safety requirements specify needs that the design, context, or usage must meet in order to make the system adequately safe. These requirements may be specially annotated to indicate their safety relevance or may just be treated as requirements that the system must satisfy.

Add trace links

Once the use case model and requirements stabilize, we add trace links using the «trace» relation or something similar. It is especially important to trace the safety measures to the elements of the FTAs that identify the need for those requirements.

Perform use case and requirements review

Once the work has stabilized, a review for correctness and compliance with standards may be done. This allows subject matter experts and stakeholders to review the requirements, use cases, and user stories in light of the safety concerns.

Example

The Pegasus example problem isn't optimal to show safety analysis because it isn't a safety-critical system. For that reason, we will use a different example to illustrate the use of this recipe.

Problem statement – medical gas mixer

The **Medical Gas Mixer (MGM)** takes in gas from wall supplies for O₂, He, N₂, and air and mixes them, and delivers a flow to a medical ventilator. When operational, the flow must be in the range of 100 ml/min to 1500 ml/min with a delivered O₂ percentage (known as **Fraction of Inspired Oxygen**, or **FiO₂**) of no less than 21%. The flow from individual gas sources is selected by the physician via the **Ventilator** interface.

Neonates face an additional hazard of *hyperoxia* – too much oxygen in the blood – as this can damage their retina and lungs.

In this example, the focus of our analysis is the use case **Mix Gases**.

Identify hazards

The fundamental hazard of this system is *hypoxia* – delivering too little oxygen to sustain health.

The average adult breathes about 7-8 liters of air per minute, resulting in a delivered oxygen flow of around 1450 ml O₂/minute. For neonates, the required flow can be as low as 40 ml O₂/minute, while for large adults, the need might be as high as 4000 ml O₂/minute at rest.

Describe hazards

The «Hazard» stereotype includes a set of tags for capturing the hazard metadata. This is shown in *Figure 2.60*. An explanation of the tags of this stereotype is in order:

- **Probability** – This is the likelihood of occurrence of the item, event, or condition.
- **ftt** – **Fault Tolerance Time** – How long a fault can be tolerated before a mishap is likely to occur.
- **fttu** – **Fault Tolerance Time Units** – The time units for the fault tolerance time.
- **Severity** – The negative impact of the mishap. Different standards specify this in various ways.
- **Risk** – The product of the probability and the severity of the item.
- **SIL** – **Safety Integrity Level** – This is a safety-standard-specific risk categorization:

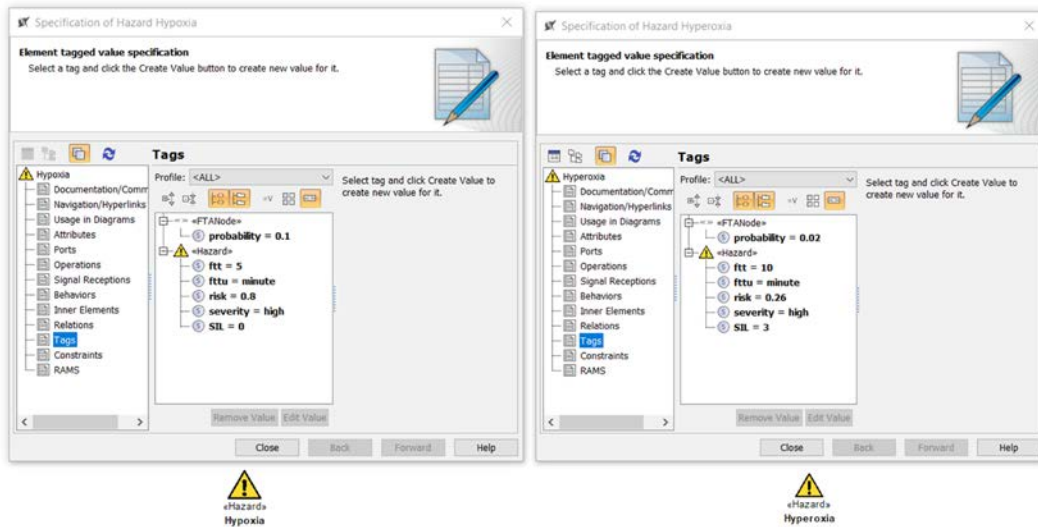


Figure 2.60: Mix Gases Hazards

Identify related conditions and events

For the rest of this example, we will focus exclusively on the **Hypoxia** hazard. There are two required conditions (or assumptions/invariants). First, that the gas mixer is in operation, and second, that there is a physician in attendance. This latter assumption means that the physician can be part of the “safety loop.”

There are a number of faults that are relevant to the **Hypoxia** hazard:

- The gas supply runs out of either air or O₂, depending on which is selected
- The gas supply valve fails for either air or O₂, depending on which is selected
- The patient is improperly intubated
- Faults in the breathing circuit, such as disconnected hoses or leak
- The ventilator commands a FiO₂ that is too low
- The ventilator commands a total flow of a specified mixture that is too low

Describe conditions and events

The «BasicFault» stereotype provides tags to hold fault metadata. The metadata for three of these faults, **Gas Supply Valve Fault**, **Improper Intubation**, and **Commanded FiO₂ Too Low**, are shown in *Figure 2.61*. Since the latter has more primitive underlying causes, it will be changed to a **Resulting Condition** and the primitive faults added:

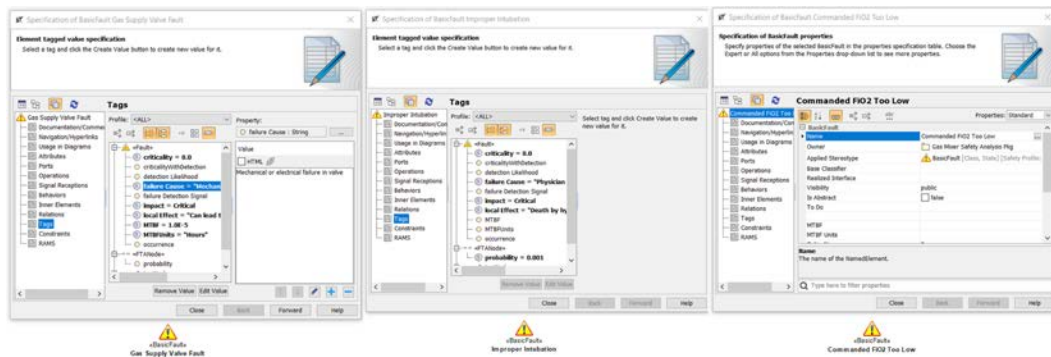


Figure 2.61: Fault Metadata

Create causality model

Figure 2.62 shows the initial FTA. This FTA doesn't include any safety mechanisms, which will be added shortly. Nevertheless, this FTA shows a causality tree linking the faults to the hazard with a combination of logic operators and logic flows:

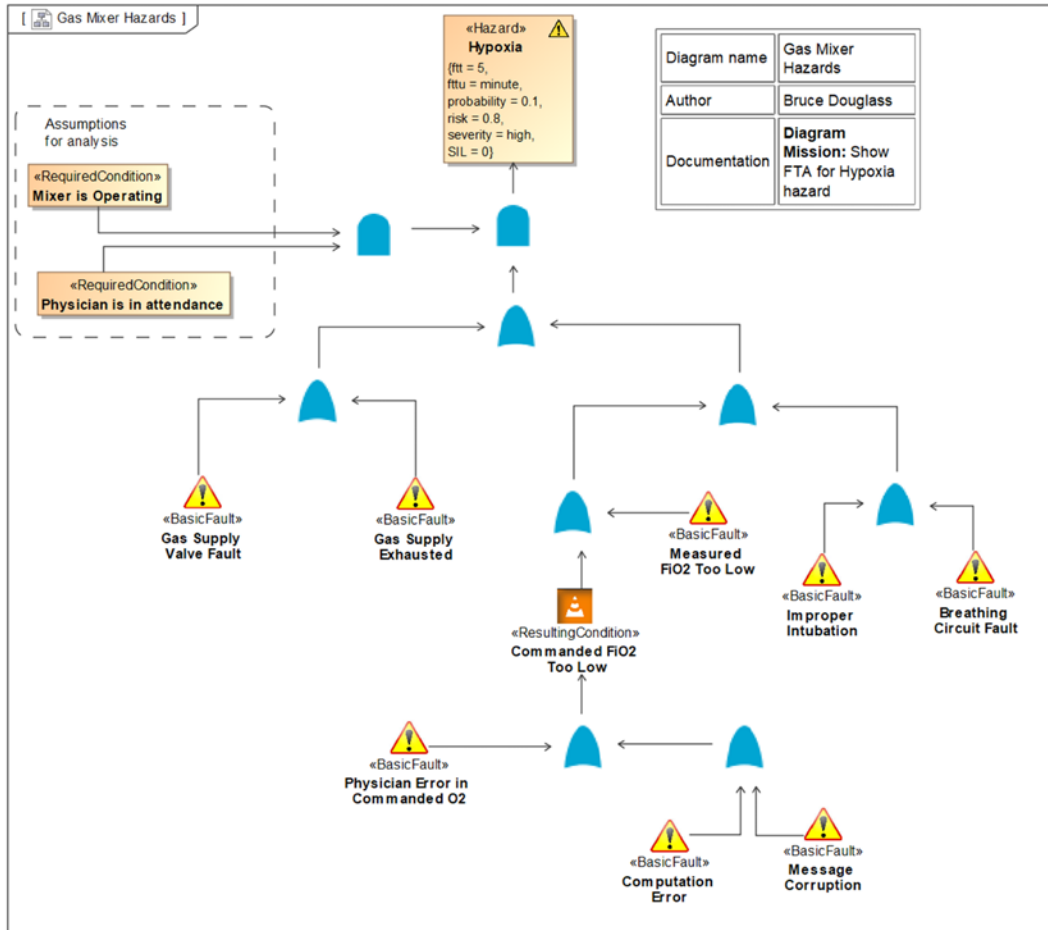


Figure 2.62: Initial FTA

Identify cut sets

There are 10 primitive fault elements, so there are potentially 2^{10} (1024) cuts in the cut set, although we are only considering cases in which the assumptions are true so that immediately reduces the set to 2^8 (256) possibilities. All of these are ORed together so it is enough to independently examine just the 8 basic faults.

Add safety measures

Adding a safety measure reduces either the likelihood or the severity of the outcome of a fault to an acceptable level. This is done on the FTA by creating *anding-redundancy*. This means that for the fault to have its original effort, both the original fault must occur *and* the safety measure must fail. The likelihood of both failing is the product of their probabilities. For example, if the **Gas Supply Valve Fault** has a probability of 8×10^{-5} , and we add a safety measure of a gas supply backup that automatically kicks in that has a probability of failure of 2×10^{-6} , then the resulting probability of both failing is 16×10^{-11} . Acceptable probabilities of hazards can be determined from the safety standard being used.

For the identified faults, we will add the following safety measures:

- **Gas Supply Valve Fault** safety measure: **Secondary Gas Supply**
- **Gas Supply Exhausted** fault safety measure: **Secondary Gas Supply**
- **Improper Intubation** fault safety measures: **CO2 Sensor on Expiratory Flow and Alarm on Fault**
- **Breathing Circuit Fault** safety measures: **Inspiratory Limb Flow Sensor and Alarm on Fault**
- **Physician Error in Commanded O2** safety measures: **Range Check Commanded O2 and Alarm on Fault**
- **Computation Error** fault safety measures: **Secondary Parallel Computation and Alarm on Fault**
- **Message Corruption** fault safety measure: **Message CRC**
- **Commanded Flow Too Low** fault safety measures: **Inspiratory Limb Flow Sensor and Alarm on Fault**

Adding these results in a more detailed FTA

To ensure readability, the inputs to the resulting condition are shown in another diagram for **FiO2 Too Low** (Figure 2.64). Figure 2.63 shows the high-level FTA diagram with safety measures added. Note that they are added in terms of what happens when they fail. Failure of safety measures is indicated with a **red bold font** for emphasis:

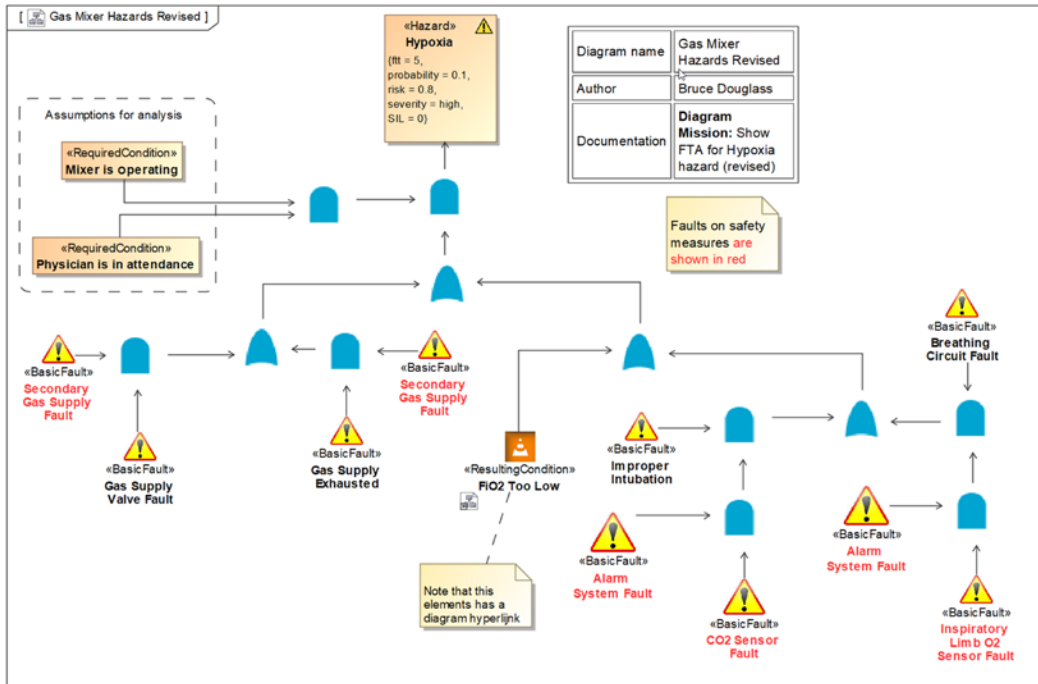


Figure 2.63: Elaborated FTA diagram

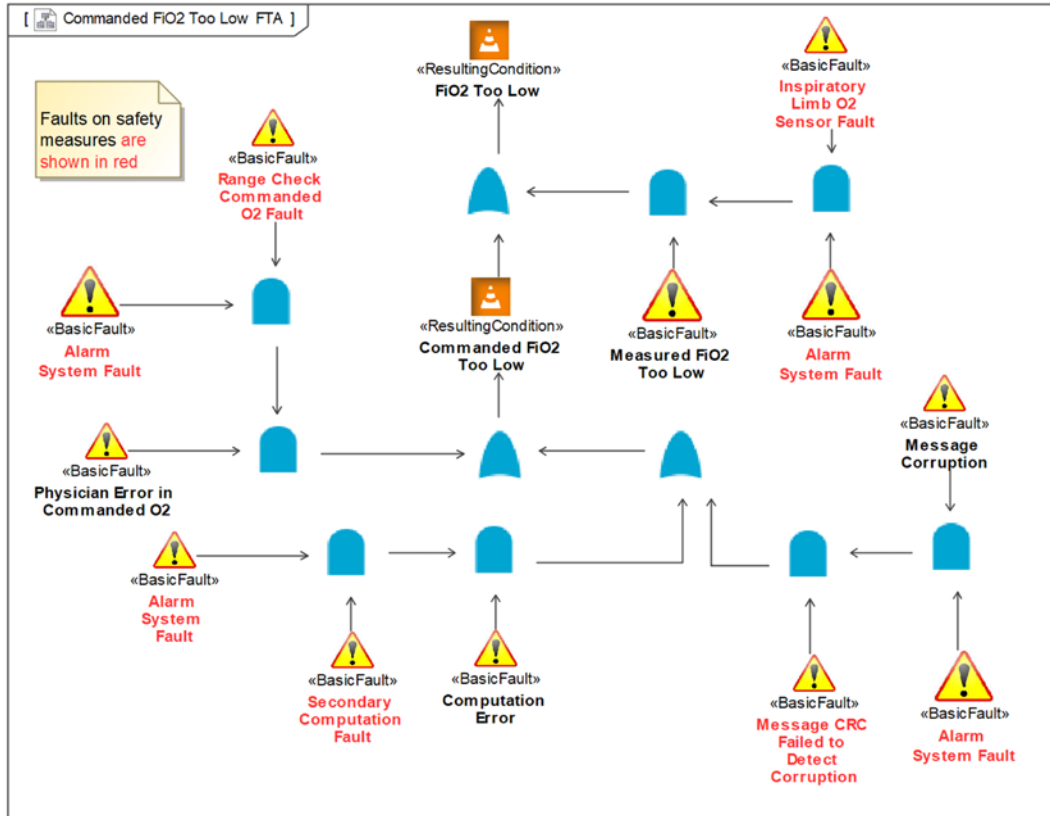


Figure 2.64: FIO2 flow too low sub-FTA

Review safety concept

The set of safety measures addresses all the identified safety concerns.

Add safety requirements

Now that we have identified the safety measures necessary to develop a safe system, we must create the requirements that mandate their inclusion. These are shown in *Figure 2.65*:

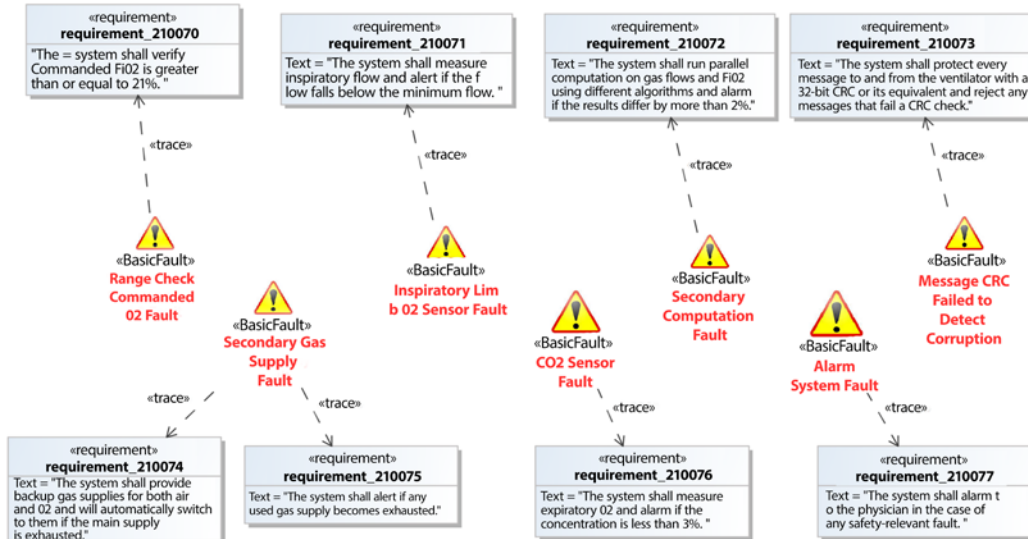


Figure 2.65: Safety requirements

Model-Based Threat Analysis

It used to be that most systems were isolated and disconnected; the only way to attack such a system required physical presence. Those days are long gone.

These days, most systems are internet-enabled and connected via apps to cloud-based servers and social media. This presents opportunities to attack these systems, compromise their security, violate their privacy, steal their information, and cause damage through malicious software.

Unfortunately, little has been done to protect systems in a systematic fashion. The most common response I hear when consulting is "Security. Yeah, I need me some of that," and the issue is ignored thereafter. Sometimes, some thought is given to applying security tests *ex post facto*, or perhaps doing some code scans for software vulnerabilities, but very little is done to methodically analyze a system from a cyber-physical security posture standpoint. This recipe addresses that specific need.

Basics of Cyber-Physical Security

Security is the second pillar of dependability. The first, safety, was discussed in the previous recipe. Key concepts for a systematic approach to cyber-security needs are:

Security – resilience to attack

Asset – a security-relevant feature of a system that the system has a responsibility to protect.

Assets have the following properties:

- Access Level Permitted
- Accountability
- Asset Kind
 - Actor
 - Information Asset
 - Currency Asset
 - Resource Asset
 - Physical Asset
 - Service Asset
 - Security Asset
 - Tangible Asset
 - Intangible Asset
- Availability
- Clearance Required
- ID
- Integrity
- Value

Asset context – The system or extra-system elements enshrouding one or more assets; e.g., a safe in which money is kept is a simple example of an asset context. An asset context may be decomposed into contained asset context elements.

Security field – The set of assets, asset contexts, vulnerabilities, and countermeasures for a system (also known as the system security posture).

Vulnerability – A weakness in the security field of an asset that may be exploited by an attack.

Threat – A means by which a vulnerability of the security field of an asset may be exploited.

Attack – The realization of a threat invoked by a threat agent.

Attack chain – A type of attack that is composed of sub-attacks. Sometimes known as a **Cyber Killchain**. Most modern attacks are of this type.

Threat agent – A human or automated threat source that invokes an attack, typically intentionally.

Security countermeasure – A means by which a vulnerability is protected from attack. Countermeasures may be passive or active and may be implemented by design elements, policies, procedures, labeling, training, or obviation. Countermeasure types include (but are not limited to):

- Access Control
- Accounting
- Active Detection
- Authentication
- Recovery
- Boundary Control
- Backup
- Encryption
- Deterrence
- Obviation
- Nonrepudiation
- Policy Action
- Response
- Scanning Detection

Role – A part a person plays in a context, e.g., a user, administrator, or trusted advisor.

Authenticated role – A role with explicit authentication, which typically includes a set of permissions.

Permission – The right or ability to perform an action that deals with an asset. A role may be granted permission to perform different kinds of access to an asset.

Access – A type of action that can be performed on a resource. This includes (and this list may be extended):

- No access
- Unrestricted access
- Read access
- Modify access
- Open access
- Close access
- Entry access
- Exit access
- Create access
- Delete access
- Remove access
- Invoke access
- Configure access
- Interrupt access
- Stop access

Security violation – The undesired intrusion, interference, or theft of an asset; this may be the result of an attack (intentional) or failure (unintentional).

Risk – The possibility of an undesirable event occurring or an undesirable situation manifesting. Risk is the product of (at least) two values: likelihood and severity. Severity in this case is a measure of the asset value.

Risk number – The numeric value associated with risk (likelihood x severity).

Modeling for Security Analysis

The UML Dependability Profile used in the previous recipe also includes cyber-physical threat modeling using the above concepts. The security information can be captured and visualized in a number of diagrammatic and tabular views. It may be downloaded at <https://www.bruce-douglass.com/safety-analysis-and-design>; however, it is profile-specific for the Rhapsody tool. For the purpose of this recipe, I created a simplistic security analysis profile for Cameo. The stereotypes used in this recipe are shown in *Figure 2.66*.

Icons I commonly use to represent those elements are shown next to the relevant stereotypes:

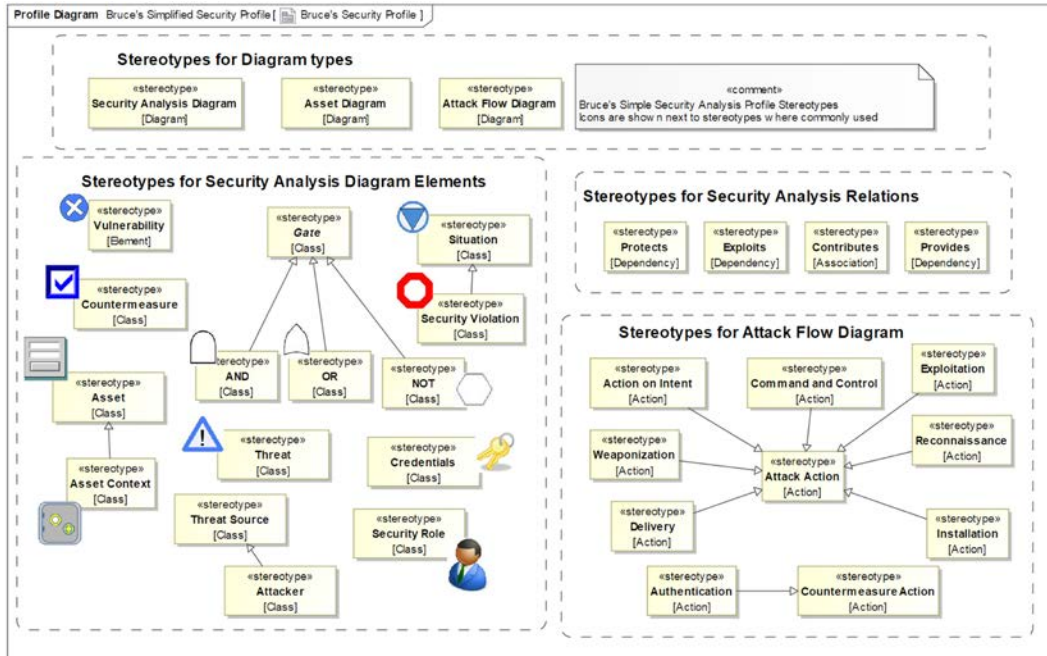


Figure 2.66: Bruce's simplified security analysis profile stereotypes

Some of the stereotypes have tagged values to hold security-relevant metadata. Figure 2.67 shows two stereotypes that have tags – **Asset** and **Asset Context** – and some value types used to define those tags:

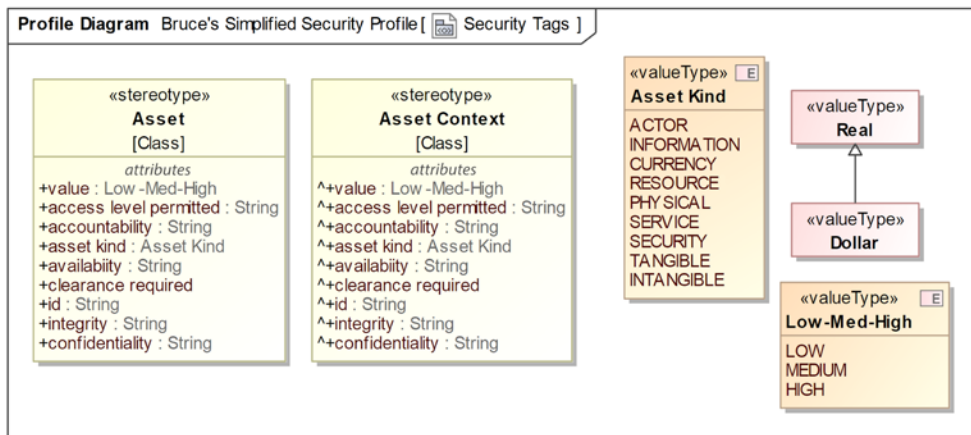


Figure 2.67: Security profile stereotype tags and types

Security analysis diagram

The **Security Analysis Diagram (SAD)** is a logical causality diagram very similar to the Fault Tree Analysis diagram used in the previous recipe. A SAD shows how assets, events, and conditions combine to express vulnerabilities, how countermeasures address vulnerabilities, and how attacks cause security violations. The intention is to identify when and where countermeasures are or should be added to improve system security. This diagram uses logical operations (**AND**, **OR**, **NOT**, **XOR**, and so on) to combine the presence of assets, asset context, situations, and events. *Figure 2.68* shows a typical SAD. You can identify the kind of element by the stereotype, such as **asset**, **asset context**, **countermeasure**, **vulnerability**, and **threat**:

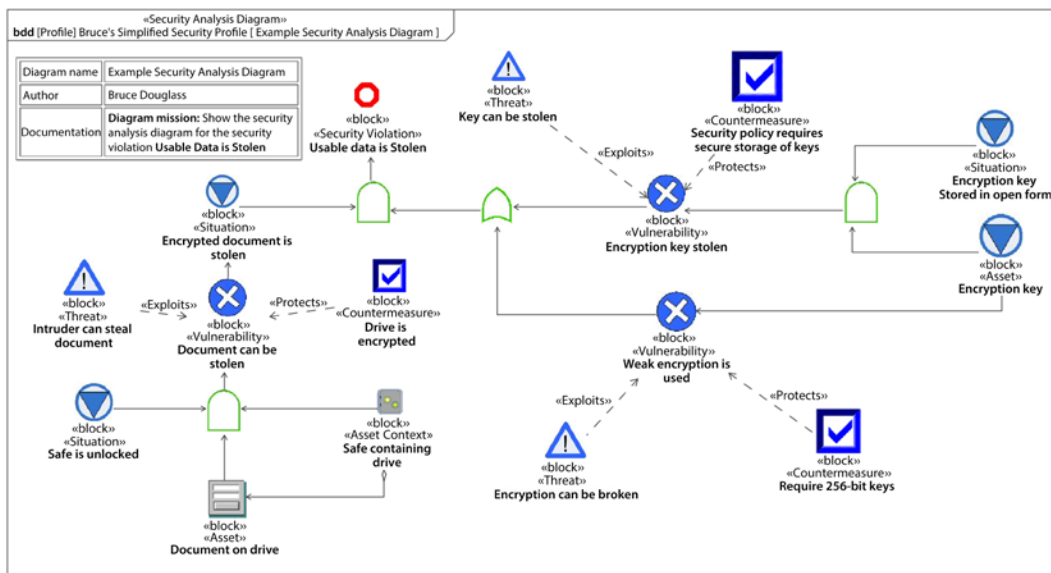


Figure 2.68: Security analysis diagram example

Asset diagram

Another useful diagram is the asset diagram. The asset diagram is meant to show the relationships among assets, asset contexts, vulnerabilities, countermeasures, supporting security requirements, and other security-relevant elements.

Figure 2.69 shows an asset diagram in use:

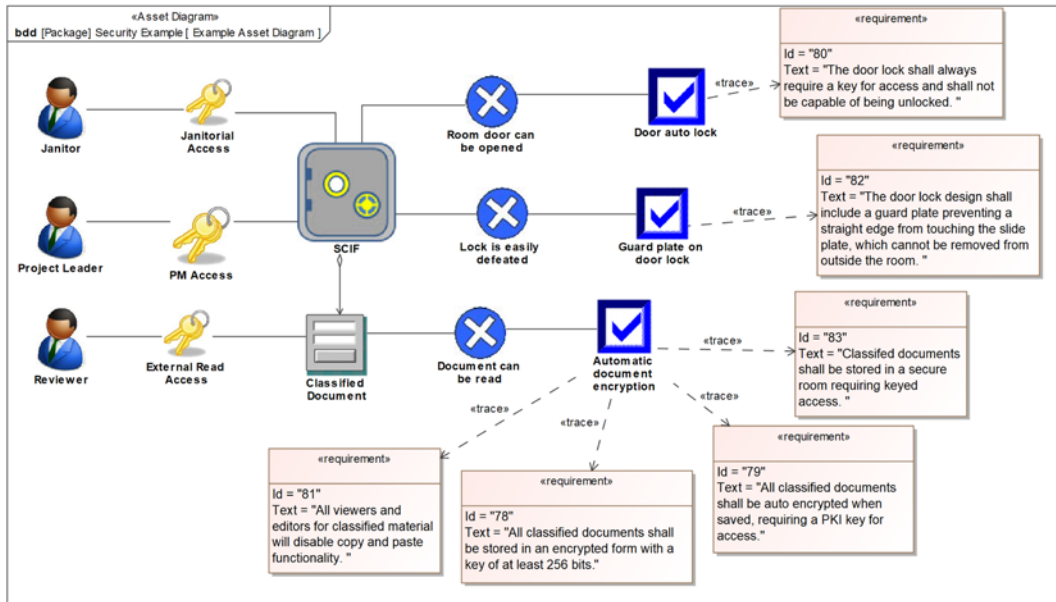


Figure 2.69: Asset diagram

Attack flow diagram

The last diagram of particular interest is the attack flow diagram. It is a specialized activity diagram with stereotyped actions to match the canonical attack chain, shown in Figure 2.70:

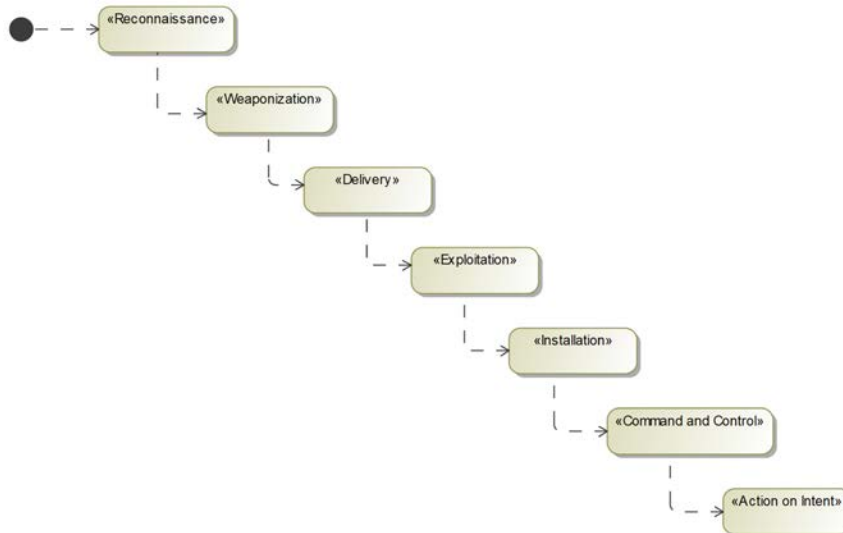


Figure 2.70: Canonical attack chain

The purpose of this diagram is to allow us to reason about how attacks unfold so that we can identify appropriate spots to insert security countermeasure actions. *Figure 2.71* shows an example of use:

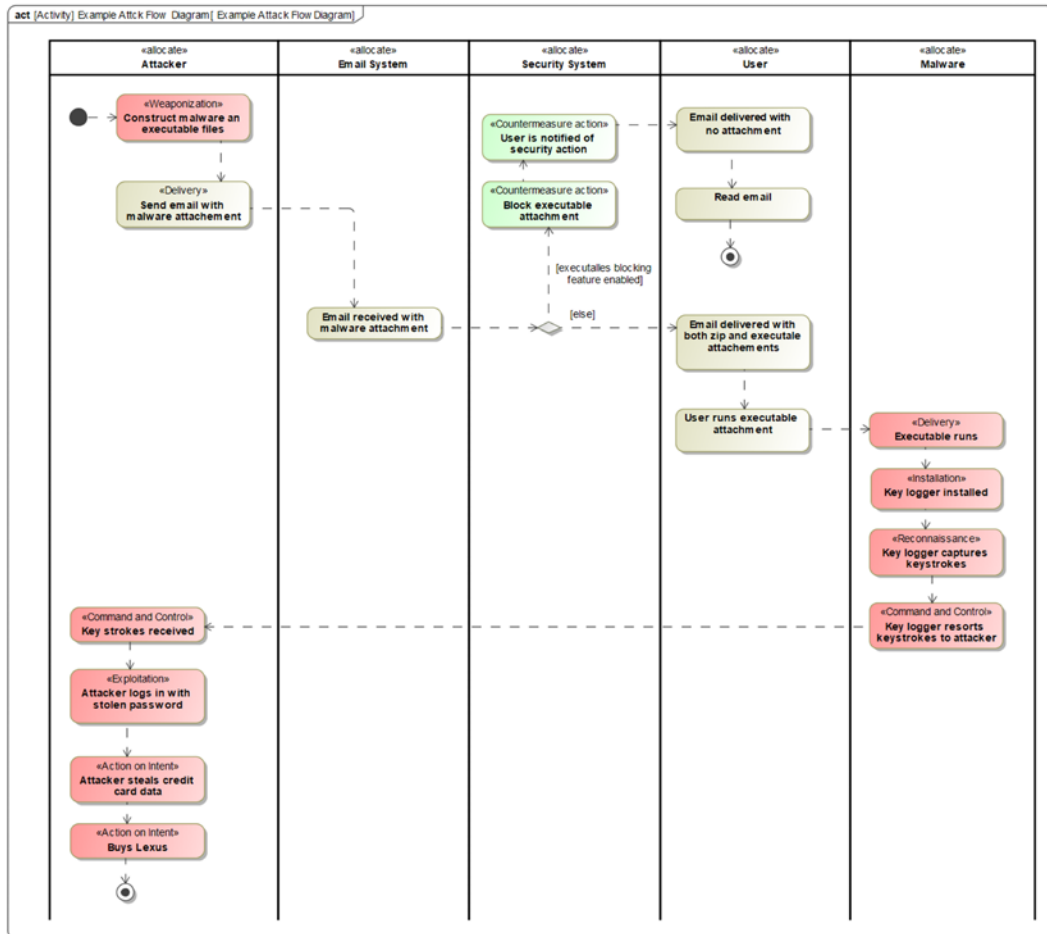


Figure 2.71: Example attack flow diagram

The stereotyped actions either identify the action as a part of the attack chain or identify the action as a countermeasure. The actions without stereotypes are normal user actions.

Tabular views

Tables and matrices can easily be constructed to summarize the threat analysis. The Security Posture Table, for example, is a tabular summary of assets, asset context, vulnerabilities, and countermeasures, and their important security-relevant metadata, including name, description, risk number, severity, probability, consequence, and impact.

Purpose

The purpose of this recipe is to identify system assets subject to attack, how they can be attacked, and where to best apply countermeasures.

Inputs and preconditions

A use case naming a capability of the system from an actor-use point of view that has been identified, described, and for which relevant actors have been identified. Note: this recipe is normally performed in parallel to one of the functional analysis recipes from earlier in this chapter.

Outputs and postconditions

The most important outcome is a set of requirements specifying how the system will mitigate or manage security concerns of the system. Additionally, a security posture concept is developed identifying the need for a set of security control measures, which is summarized in cyber-physical threat analysis.

How to do it

The workflow for this recipe is shown in *Figure 2.72*:

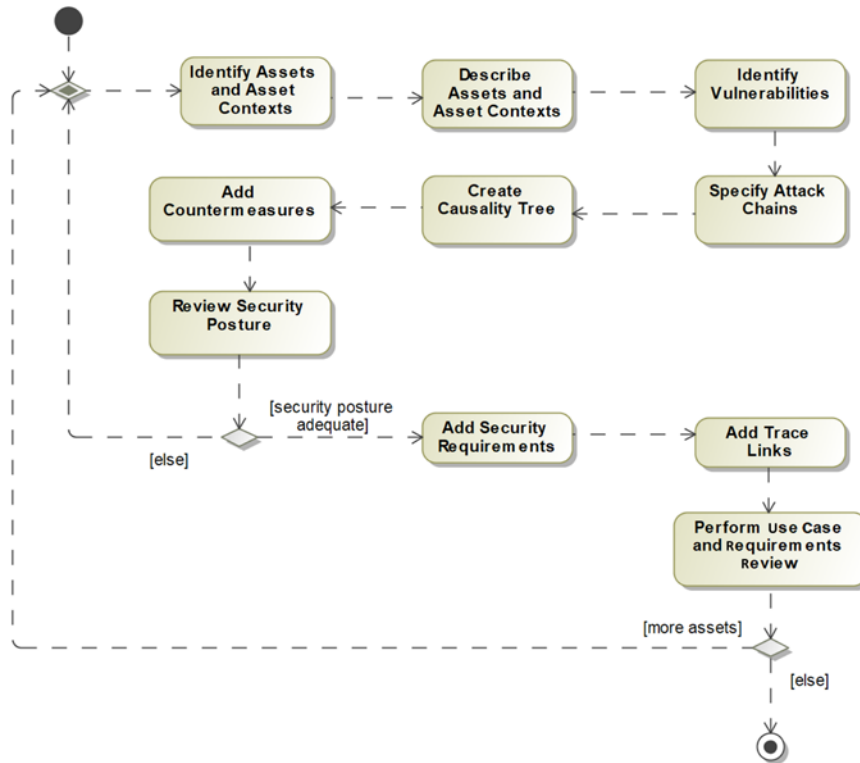


Figure 2.72: Security analysis workflow

Identify assets and asset contexts

Assets are system or environmental features of value that the system is charged to protect. Assets can be classified as being one of several types, including:

- **Information** – information of value, such as a credit card number
- **Currency** – money, whether in physical or virtual form
- **Resource** – a capability, means, source, system, or feature of value, such as access to GPS for vehicle navigation
- **Physical** – a tangible resource that can be physically compromised, threatened, or damaged, such as a gas supply for a medical ventilator
- **Service** – a behavior of the system that provides value, such as delivering cardiac therapy
- **Security** – a security measure that can be compromised as a part of an attack chain, such as a firewall

Of course, these categories overlap to a degree, so categorize your assets in a way that makes sense to you and your stakeholders.

Asset diagrams can be used to create and visualize assets. Assets are system or environmental features that have a value that your system is responsible to protect. Create one or more asset diagrams to capture the assets and asset contexts. You can optionally add access roles, permissions, and vulnerabilities, but the primary purpose is to identify and understand the assets.

Describe assets and asset context

Assets have several properties you may want to represent. At a minimum, you want to identify the asset kind and the value of the asset. Asset value is important because you will be willing to spend greater cost and effort to protect more valuable assets. You may also want to specify the asset availability, clearance, or access level required.

Identify vulnerabilities

Vulnerabilities are weaknesses in the system security field; in this context, we are especially concerned with vulnerabilities specific to asset and asset contexts. If you are using known technology, then sources such as the **Common Vulnerability Enumeration (CVE)**, see <https://cve.mitre.org/cve/>) or **Common Weakness Enumeration (CWE)**, see <https://cwe.mitre.org/>) are good sources of information.

Specify attack chains

Most attacks are not a single action, but an orchestrated series of actions meant to defeat countermeasures, gain access, compromise a system, and then perform “actions on objective” to exploit the asset. Use the attack flow or attack scenario diagrams to model and understand how an attack achieves its goals and where countermeasures might be effective.

Create causality tree

Express your understanding of the causal relations between attacks, vulnerabilities, and countermeasures on security analysis diagrams. These diagrams are similar to FTAs used in safety analysis.

Add countermeasures

Once a good understanding is achieved of the assets, their vulnerabilities, the attack chains used to penetrate the security field, and the causality model, you’re ready to identify what security countermeasures are appropriate and where in the security field they belong.

Review security posture

Review the updated security posture to ensure that you've identified the correct set of vulnerabilities, attack vectors, and countermeasures. It is especially important to review this in the context of the CVE and CWE.

Add security requirements

When you're satisfied with the proposed countermeasures, add requirements for them. As with all requirements, these should specify what needs to be done and not specifically how, since the latter concern is one of design.

Add trace links

Add trace links from your security analysis to the newly added requirements, from the associated use case to the requirements, and from the use case to the security analysis. If an architecture already exists, also add «satisfy» relations from the architectural elements to the security requirements, as appropriate.

Perform use case and requirements review

This final step of the recipe reviews the set of requirements for the use case, including any requirements added as a result of this recipe.

Example

For this example, we'll consider the use case **Measure Performance Metrics**. This use case is about measuring metrics such as heart rate, cadence, power, (virtual) speed, and (virtual) distance and uploading them to the connected app. The use case is shown in *Figure 2.73*:

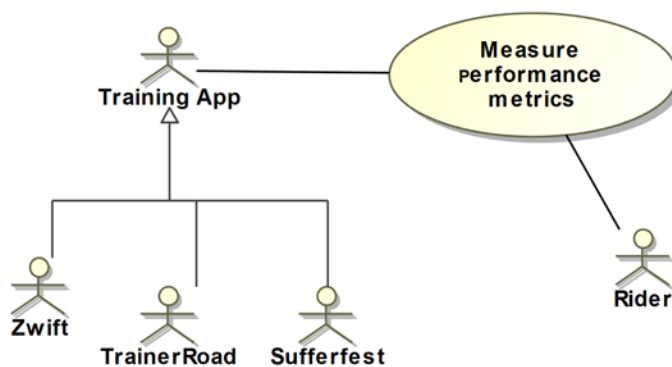


Figure 2.73: Measure performance metrics use case

Identify assets and asset contexts

There are two kinds of assets that might be exposed; the login ID and password used during the connection to the app and the rider's privacy-sensitive performance data. The assets of concern are the **Ride Login Data** and **Rider Performance Metrics**.

Other use cases potentially expose other assets, such as the **Update Firmware** use case exposing the system to malware, but those concerns would be dealt with during the analysis of the latter use case.

Describe assets and asset contexts

The asset metadata is captured during the analysis. It is shown in *Figure 2.74*. Both assets are of asset kind **INFORMATION_ASSET**. The **Rider Login Data** is a High valued asset, while the **Rider Performance Data** is of Medium value. To create this table, apply the «asset» stereotype to the elements in the table, add the security metadata to those elements, then create a table with **Tools > Generic Table Wizard**. Once the table is created, add the columns for the tagged values with the **Columns > Select Columns** tool:

#	Name	Documentation	asset kind	Access Permitted	Availability	Confidentiality	Value
1	Rider Login Data	Log-in data includes: <ul style="list-style-type: none"> • Rider ID • Password • app-specific login data Note: The app itself is required for maintaining security once the data is uploaded. The system is responsible for security during the internal storage and during upload.	INFORMATION	Read / write	During log in process and when systems is on (storage)	Sensitive	HIGH
2	Rider Performance Metrics	Performance data includes <ul style="list-style-type: none"> • Date and time of measurement • Heart rate • Power • Cadence • Speed 	INFORMATION	Read only	During transmission	Moderate	MEDIUM

Figure 2.74: Asset metadata

Identify vulnerabilities

Next, we look to see how the assets express vulnerabilities. We can identify three vulnerabilities that apply to both assets: impersonation of a network, impersonation of the connected app, and sniffing the data as it is sent between the system and the app. See *Figure 2.75*:

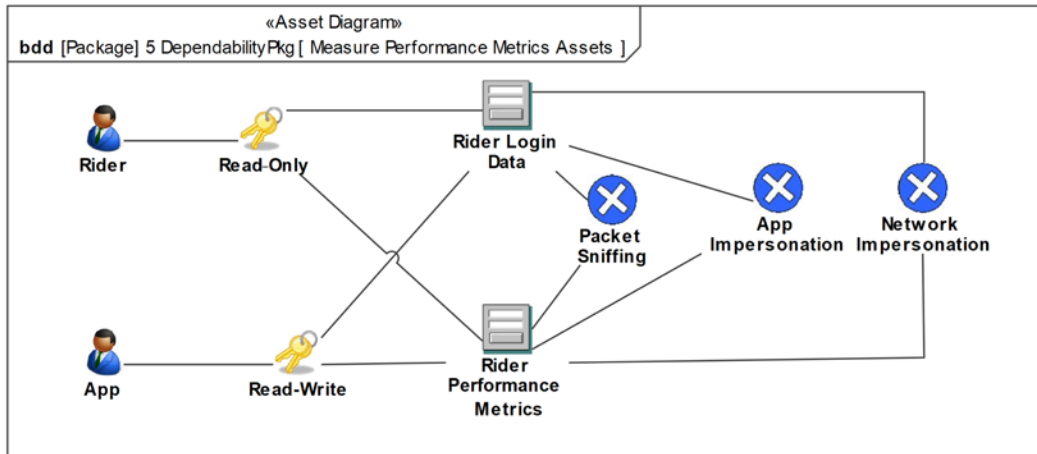


Figure 2.75: Asset vulnerabilities

Specify attack chains

Figure 2.76 shows the attack chain for the **Measure Performance Metrics** use case. It is further decomposed into a *call behavior*, shown in *Figure 2.77*.

These attack chains show the normal processing behavior along with the attack behaviors of the adversary and the mitigation behaviors of the system:

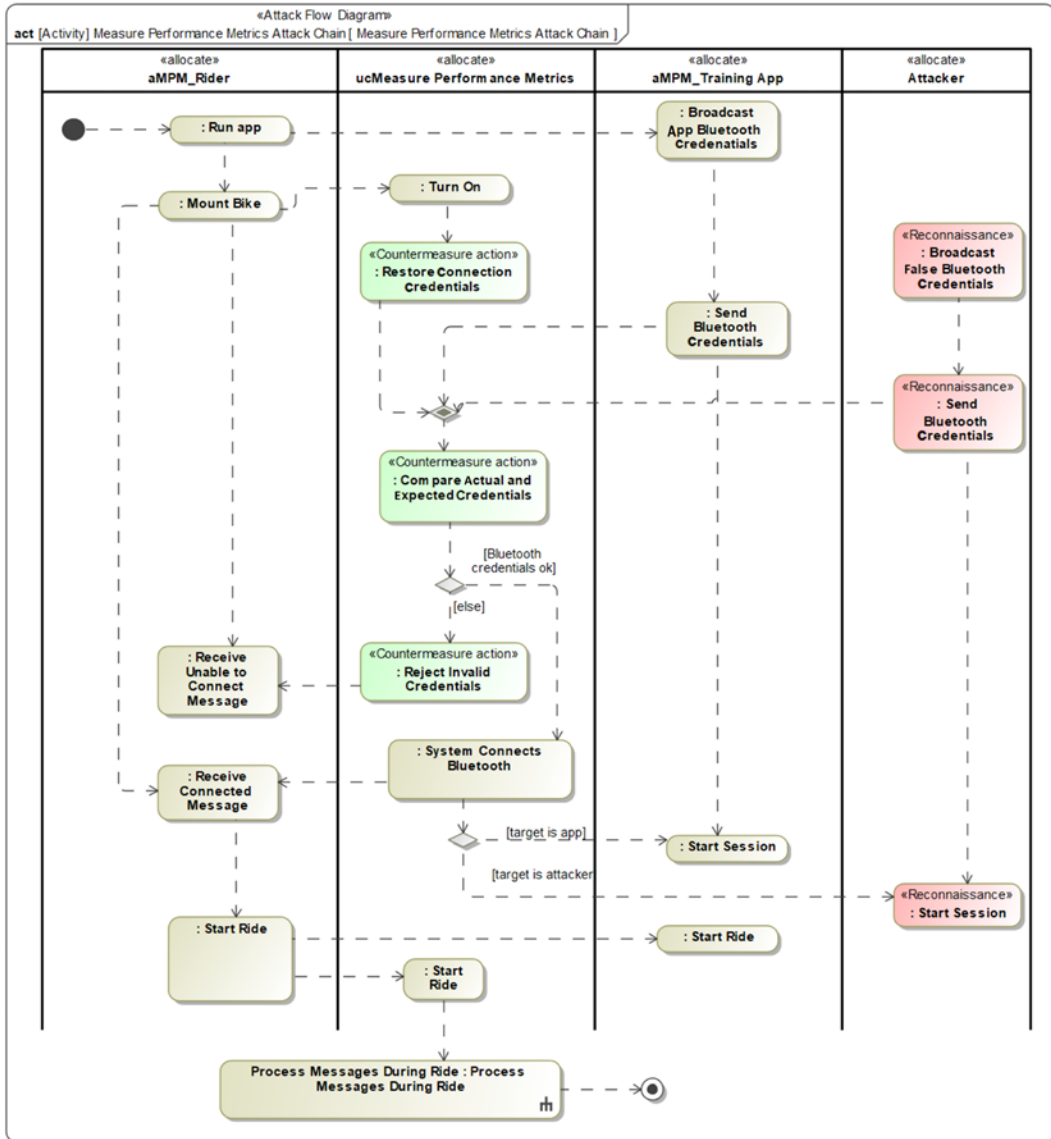


Figure 2.76: Measure performance metrics attack chain

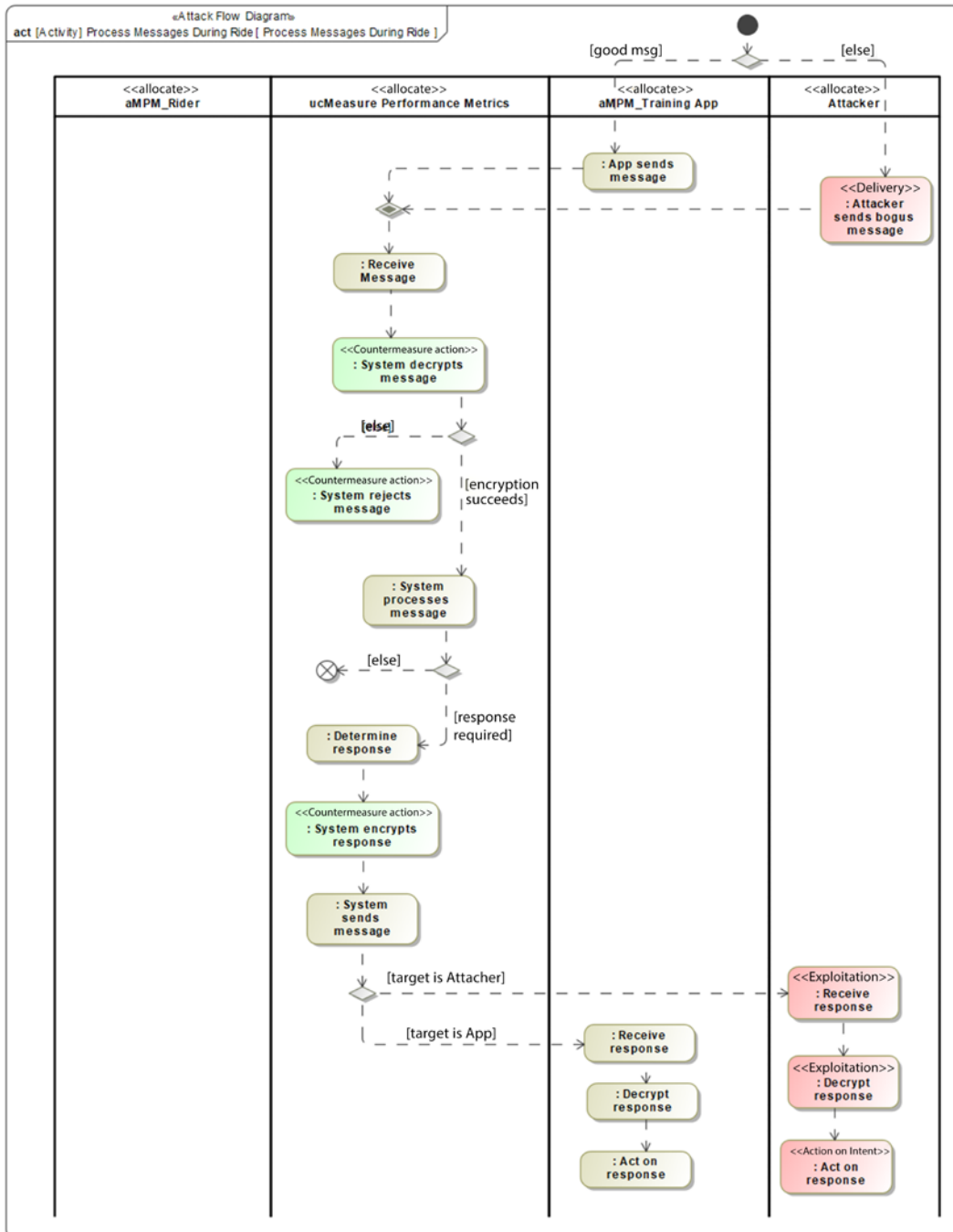


Figure 2.77: Process messages attack chain

Create causality tree

Now that we've identified and characterized the assets, vulnerabilities, and attacks, we can put together a causality model. This is shown in *Figure 2.78* for the compromising of login data and credentials, and in *Figure 2.79* for the rider metric data:

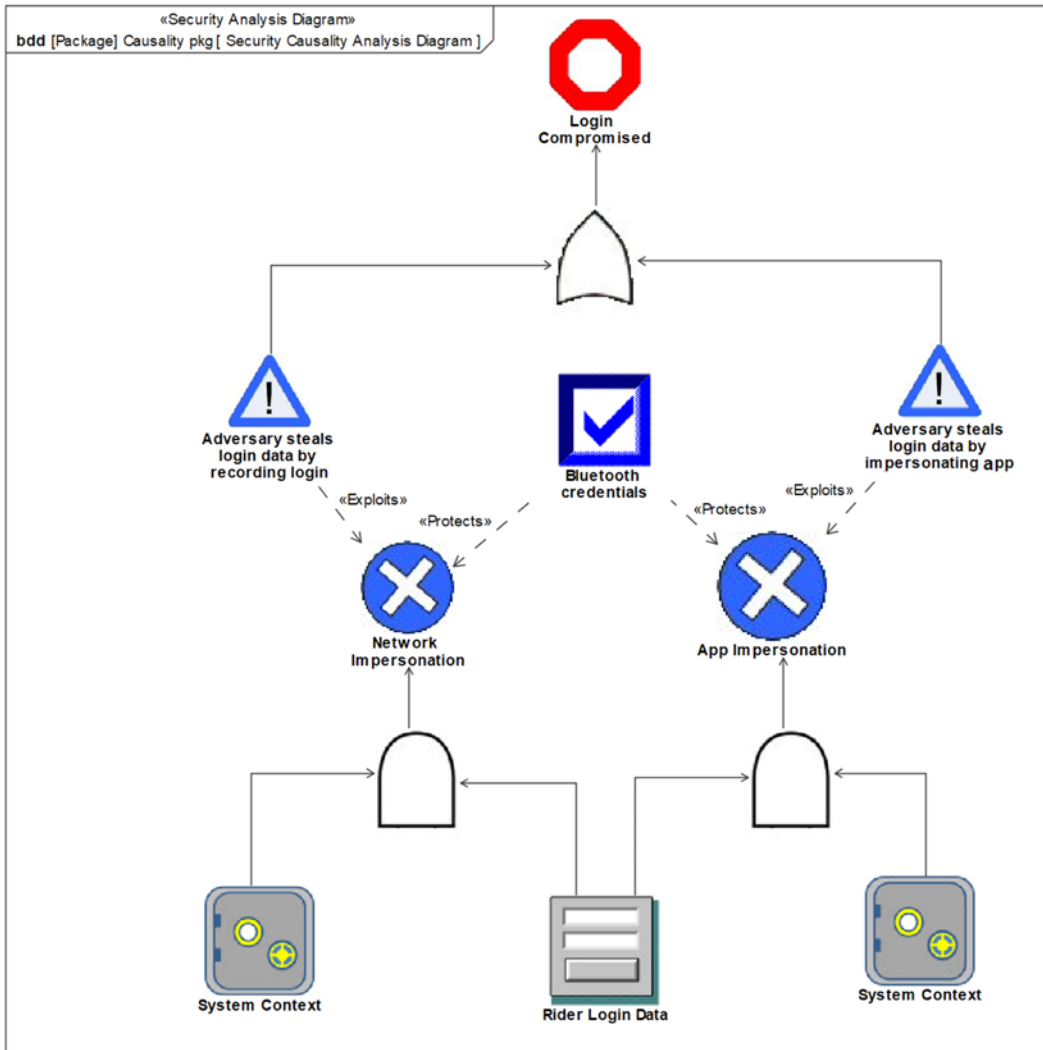


Figure 2.78: Security analysis diagram for rider login data

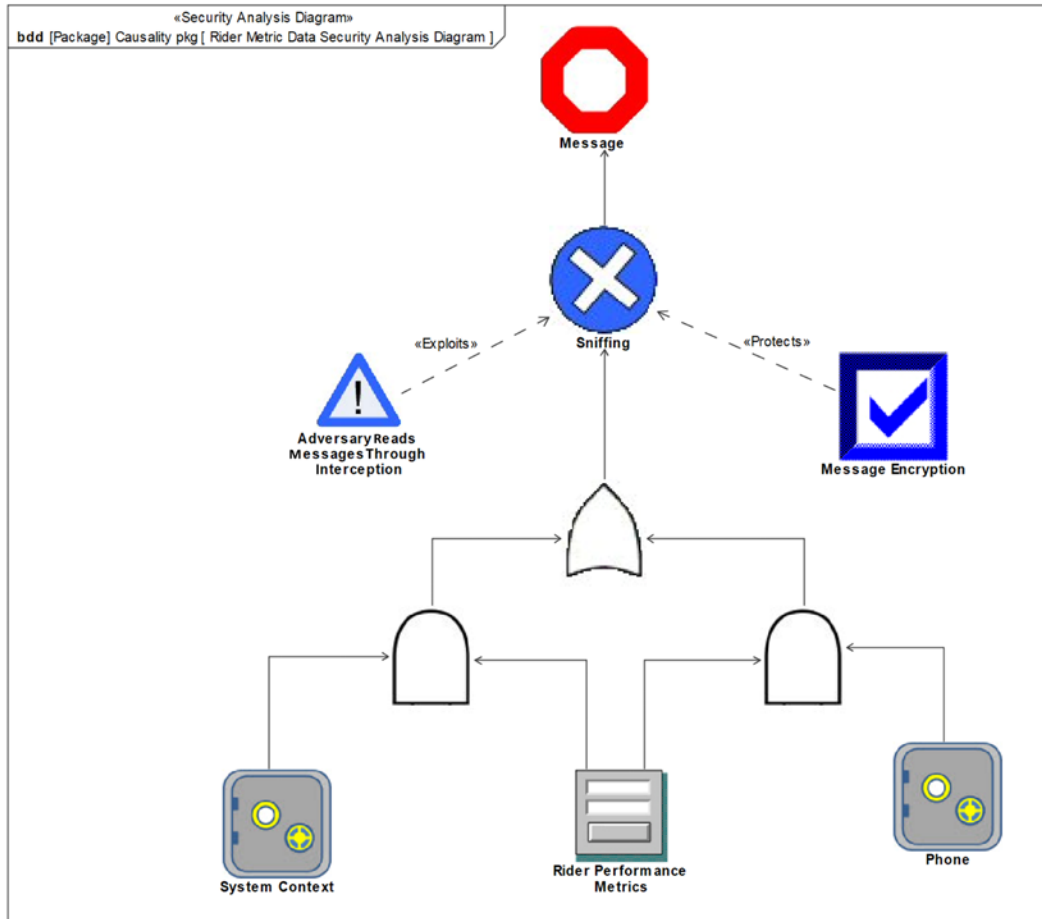


Figure 2.79: Security analysis diagram for rider metric data

Add countermeasures

We can see in the previous two figures that our causality diagram has identified two security countermeasures: the use of credentials for Bluetooth connection and the addition of encryption for message data.

Review security posture

In this step, we review our security posture. The security posture is the set of assets, asset contexts, vulnerabilities, and countermeasures. In this case, the assets are the rider login data and the rider metrics data. The login data includes the username and password.

The metrics data includes all the metrics gathered, including speed, distance, elapsed time, date and time of workout, power, cadence, and heart rate.

There are two asset contexts: the system itself and the phone hosting the app. The latter context is out of the system's scope and we have limited ability to influence its security, but we can require the use of the protections it provides. Notably, this includes Bluetooth credentials and encryption of data during transmission. Other use cases may allow us better control over the security measures in this asset context. For example, the use case **Configure System** uses a configuration app of our own design, which we can ensure stores data internally in encrypted form; we have no such control over the third-party training apps.

We identified three vulnerabilities. During login, the system can be compromised either with network or app impersonation. By pretending to be a trusted conduit or trusted actor, an adversary can steal login information. We address these concerns with two explicit countermeasures: message encryption and the use of credentials. Security could be further enhanced by requiring multi-factor authentication, but that was not seen as necessary in this case. During rides, the system transmits metric data to the app for storage, display, and virtual simulation. An adversary could monitor such communications and steal that data. This is addressed by encrypting messages between the system and the app.

Add security requirements

The security requirements are simply statements requiring the countermeasure design and implementation. In this case, there are only two such requirements:

- The system shall require the use of a Bluetooth credential agreement between the system and the app to permit message traffic.
- The system shall encrypt all traffic between itself and the app with at least 128-bit encryption.

Add trace links

The new requirements trace to the **Measure Performance Metrics** use case. Further, trace links are added from the countermeasures to the requirements, linking our analysis to the requirements.

Perform use case and requirements review

We can now review the use case, functional analysis, and dependability analyses for completeness, accuracy, and correctness.

Specifying Logical System Interfaces

System interfaces identify the sets of services, data, and flows into and out of a system. By *logical interfaces*, we mean abstract interfaces that specify the content and precision of the flows but not their physical realization. For example, a system interface to a radar might include a message **herezaRadarTrack(r: RadarTrack)** as a SysML event carrying a radar track as a parameter without specifying what communication means will be used, let alone the bit-mapped structure of the realizing 1553 Bus message. Nevertheless, the specification of the interface allows us to consider the set of services requested of the system by actors, the set of services needed by the system from actors, and the physical and information flow across the system boundary.

The initial set of interfaces is a natural outcome of use case analysis. Each use case characterizes a set of interactions of the system with a group of actors for a similar purpose. These interactions necessitate system interfaces. This recipe will focus on the identification of these interfaces and the identification of the data and flows that they carry; the actual definition of these data elements is described in the last recipe in this chapter, *Creating the Logical Data Schema*.

The logical interfaces from a single use case analysis are only a part of the entire set of system interfaces. The set interfaces from multiple use cases are merged together during system architecture definition. This topic is discussed in the recipes of the next chapter, *Developing Systems Architectures*. Those are still logical interfaces, however, and abstract away implementation detail. The specification of physical interfaces from their logical specification is described in *Chapter 4, Handoff to Downstream Engineering*.

A Note about SysML Ports and Interfaces

SysML supports a few different ways to model interfaces, and this is intricately bound up with the topic of ports. SysML has the *standard port* (from UML), which is typed by an interface. An interface is similar to an abstract class; it contains specifications of services but no implementation. A block that realizes an interface must provide an implementation for each operation specified within that interface. UML ports are typed by the interfaces they support. A port may either *provide* or *require* one or more interfaces. If an interface is *provided* by the system, that means that the system must provide an implementation that realizes the requested services. If an interface is *required*, then the system can request an actor to provide those services. These services can be synchronous calls or asynchronous event receptions and can carry data in or out, as necessary. Note that the difference between provided and required determines where the services are implemented and not the direction of the data flow.

These interfaces are fundamentally about services that can, incidentally, carry data. SysML also defines *flow ports* that allow data or flow to be exchanged without services being explicitly involved. Flow ports are bound to a single data or flow element and have an explicit flow direction: either into or out of the element. Block instances could bind flow ports to internal value properties and connect them to flow ports on other blocks that were identically typed.

SysML 1.3 and later deprecate these standard and flow ports and added the *proxy port*. To be clear, “deprecated” means that the use of these ports is “discouraged” but they are still part of the standard, so feel free to use them. Proxy ports essentially combine both the standard ports and flow ports. The flows specified sent or received by a proxy part are defined to be *flow properties* rather than value properties, a small distinction in practice. More importantly, proxy ports are not typed by *interfaces* but rather by *interface blocks*. Interface blocks are more expressive than interfaces in that they can contain flow properties, nested parts, and proxy ports themselves. This allows the modeling of some complex situations that are difficult with simple interfaces. With proxy ports, gone are the “lollipop” and the “socket” notations; they are replaced by the port and port conjugate (“~”) notation. In short, standard ports use interfaces but proxy ports use interface blocks. The examples in this book exclusively use proxy ports and interface blocks and not standard ports.

This recipe specifically refers to the identification and specification of logical interfaces during use case specification, as experience has shown this is a highly effective means for identifying the system interfaces.

Continuous flows

Systems engineering must contend with something that software development does not: continuous flows. These flows may be information but are often physical in nature, such as material, fluids, or energy. SysML extends the discrete nature of UML activities with the «continuous» stereotype for continuous flows. The «stream» stereotype (from UML) refers to object flows (“tokens”) that arrive as a series of flow elements at a given rate. «continuous» is a special case where the time interval between streaming flow elements approaches zero. In practice, «stream» is used for a flowing stream of discrete elements, often at a rate specified with the SysML «rate» stereotype, while «continuous» is used for truly continuous flows. An example of «stream» might be a set of discrete images sent from a video camera at a rate of 40 frames per second. An example of a «continuous» flow might be water flowing through a pipe or the delivery of electrical power.

In my work, I use these stereotypes on flows in sequence diagrams as well. I do this by applying the stereotypes to messages and through the use of a *continuous* interaction operator. To create the flows, I use a standard **Message** on the sequence diagram and add item flows to it with Cameo's **Item Flow Manager**. To enable that, the relations between the elements typing the lifelines must have such flows defined. *Figure 2.80* shows such a context for a system that heats water for bathing:

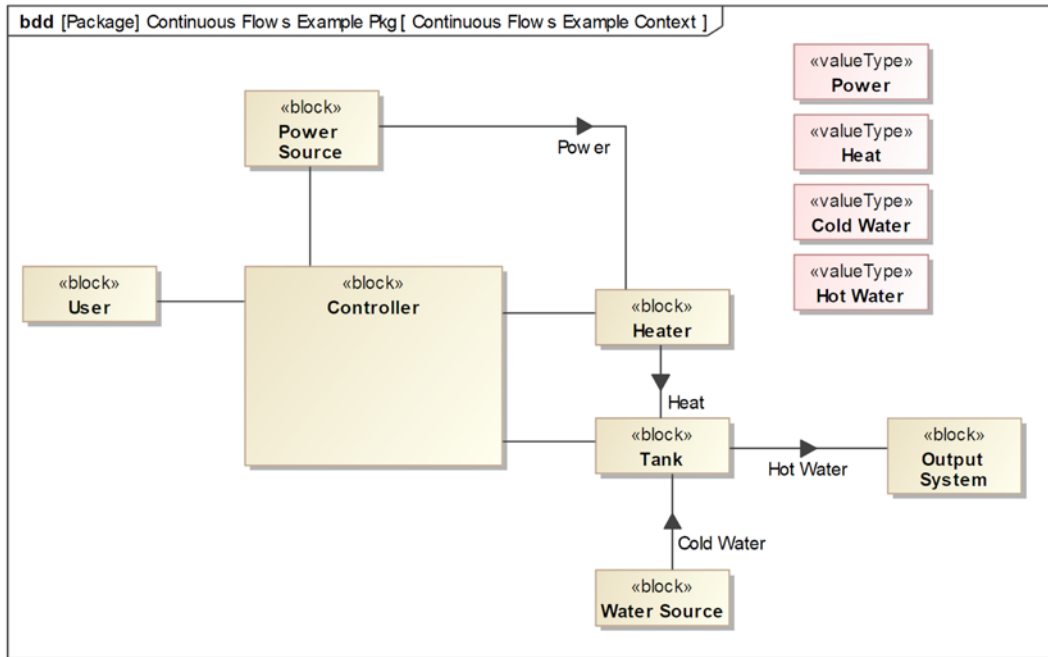


Figure 2.80: Structural context for continuous flows example

Since the flows are defined, they can be added to the messages sent between the lifelines defined by those blocks. While the Rhapsody tool allows me to create new interaction operators, Cameo does not. Thus, in Cameo, I use a **Critical Region** interaction operator and apply the *continuous* stereotype to it.

The resulting sequence diagram is shown in *Figure 2.81*:

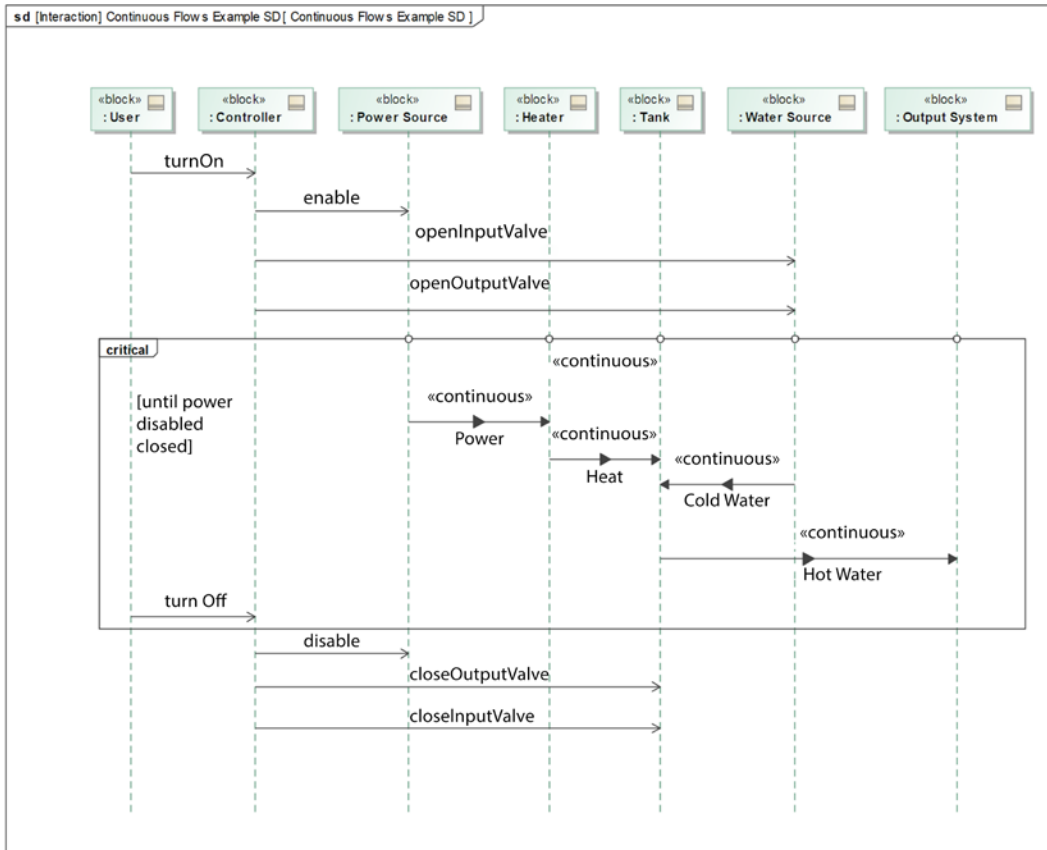


Figure 2.81: Continuous flows on sequence diagrams

The figure shows flows (messages with item flows) marked with the «continuous» stereotype. This indicates that the flow is continuous throughout its execution context. That context can be the entire diagram or limited to an interaction operator, as it is in this case. Within a context, there is no ordering among «continuous» flows; this is in contrast to the normal *partial ordering* semantics of SysML sequence diagrams in which “lower in the diagram” corresponds (roughly) to “later in time.” However, «continuous» flows are active throughout their execution context and so the ordering of continuous flows is inherently meaningless.

The use of the interaction operator with the *continuous* stereotype defines the context during which the flows are continuous and unordered. Any normal (non-continuous) messages within the interaction operator still operate via the normal partial ordering semantics.

Purpose

The purpose of this recipe is to identify the exchange of services and flows that occur between a system and a set of actors, especially during use case analysis.

Inputs and preconditions

The precondition is that a use case and set of associated actors have been identified.

Outputs and postconditions

Interfaces or interface blocks are identified, as well as which actors must support which interfaces or interface blocks.

How to do it

Figure 2.82 shows the workflow for this recipe. This overlaps with some of the other recipes in this chapter but focuses specifically on the identification of the system interfaces:

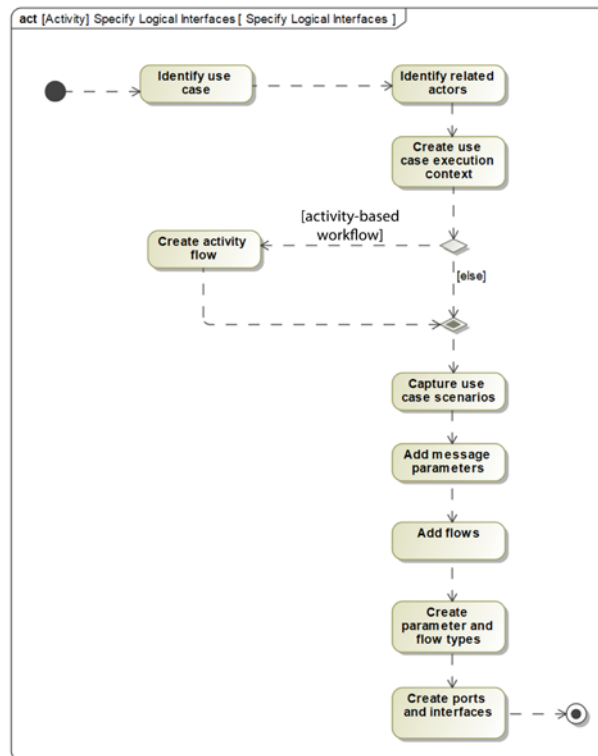


Figure 2.82: Specify logical interfaces workflow

Identify use case

This first step is to identify the generic usage of the system that will use the to-be-identified system interfaces.

Identify related actors

The related actors are those people or systems outside our scope that interact with the system while it executes the current use case. These actors can send messages to the system, receive messages from the system, or both using the system interfaces.

Create the execution context

The use case execution context is a kind of modeling “sandbox” that contains an executable component consisting of executable elements representing the use case and related actors. The recommended way to achieve this is to create separate blocks representing the use case and the actors, connected via ports. Having an isolated simulation sandbox allows different system engineers to progress independently on different use case analyses.

Create activity flow

This step is optional but is a popular way to begin to understand the set of flows in the use case. This step identifies the actions – event reception actions, event send actions, and internal system functions – that define the set of flows of the use case.

Capture the use case scenarios

Scenarios are singular interactions between the system and the actors during the execution of the use case. When working with non-technical stakeholders, they are an effective way to understand the desired interactions of the use case. We recommend starting with normal, “sunny day” scenarios before progressing to edge case and exceptional “rainy day” scenarios. It is important to understand that every message identifies or represents one or more requirements and results in messages that must be supported in the derived interfaces. If the **Create Activity Flow** task is performed, then the sequence diagrams can be derived from those flows.



Use asynchronous events for all **actor > system and system > actor** service invocations. This is specifying the logical interfaces and so the underlying communication mechanism should be abstracted away. Later, in the definition of the physical interfaces and data schema, these can be specified in a technology-specific fashion.

Add message parameters

These signals often carry data. This data should be explicitly modeled as signal attributes that become message arguments.

Add flows

Use UML “flows” to indicate discrete flows of information, material, fluids, or energy exchanges between the system and an actor that are not intimately bound to a service request. Stereotype these flows as «continuous» when appropriate, such as the flows of energy or fluids.

Create parameter and flow types

The event arguments must be typed by elements in the logical data schema (see the recipe *Creating the Logical Data Schema*). The same is true for flow types. Because these types are specifications, they will include not only some (logical) base types, but also units, ranges, and other kinds of metadata.

Create ports and interfaces

Based on the defined interaction of the system with the actors while executing the use case, add ports between the actor and use case blocks and type these ports with interfaces. These interfaces will enumerate the services and flow going between the actors and the system. Technically speaking, this can be done using UML standard ports and SysML flow ports, or the more modern SysML 1.3 proxy ports.

Example

This example will use the **Control Resistance** use case but we will a different approach than was used for this use case in the *Functional Analysis with Activities* recipe, just to demonstrate that there are alternative means to achieve similar goals in MBSE.

Identify use case

The **Control Resistance** use case focuses on how resistance is applied to the pedals in response to simulated gearing, conditions, and user-applied force. The description was shown previously in *Figure 2.19*.

Identify related actors

There are two actors here for this use case: **Rider** and **Training App**. The **Rider** provides power to and receives resistance from the pedals. The **Training App** is sent the rider power information.

Create the execution context

Creating the execution context creates blocks that represent the actors and use cases for the purpose of analysis and simulation. They contain proxy ports that will be defined by the interfaces identified in this workflow:

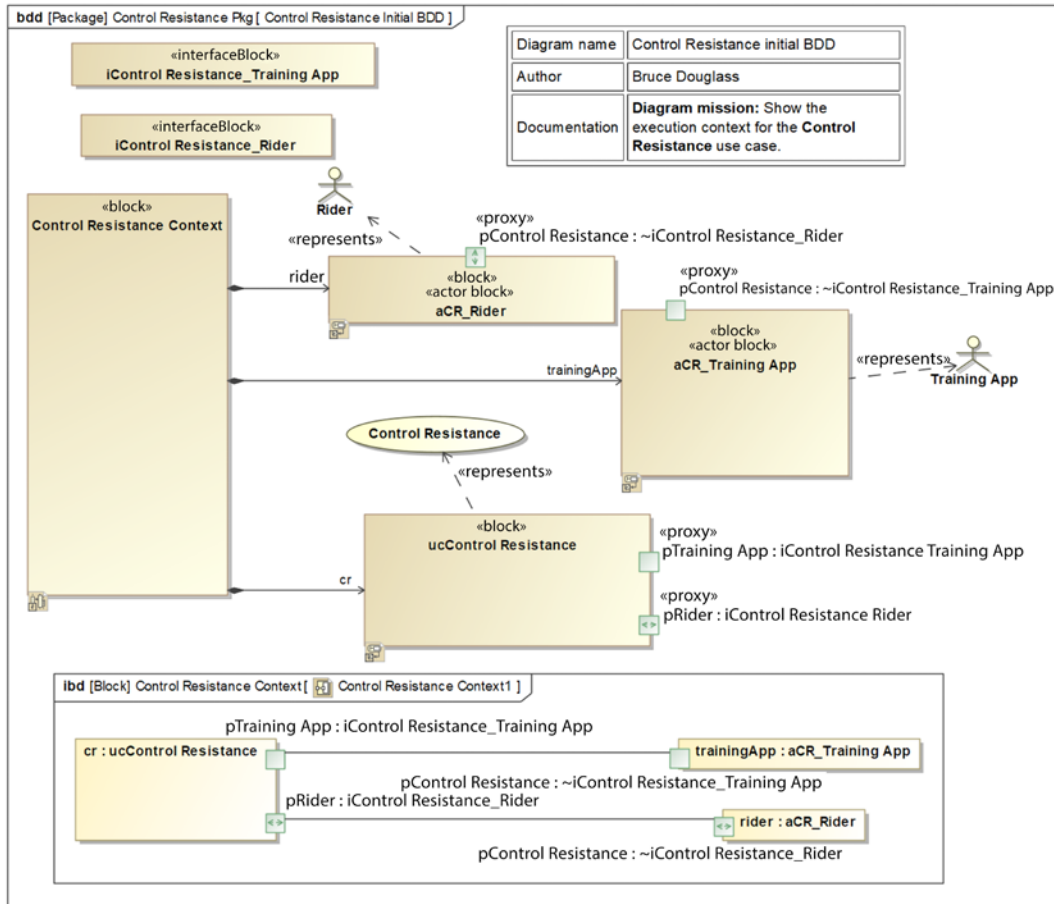


Figure 2.83: Control resistance execution context for interface definition

Create activity flow

The activity flow shows the object and control flows for the use case. In this example, we will show continuous flows in addition to discrete flows. The activity is decomposed into three diagrams. The top level is shown in *Figure 2.84*. This diagram shows the system sending bike data to the **Training App** on the left. The center part, containing the **Determine Base Pedal Resistance call behavior**, does the bulk of the functional work of the use case.

Note that it takes the computed base resistance on the pedal and adjusts it for its current angular position. On the right, pedal cadence is computed from pedal speed.

Discrete events, such as turning the system on and off or changing the gears, are simple to model in the activity diagrams; they can easily be modeled as either *signal receptions* for incoming events or *send signal actions* for outgoing events. Of course, these events can carry information as arguments as needed.

It is somewhat less straightforward to model continuous inputs and outputs. These are modeled on the flow properties **pedalSpeed** and **pedalResistance**. The activity uses a *change event* for monitoring when the pedal speed changes and uses this, along with other information, to compute and apply resistance at the point of the pedal:

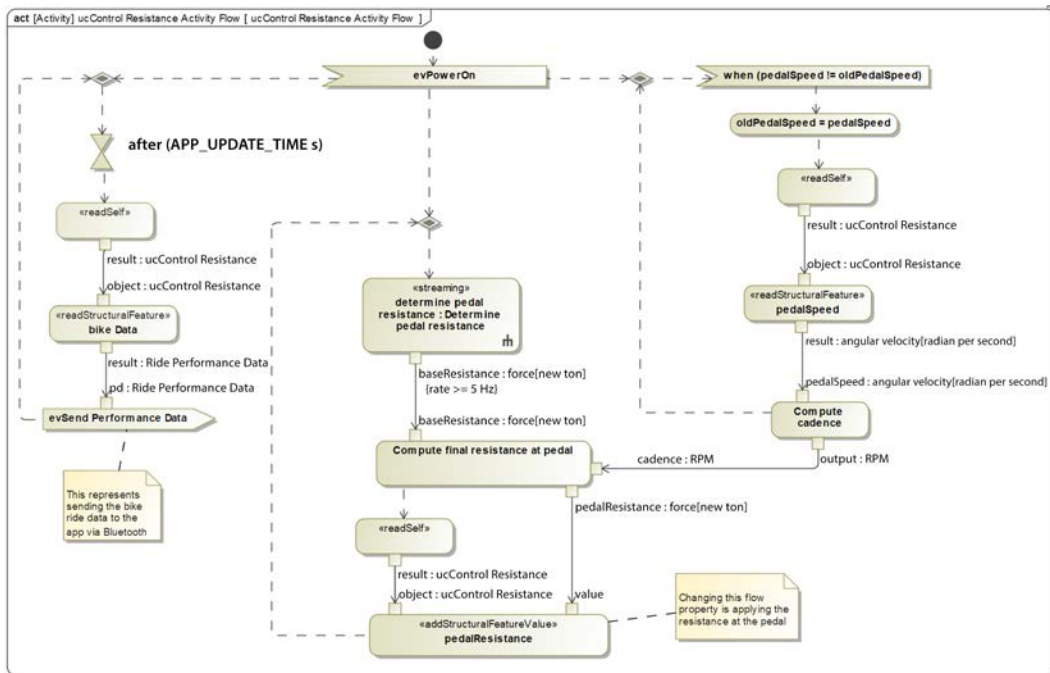


Figure 2.84: Control resistance activity flow for creating interfaces

Figure 2.85 shows the details for the **Determine Pedal Resistance** call behavior from the previous figure. In it, we see the base pedal resistance is computed using another call behavior, **Compute Bike Physics**. The **Determine Pedal Resistance** behavior never terminates (at least until the entire behavior terminates) so it uses a «rate» stereotype to indicate the data output on these activity parameter streams, although Cameo doesn't show it graphically. I added a constraint to show the rate.

Remember that normal activity parameters require the activity to terminate before they can output a value:

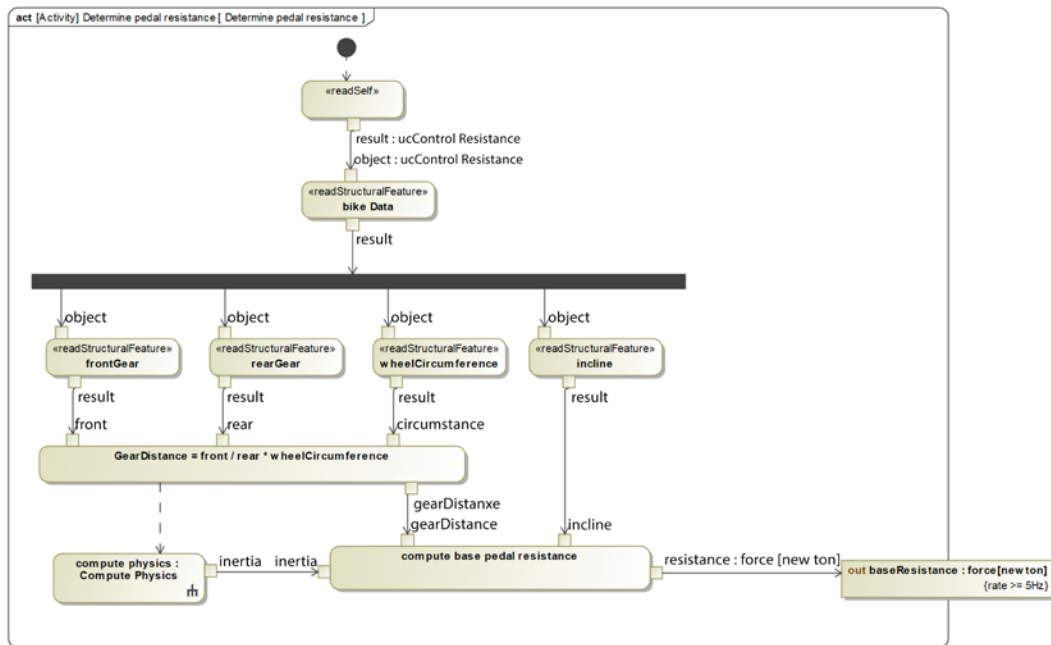


Figure 2.85: Determine base pedal resistance activity

Lastly, we have the call behavior **Compute Physics**, shown in *Figure 2.86*. This simulates the physics of the bike using the rider mass, current incline, current speed, and the power applied by the rider to the pedal to compute the resistance to movement, and couples that with the combined bike and rider inertia to compute the simulated bike speed and acceleration:

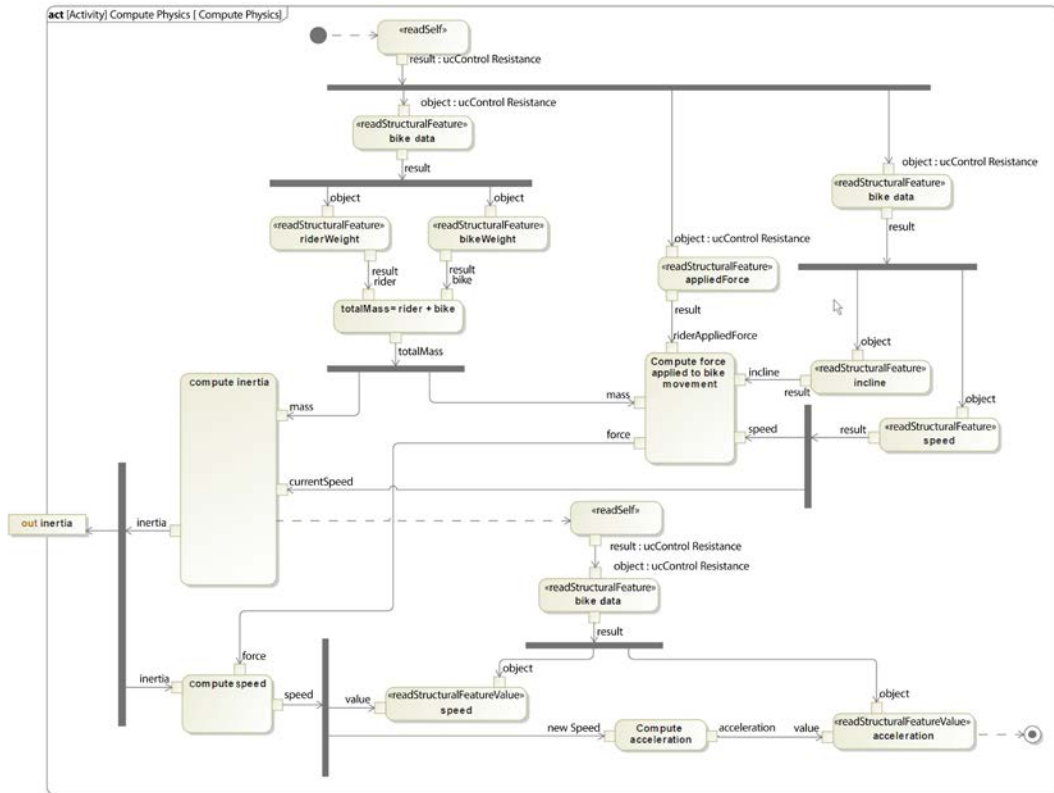


Figure 2.86: Compute bike physics activity

Capture the use case scenarios

The interfaces can be produced directly from the activity model but it is often easier to produce them from a set of sequence diagrams derived from the activity model. Accept event actions and flows on activities don't indicate the source but this is clearly shown in the sequences. If you do create a set of sequence diagrams, it is adequate to produce the set of scenarios such that all inputs and outputs and internal flows are represented in at least one sequence diagram.

The next three diagrams show the functional behavior modeled to follow the same structure as the activity model. *Figure 2.87* shows the high-level behavior. Note the use of «continuous» flows for the resistance to movement supplied by the system (**pedal resistance**) and the speed of the pedal (**pedal speed**). The *critical region* interaction operator is stereotyped as «continuous» and defines a scope for continuous flows. Note the use of a parallel interaction operator to show things that are going on concurrently, and loops and critical regions nested within it. The referenced interaction occurrence **Determine Pedal Resistance Scenario** references the sequence diagram shown in *Figure 2.7*:

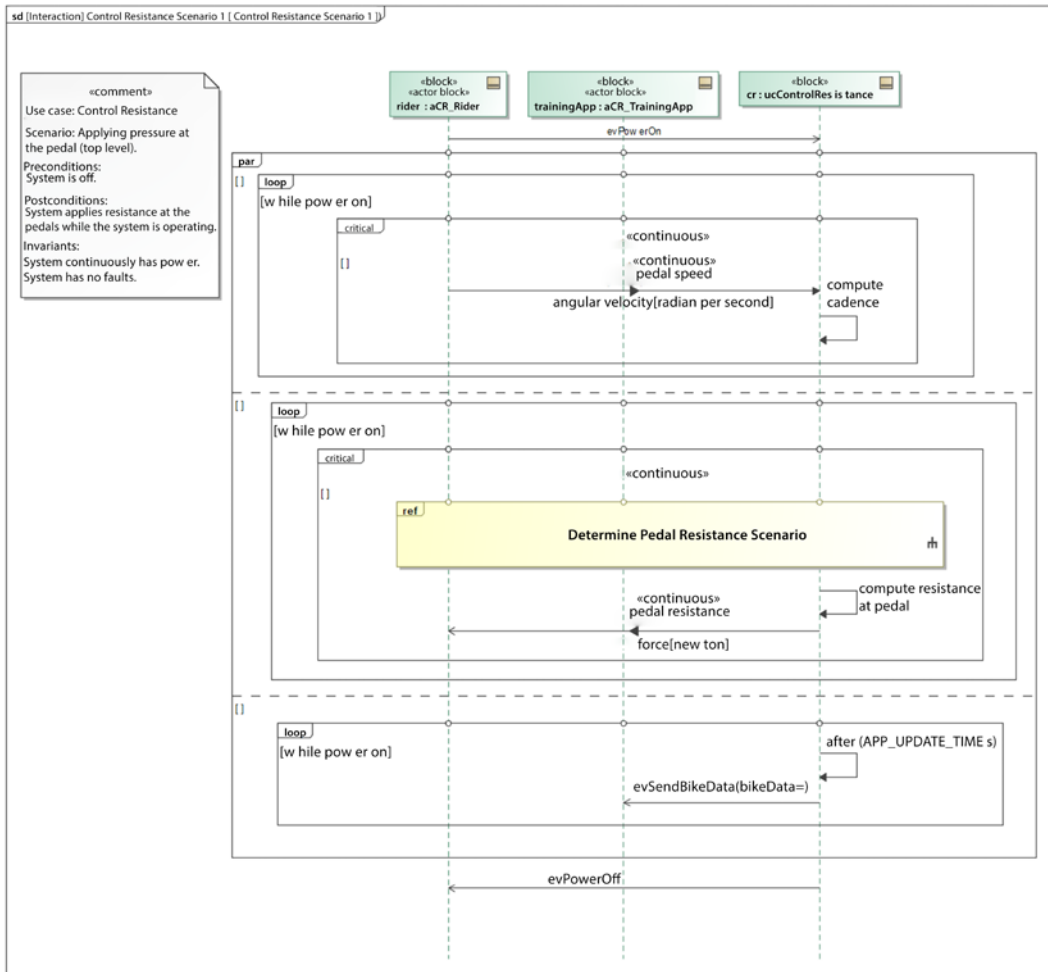


Figure 2.87: Control resistance scenario

Throughout the entire scenario shown in *Figure 2.88*, the continuous flows are active, so no scoping continuous interaction occurrence is required. This scenario also includes a nested scenario, the referenced **Compute Physics**, shown in *Figure 2.89*. This scenario includes a continuous flow but doesn't need a critical region stereotyped as continuous because the scope of the continuous flow is the entire sequence diagram:

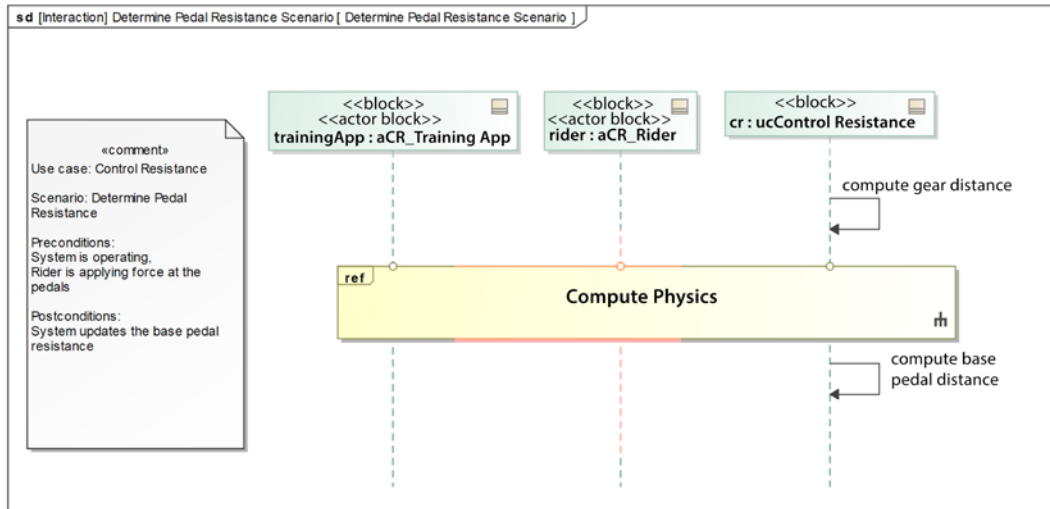


Figure 2.88: Determine pedal resistance scenario

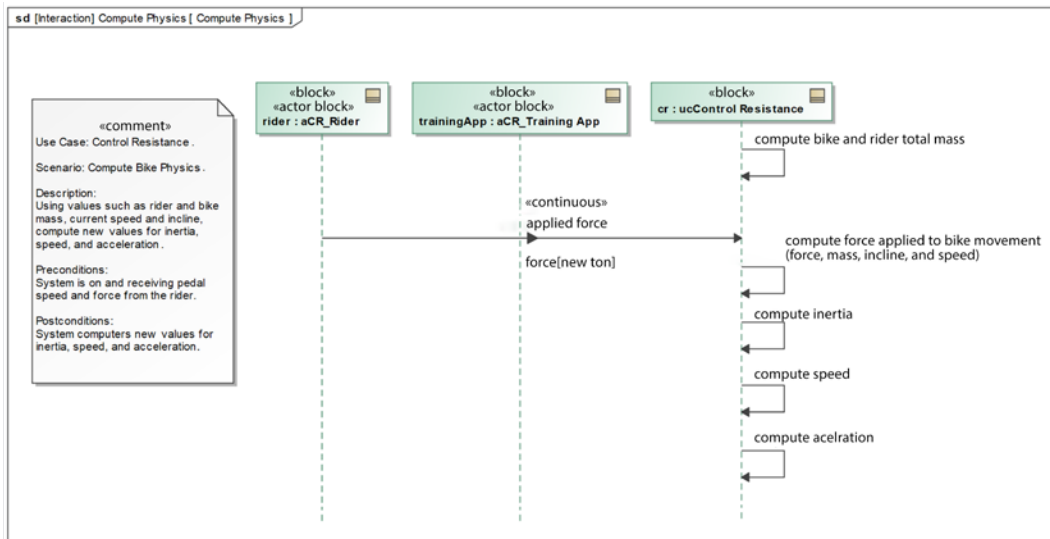


Figure 2.89: Compute bike physics

Add message parameters

Rather than show all the stages of development of the scenarios, the previous step is shown already including the message parameters.

Add flows

Rather than show all the stages of development of the scenarios, the previous step is shown already including the continuous flows.

Create parameter and flow types

The detail of how to create all the types is the subject of the next recipe, *Creating the Logical Data Schema*. The reader is referred to that recipe for more information.

Create ports and interfaces

Now that we have the set of flows between the actors and the system and have characterized them, we can create the interfaces. In this example, we are using the SysML 1.3 standard approach of using proxy ports and interface blocks, rather than standard ports, flow ports, and standard interfaces. This is a bit more work than using the older approach but is more modern and descriptive.

Figure 2.83 shows the execution context of the use case analysis for the **Control Resistance** use case. Although the diagram is itself a block definition diagram, Cameo allows you to show diagrams within diagrams; thus the corresponding internal block diagram is shown at the bottom of the figure. The IBD parts of the use case block **Uc_ControlResistance** and the actor blocks **aCR_Rider** and **aCR_TrainingApp** with SysML connectors between the appropriate ports. Note that, by convention, the unconjugated form interface is referenced at the use case block end of the connector and the conjugated form is used at the actor end, as indicated by the tilde (“~”) in front of the interface block name.

At the top of the diagram are the (current empty) interface blocks that will be elaborated on in this step. Later, during architecture development, these interface blocks will be added to the interfaces provided by the system and decomposed and allocated to the subsystems.

A note about naming conventions

The package **Functional Analysis Pkg::Control Resistance Pkg** holding the model used in this recipe provides a sandbox for the purpose of analyzing the **Control Resistance** use case. To that end, a block representing the use case is created and given the name **uc_ControlResistance**.

For the actors, local blocks are created for the purpose of analysis and are given the names of a (for actor), followed by the initials of the use case (**CR**) followed by the name of the actor (with white space removed). So, these sandbox actor blocks are named **aCR_Rider** and **aCR_TrainingApp**. The interfaces are all named **i <use case block name> “_” <actor block name>**, as in **iControlResistance_Rider**. This makes it easy to enforce naming consistency at the expense of sometimes creating lengthy names.

Since the flows are shown on the sequence diagrams, it is a simple matter to add these elements to the interface blocks:

- For each message from the use case to an actor, add that signal reception as required to the interface block defining that port.
- For each message from an actor to the use case, add that signal reception as provided to the interface block defining that port.
- For each flow from the use case to an actor, add that flow as an output flow property to the interface block defining that port.
- For each flow from an actor to the use case, add that flow as an input flow property to the interface block defining that port.

The result is shown in *Figure 2.90*. Note that the event receptions are either **provided (prov)** or **required (reqd)** while the flow properties are either in or out (from the use case block perspective):

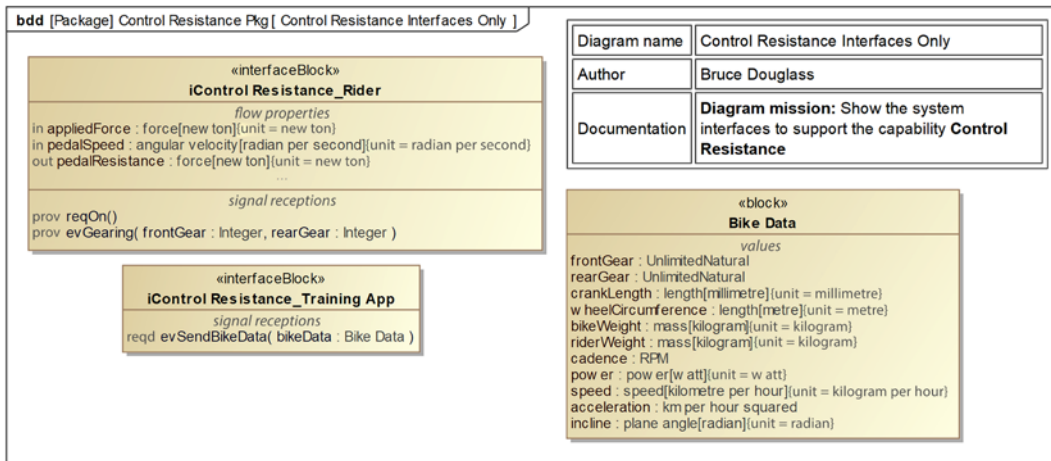


Figure 2.90: Created interface blocks

You should note that these are, of course, logical interfaces. As such, they reflect the intent and content of the messages but not their physical realization. For example, bike data sent to the training app is modeled in the logical interface as an event, but the implementation will actually be as a Bluetooth message. The creation of physical interfaces from logical ones is discussed in the last chapter.

Creating the Logical Data Schema

A big part of the specification of systems is specifying the inputs and outputs of the system as well as what information a system must retain and manage. The input and outputs are data or flows and may be direct flows or may be carried via service requests or responses. Early in the system engineering process, the information captured about these elements is logical. The definition of a logical schema is provided below, along with a set of related definitions.

Definitions

Data Schema: a data or type model of a specific problem domain that includes blocks, value properties, value types, dimensions, units, their relations, and other relevant aspects collectively known as *metadata*. This model includes a type model consisting of a set of value types, units, and dimensions, and a usage model showing the blocks and value properties that use the type model.

Logical Schema: a data schema expressed independently from its ultimate implementation, storage, or transmission means.

Value Property: a property model element that can hold values. Also known as a variable or attribute (UML).

Value Type: specify value sets applied to value properties, message arguments, or other parameters that may carry values. The system predefines primitive scalar types including number, integer, real, complex, Boolean, and string. These base types may have additional properties or constraints, specified as metadata.

Metadata: literally “data about data,” this term refers to ancillary properties or constraints on data, including:

- **Extent** – the set of values of an underlying base value type that are allowed. This can be specified as:
 - a subrange, as in 0 .. 10
 - a low-value and high-value pair, as in “low value = -1, high value = 1”
 - an enumerated list of acceptable values
 - a specification of prohibited values
 - the specification of a rule or constraint from which valid values can be determined
- **Precision** – the degree exactness of specified values; for scalar numeric values, this is often denoted as the “number of significant digits”
- **Accuracy** – the degree of conformance to an actual value, often expressed as \pm <value>, as in “ ± 0.25 ”. Accuracy is generally applied to an output or outcome
- **Fidelity** – the degree of exactness with which value may be specified. Fidelity is generally applied to an input value
- **Latency** – how long after a value property occurs is the value representation updated
- **Availability** – a measure of what percentage of the system lifecycle value is actually accessible

Note that these properties are sometimes not properties of the value type but of the value property specified by that value type. In any case, in SysML, these properties are often expressed in tags, metadata added to describe model elements:

Quantity Kind: specifies the *kind* of value (its dimensionality). Examples include *length*, *weight*, *pressure*, and *color*. Also known as **Dimension** in SysML 1.

Unit: specifies a standard against which values in a dimension may be directly compared. Examples include *meters*, *kilograms*, *kilopascals*, and *RGB* units. The SysML provides a model library of SI units that are directly available for use in models. However, it is not uncommon to define your own as needed.

Recommendation: Each value property should be typed by a unit, unless it is unitless, in which case, it should be typed by a defined value type.

Schematically, these definitions are shown in *Figure 2.91* in the data schema metamodel:

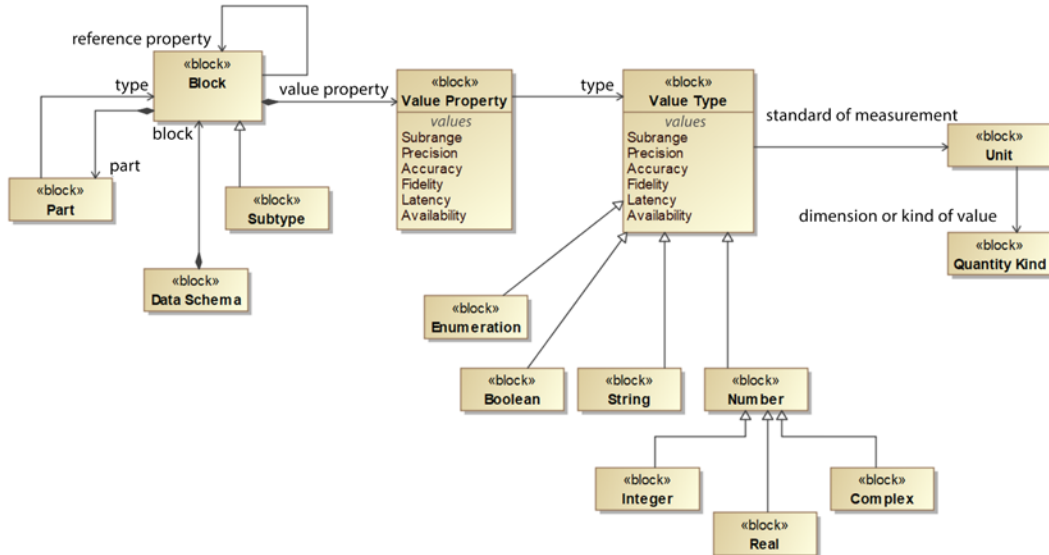


Figure 2.91: Data schema metamodel

Although this is called the **data schema**, it is really an information schema as it applies to elements that are not data per se, such as physical flows. In this book, we will use the common term **data schema** to apply to flows as well.

Beyond the underlying type model of the schema, described above, the blocks and their value properties and the relations among them constitute the remainder of the data schema. These relations are the standard SysML relations: association, aggregation, composition, generalization, and dependency.

Example

So, what does a diagram showing a logical data schema look like?

Typically, a data schema is visualized within a block definition diagram and shows the data elements and relevant properties. Consider an aircraft navigation system that must account for “own craft” position, velocity, acceleration, flight plans, attitude, and so on. See *Figure 2.92*:

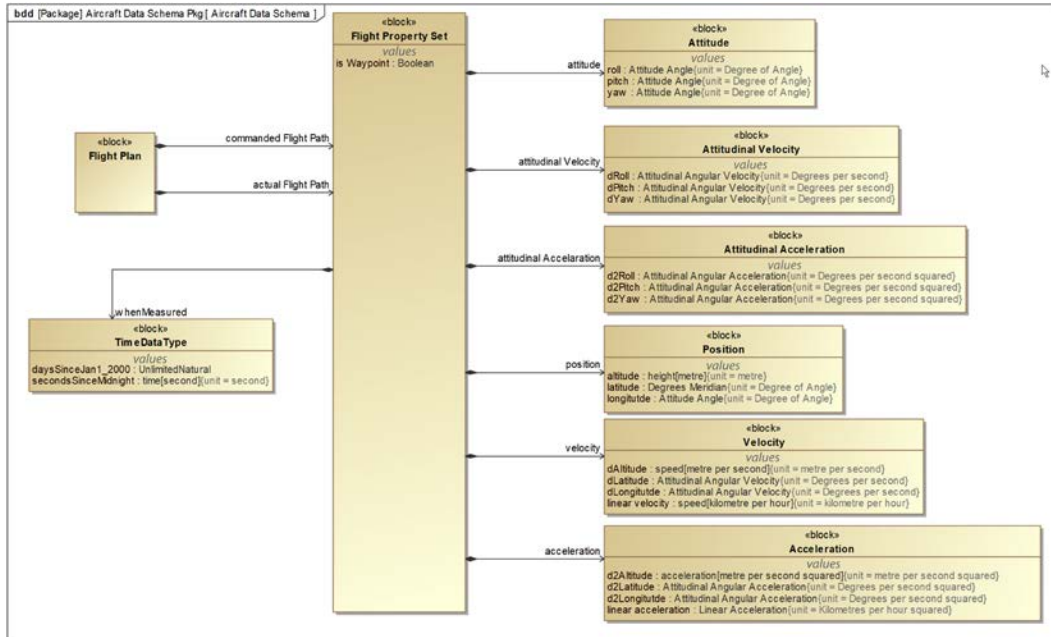


Figure 2.92: Data schema for flight property set

You can see in the figure, a **Flight Property Set** contains **Airframe_Position**, **Airframe_Velocity**, **Airframe_Acceleration**, and so on. These composed blocks contain value properties that detail their value properties; in the case of **Airframe_Position**, these are **altitude**, **latitude**, and **longitude**. **Altitude** is expressed in the unit **Meters** (defined in the Cameo SysML type library) while **latitude** and **longitude** are defined in terms of the unit **Meridian_Degrees**, which is not in the SysML model library (and so is defined in the model).

On the left of the diagram, you can see that a **Flight Plan** contains multiple **Flight Property Sets** identifying planned waypoints along the **commanded flight path**. These **Flight Property Sets** may be actual current information (denoted with the **actual Flight Path** role end) or commanded (denoted with the **commanded Flight Path** role end). The latter forms a list of commanded flight property sets and so stores the set of commanded waypoints.

Figure 2.93 shows the scalar types, units, and quantity kind used in Figure 2-92:

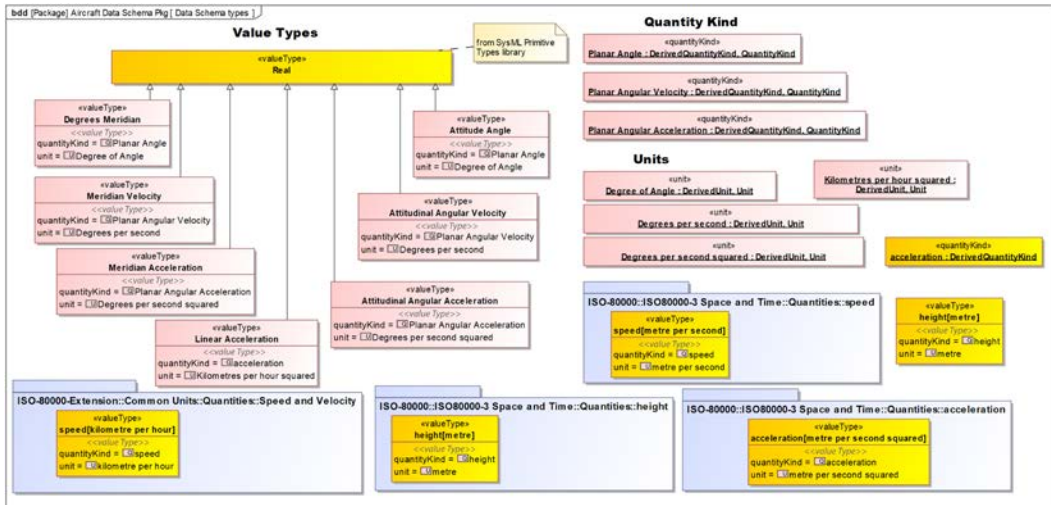


Figure 2.93: Types, units, and quantity kinds

This is a serviceable data schema, however, details about the data (metadata) are missing. What are the allowed ranges of values (known as a type's extent); what is acceptable precision (degree of detail in representation), accuracy (conformance to a value's specification), and fidelity (degree of precision of inputs)? Figure 2.94 shows an example of applying a stereotype with tags that define this metadata. In this case, «qualified» adds **low-value** and **high-value** tags for specifying an extent, as well as **accuracy** and **fidelity** tags:

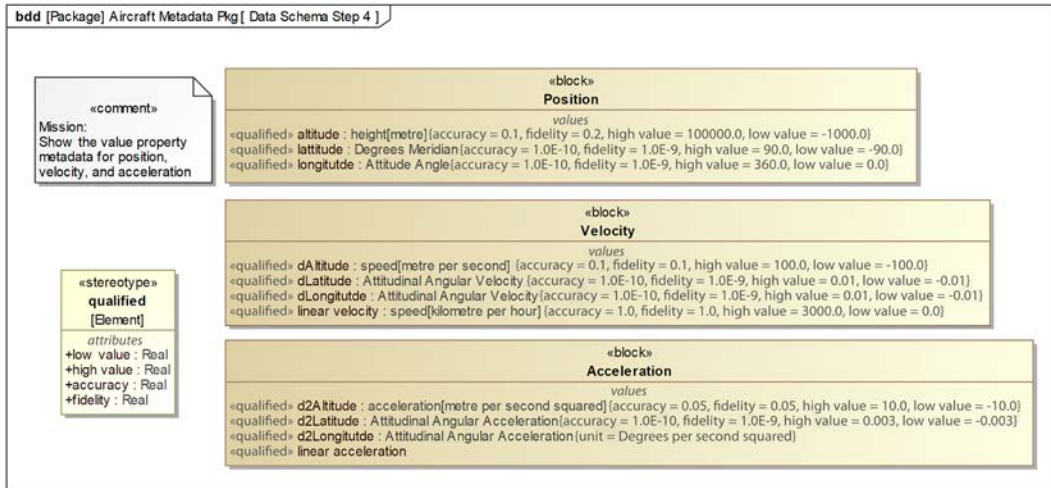


Figure 2.94: Specifying metadata with tags

Purpose

The purpose of the logical data schema is to understand the information received, stored, manipulated, and transmitted by a system. In the context of the capture of system specification, it is to understand and characterize data and flows that cross the system boundary to conceptually solidify the interfaces a system provides or requires.

Inputs and preconditions

The precondition is that a use case and set of associated actors have been identified or that structural elements (blocks) have been identified in an architecture or design.

Outputs and postconditions

The output is a set of units, dimensions, and types (the type model), and value properties that they specify along with the relations among the value types and blocks that own them (the usage model).

How to do it

The workflow has a number of small, well-defined steps. The overall workflow for this recipe is shown in *Figure 2.95*:

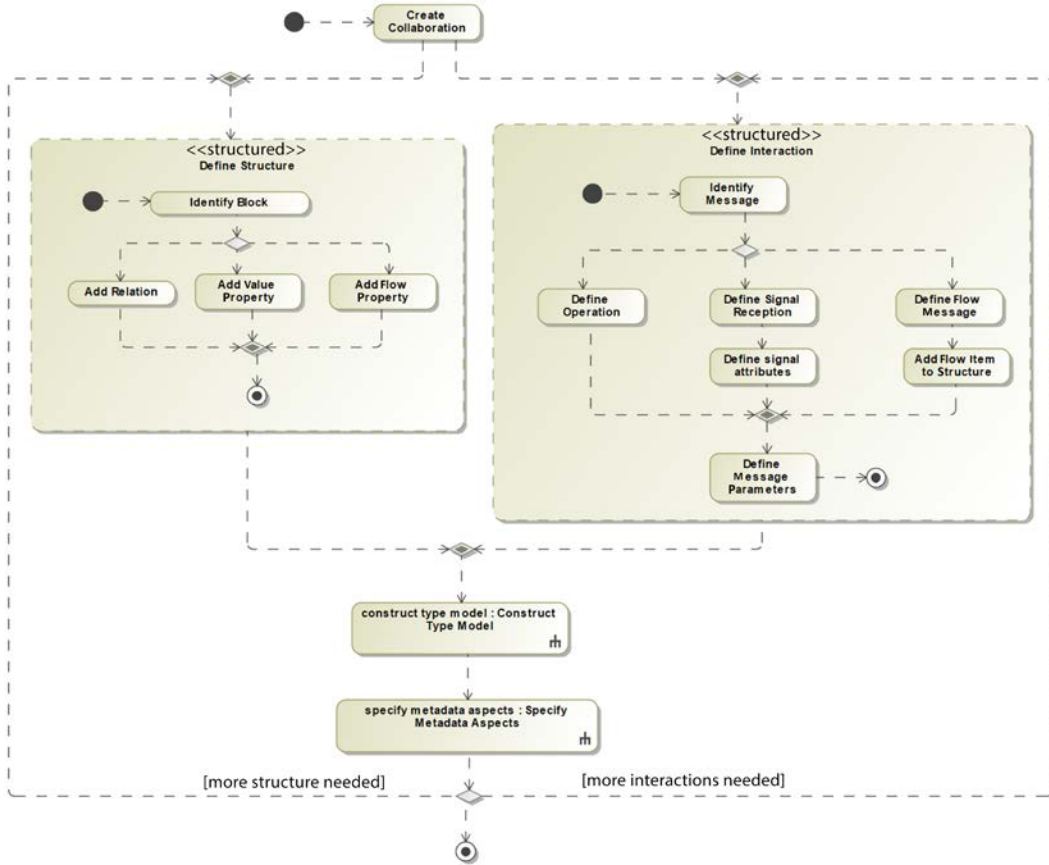


Figure 2.95: Creating the logical data schema

This workflow contains two elaborated call behaviors: **Construct Type Model** and **Specify Metadata Aspects**. The first of these is detailed in *Figure 2.96* and the latter in *Figure 2.97*:

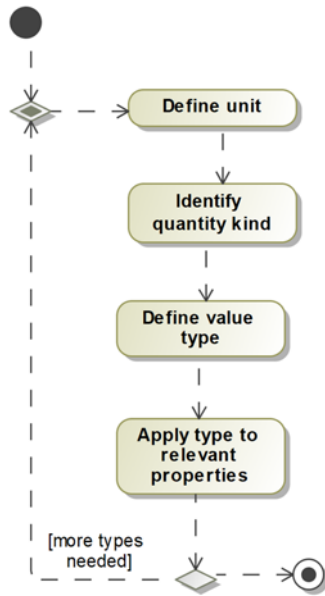


Figure 2.96: Construct the type model

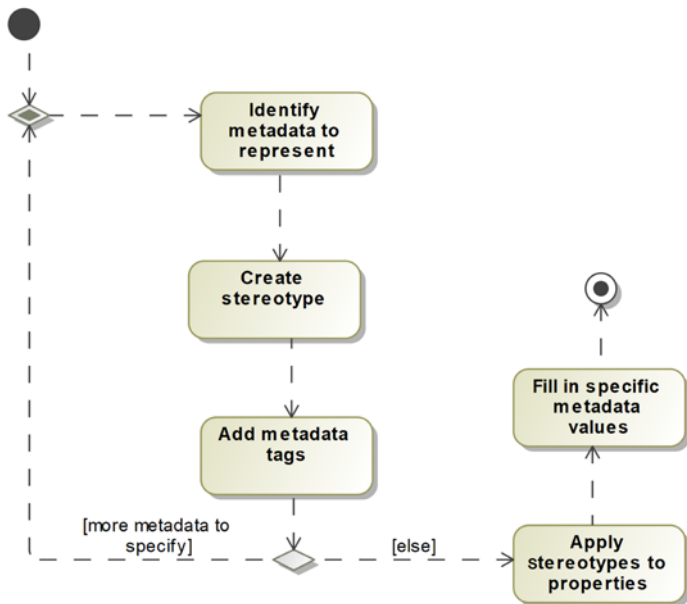


Figure 2.97: Specify metadata aspects

Create collaboration

This task creates the collaboration of elements that will own or use the elements defined within the data schema. This provides the context in which the types may be considered. In the case of system specification, this purpose is served by defining the use case and its related actors, or by the execution context of block stand-ins for those elements. In a design context, it is generally some set of design elements that relate to some larger-scale purpose, such as showing an architectural aspect or realizing a use case.

Define structure

This structured activity adds blocks and other elements to the collaboration, detailed in the steps *Identify Block*, *Add Relation*, *Identify Value Properties*, and *Add Flow Property*.

Identify block

These are the basic structural element of the collaboration, although rarely value properties may be created without an owning block.

Add relation

These relations link the structural element together allowing them to send messages to support the necessary interactions. They might be associations to add reference properties, compositions to add part properties, or generalizations to define specialization relations.

Identify value property

This step identified the data property features of the blocks.

Identify flow property

This step identified the flow property features of the blocks. Remember, a flow property is nothing more than a value property that flows across a block's encapsulation boundary without the explicit use of services.

Define interaction

The interaction consists of a set of message exchanges among elements in the collaboration. This is most often shown as sequence diagrams. This structured activity defines an internal workflow.

Identify message

Messages are the primitive elements of interaction. These may be synchronous (such as function calls), asynchronous (as in asynchronous signal receptions), or flows (represented as flow items on a synchronous or asynchronous message). A single interaction contains a set of ordered messages.

Define operation

For synchronous call operation messages, the block receiving the message must define an operation to be invoked. The step includes the operation signature, as defined by its parameters, and any return values.

Define signal reception

For asynchronous signal receptions, a signal must be defined.

Define signal attributes

Signals can carry information, just as operation calls can. The signal attributes specify the information passed along with the signal.

Define flow message

As we have seen in previous recipes, flows can be sent between elements of collaboration, as defined with flow properties and item flows. These are represented in the scenarios as flow messages – that is, messages with flow items.

Define flow item to structure

For correctness and consistency, the flow items must correspond to flow properties defined in the collaborating elements.

Add message parameters

Most messages, whether synchronous or asynchronous, carry information in the form of parameters (sometimes called *arguments*). The types of this data must be specified in the data model.

Construct type model

Once a datum is identified, it must be typed. This call behavior is detailed in the following steps.

Define unit

Most data relies on units for proper functioning and too often, units are implied rather than explicitly specified. This step references existing units or creates the underlying unit and then uses it to type the relevant value properties. The SysML defines a non-normative extension to include a model library of SI units. Cameo, the tool being used here, has a rich set of units and quantity kinds as defined by the SI standard. Not all types are present, and it is common to add additional units and quantity kinds, as needed.

Define quantity kind

Most units rely on a quantity kind (aka *dimension*). For example, the unit **Meter** might represent the quantity kind of **distance**, **length**, **height**, or **width**. Quantity kinds have many different units available. **Length**, for example, can be expressed in units of **cm**, **inches**, **feet**, **yards**, **meters**, **miles**, **kilometers**, and so on.

Define value types

The underlying value type is expressed in the action language for the model. This might be C, C++, Java, Ada, or any common programming or data language. OMG also defined an abstract action language called **Action Language for Foundational (ALF) UML**, which may be used for this purpose. See <https://www.omg.org/spec/ALF/About-ALF/> for more information. This book uses Groovy as the action language because it is supported in the Cameo tool, but there are equally valid alternatives.

Apply type to relevant properties

This step applies the defined type to the relevant value and flow properties owned by the elements of the collaboration.

Specify metadata aspects

It is almost always inadequate to just specify the value type from the underlying action language. There are other properties of considerable interest. As described earlier in this section, they include extent, precision, latency, and availability. There are many additional metadata properties that may be of concern, including some that may be domain-specific. This call behavior action defines a simple workflow for the identification and application of metadata to the value and flow properties.

Identify metadata of interest

The first step in this nested workflow is to identify which metadata properties should be represented. These will be properties of the data, such as the **extent**, **precision**, **fidelity**, **latency**, and so on.

Create stereotype

The recommendation here is to use stereotypes with tags that define the metadata properties of interest.

Add metadata tags

Add tags to represent the metadata of interest. Not all value or flow properties to which the stereotype is applied must use or provide values for all tags.

Apply stereotypes to properties

Once the stereotypes are defined, the value and flow properties can be stereotyped so that they can own the metadata. Sometimes the stereotypes will be applied to value types or units.

Fill in specific metadata properties

Different value types will provide different values for the metadata tags. For example, consider a stereotype **Defined Extent** that has tags **Low Value** (specifying the lowest acceptable value for the property) and **High Value** (specifying the largest acceptable value). A block **Hand** might have a value property **fingerCount** for which the lowest value is 0 and the highest value is 10. Another block – say, **Car** – might have a value type **numberOfDoors** with a lowest possible value of 2 and a highest possible value of 5.

Example

This example will use the **Measure Performance Metrics** use case. The earlier recipe *Model-Based Threat Analysis* used this use case to discuss modeling cybersecurity. We will use it to model the logical data schema. For the most part, the data of interest is the performance data itself, although the threat model identified some additional security-relevant data that can be modeled as well.

Create collaboration

The use case diagram in *Figure 2.73* provides the context for the data schema, but usually, the corresponding block definition and internal block diagrams of the execution context are used.

That diagram is shown in *Figure 2.98*:

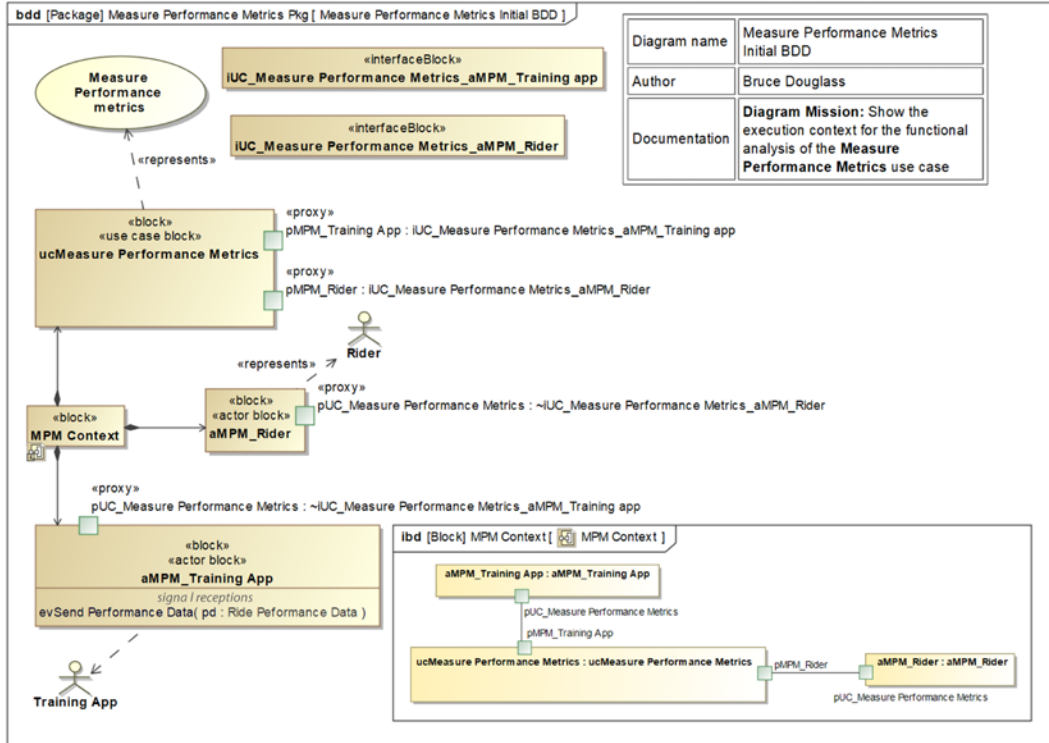


Figure 2.98: Measure performance metrics execution context

Define structure

This task is mostly done by defining the execution context, shown in *Figure 2.98*. In this case, the structure is pretty simple.

Identify blocks

As a part of defining the structure, we identified the primary functional blocks in the previous figure. But now we need to begin thinking about the data elements as blocks and value types. *Figure 2.99* shows a first cut at the likely blocks. Note that we don't need to represent the data schema for the actors because *we don't care*. We are not designing the actors since they are, by definition, out of our scope of concern. We must, on the other hand, define the data schema for information passed between the actors and the system:

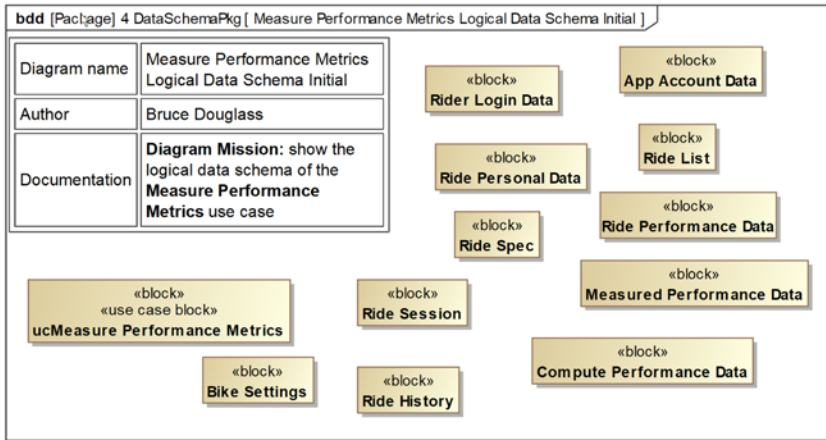


Figure 2.99: Blocks for measure performance data schema-initial

Add relations

The instances of the core functional blocks are shown in *Figure 2.98*. Relations of the data elements to the use case block are shown in *Figure 2.100*. This is the data that the use case block knows (owns) or uses:

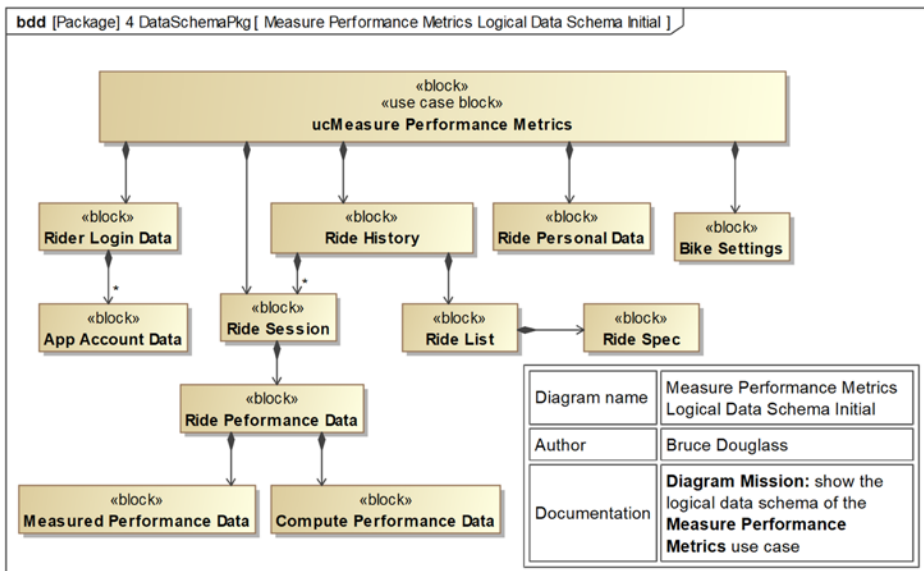


Figure 2.100: Data schema with relations

Identify value properties

The blocks provide owners with the actual data of interest, which is held in the value properties.

Figure 2.101 shows the blocks populated with value properties relevant to the use case:

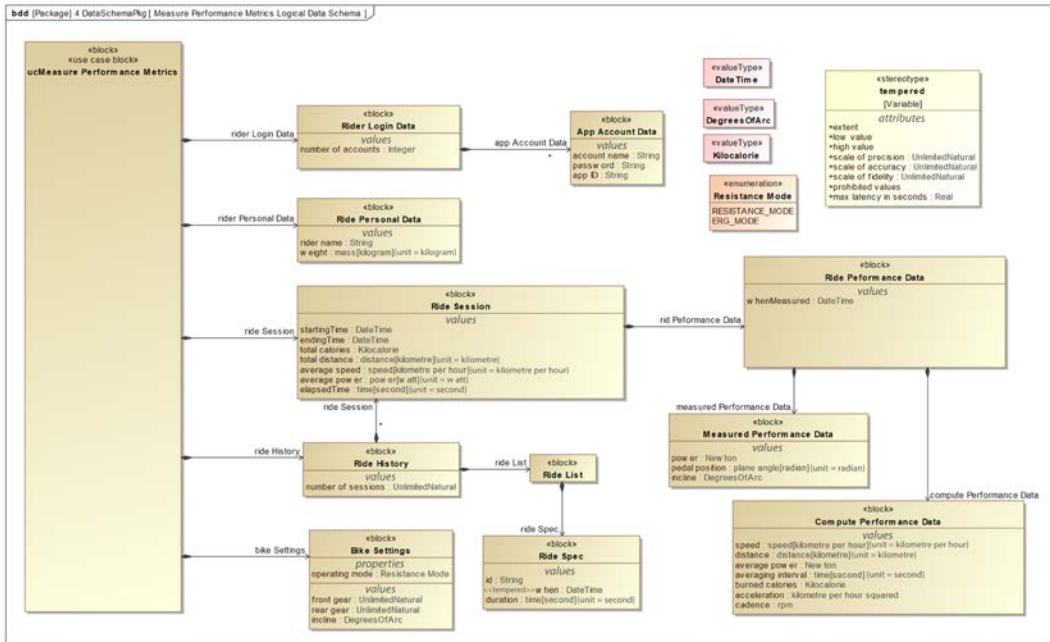


Figure 2.101: Data schema value properties

Define interaction

Another way to find data elements to structure is to look at the messaging; this is particularly relevant for use case and functional analysis since the data on which we focus during this analysis is the data that is sent or received. This is represented as a structured action in Figure 2.95. We will employ all its contained actions together.

The first interaction we'll look at is for uploading real-time ride metrics during a ride. This is shown in Figure 2.102:

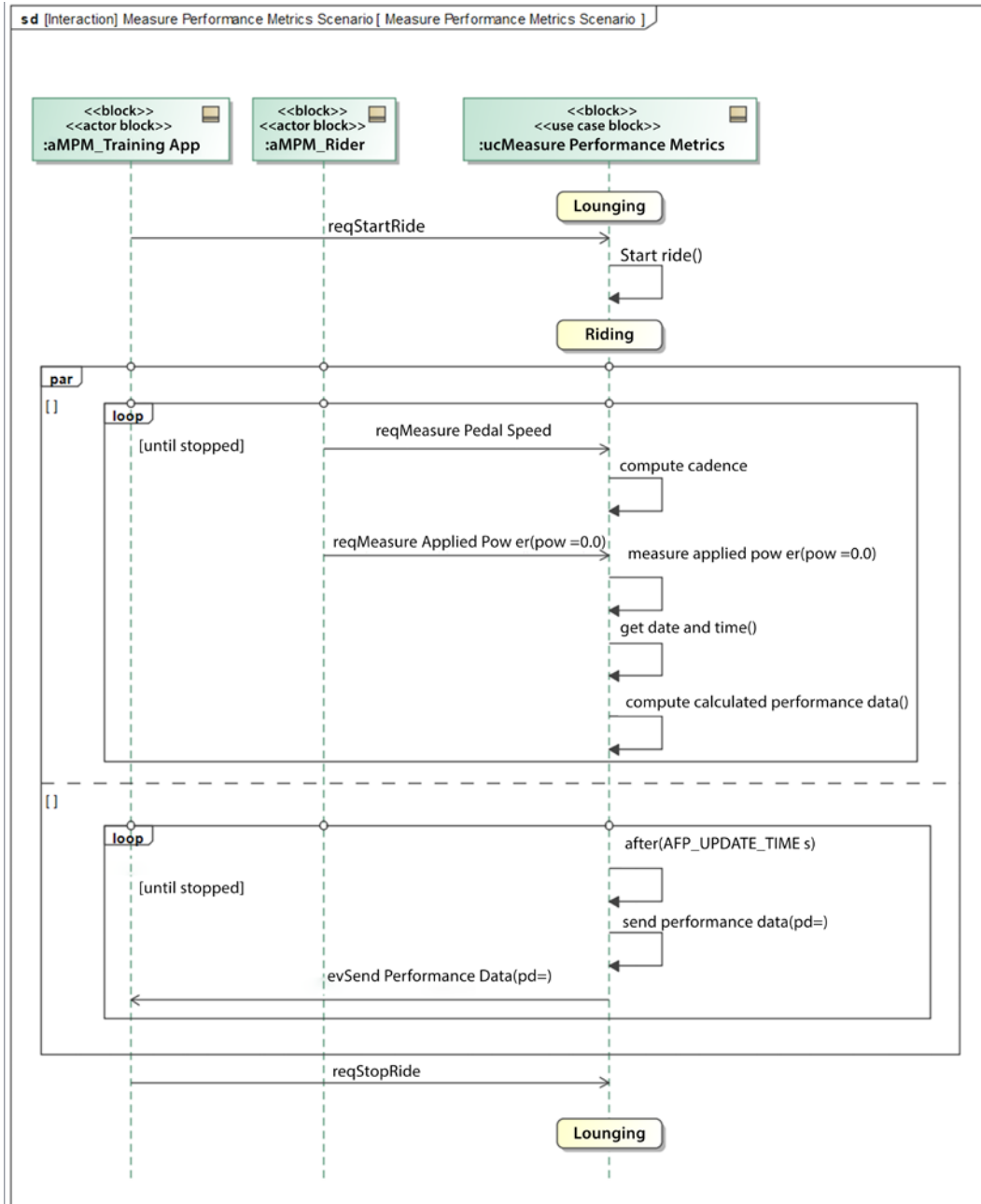


Figure 2.102: Real-time ride metrics scenario

The second interaction is for uploading an entire stored ride to the app. This is in *Figure 2.103*:

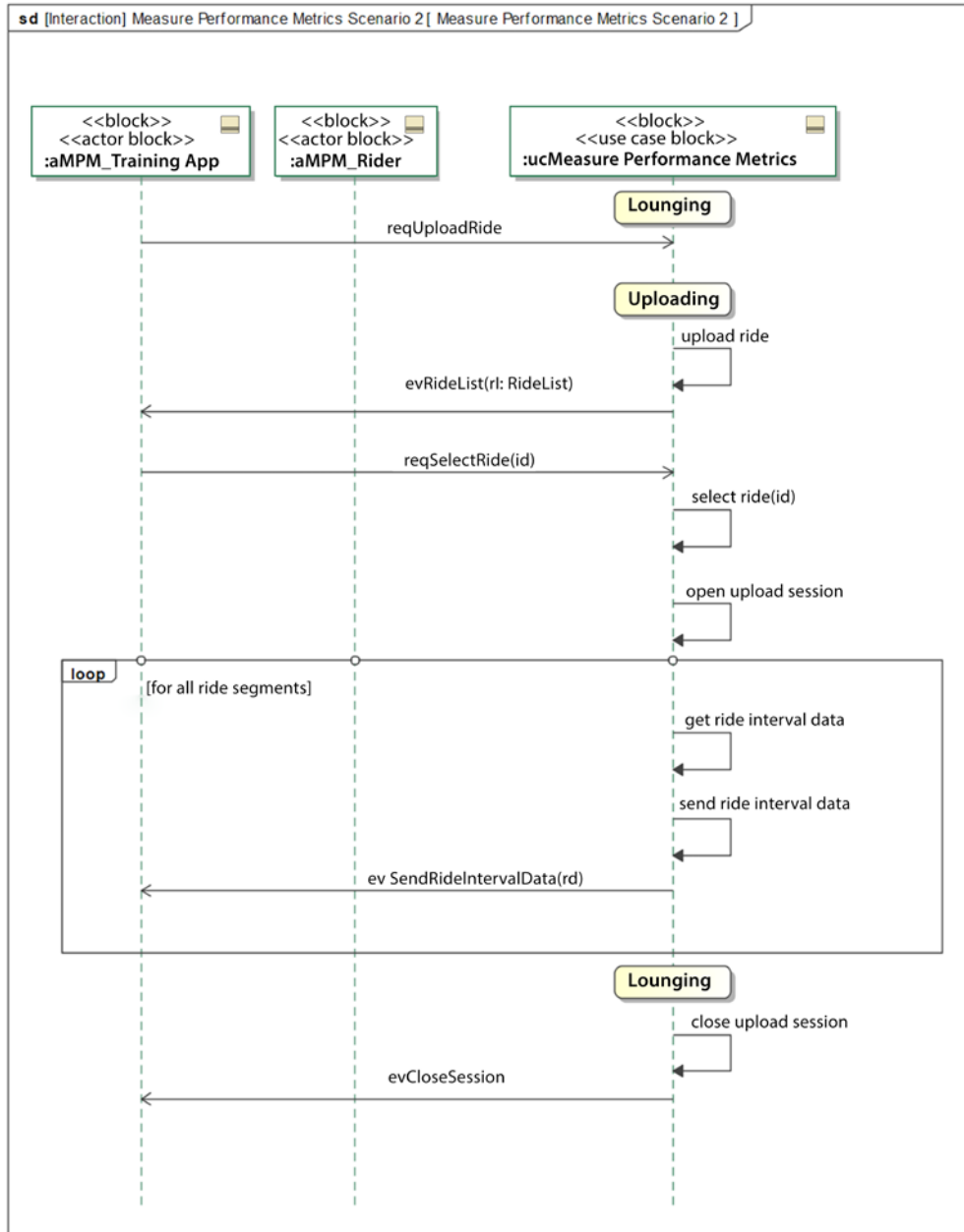


Figure 2.103: Upload saved ride

Note that these are just two of many scenarios for the use case, as these do not consider concerns such as dropped messages, reconnecting sessions, and other rainy-day situations. However, this is adequate for our discussion.

In this step, when we draw a “message to self,” this will be realized as an operation in the owning block; received signals will be realized as a signal reception, and flows are realized as flow properties.

Construct type model

Figure 2.101 goes a long way for the definition of the type model. The blocks define the structured data elements, but at the value property level, there is still work to be done. The underlying value types must be identified, their units and dimensions specified, and constraints placed on their extent and precision.

Most, but not all, value properties must be defined in terms of **units** and **quantity kinds**.

Define units

Identify quantity kind

It is common for engineers to just reference base types – **Integer**, **Real**, and so on – to type value properties, but this can lead to avoidable design errors. This is because value types may not be directly comparable, such as when **distanceA** and **distanceB** are both typed as **Real** but in one case are captured in **Kilometers** and in the other, **Miles**. Further, one cannot reason about the extent of a type (the permitted set of values) without specified units. For this reason, we recommend – and will use here – unit definitions to disambiguate the values we’re specifying.

The SI Units model library of the SysML specification is an optional compliance point for the standard, and Cameo does an excellent job of providing a set of units and quantity kinds. In this model, we will reference some that exist in that library and create those that do not.

Figure 2.101 uses a number of special units for value properties and operation arguments, including:

- Degrees of Arc
- Newton
- DateTime
- Kilometers per Hour
- Kilometers per Hour Squared
- Kilocalorie

- RPM
- Kilometer
- Resistance Mode

Some of these exist in the SI library and are included in the Cameo tool, such as **Newton** and **Kilometer** (although Cameo uses the English spelling of the latter, **Kilometre**). Others could have been expected to be included, but are not, such as **Degrees of Arc**, **Kilocalorie**, and **RPM**. Still others are application-specific, including **Resistance Mode**.

Degrees of Arc are a measure of angular displacement and are used for the cycling incline. **RPM** is a measure of rotational velocity used for pedaling cadence. **DateTime** is a measure of when data was measured. **Kilometer** is a measure of linear distance, while **Kilometers per hour** is a measure of speed, and **Kilometers Per Hour Squared** is a measure of acceleration. **Kilocalorie** is a measure of energy used to represent the Rider's energy output. In our model, we will define all these as units. They will be defined in terms of their dimension in the next section.

Cameo uses a naming convention for value types in its SI library. A value type is given the name of the quantity kind + [+ unit +], as in **speed[kilometre per hour]**. This is a useful naming convention telling the modeler about the type, quantity kind, and units in a simple, consistent way.

Figure 2.104 shows the value types, units, and dimensions defined for this logical data schema. **Resistance Mode** is an enumeration value type with two possible values, **RESISTANCE_MODE** and **ERG_MODE**:

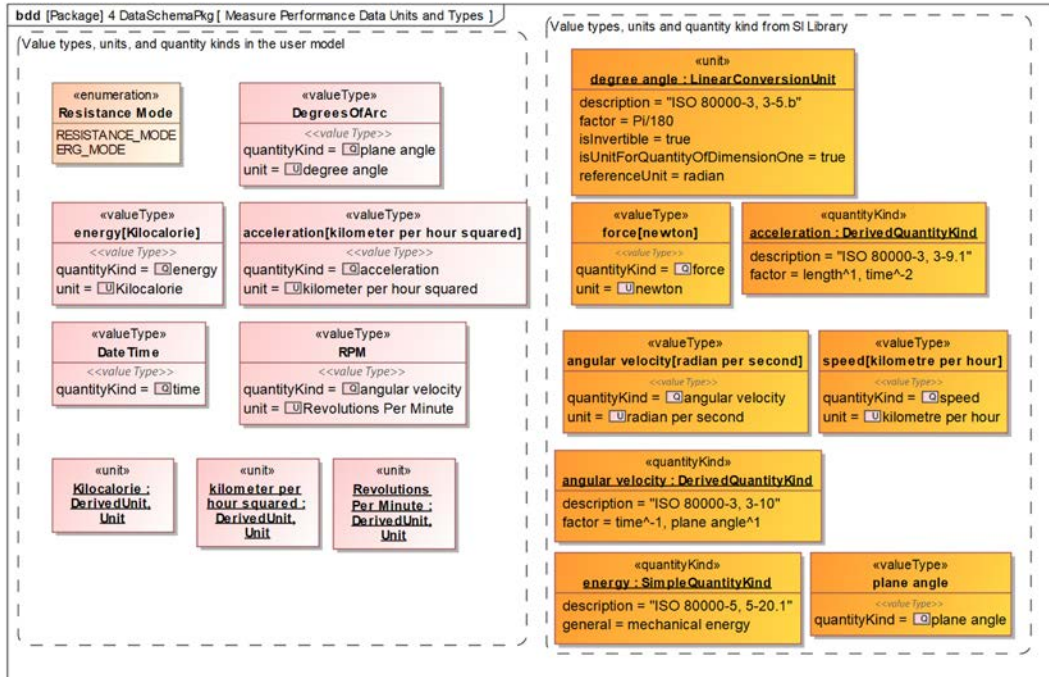


Figure 2.104: Value types, units, and quantity kinds

Define value type

Once the units and quantity kinds are defined, we can define the value types. Once again, the Cameo SI Types library provides a large set of values types. The naming convention for value types is **quantity kind + [+ unit]**, so you get a value type such as **power[watt]**. Knowing this allows you to find these value types in the types library. We will mostly follow that naming convention here. The value type **acceleration[kilometers per hour squared]** in *Figure 2.104* follows that convention, although **Degrees of Arc** does not (to demonstrate an alternative).

Apply type to relevant properties

Now that we have the value types, we can use them to type the value properties in our model.

Define value type and properties for metadata

The last thing we must do is specify the relevant value type properties. In the logical data schema, this means specifying the extent and precision of the values. This can be done at the unit/value type level; in this case, the properties apply to all values of that unit or type. These properties can also be applied at the value property level, in which case the scope of the specification is limited to the specific values but not to other values of the same unit or type.

One way to specify these properties is to specify them as SysML tags within a stereotype, apply the stereotype to the relevant model elements, and then elaborate on the specific values. To that end, we will create a stereotype, «tempered». This stereotype applies to object nodes, types, parameters, and variables (and therefore, also to pins, value types, units, and value properties) in the SysML metamodel.

The stereotype provides three ways to specify the extent. The first is the **extent** tag, which is a string in which the engineer can specify a range or list of values, such as **[0.00 .. 0.99]** or **0.1, 0.2, 0.4, 0.8, 1.0**. Alternatively, for a continuous range, the tags **lowValue** and **highValue**, both of type **Real**, can serve as well; in the previous example, you can set **lowValue** to **0.0** and the **highValue** to **0.99**. Lastly, you can provide a range or list of prohibited values in the **prohibitedValues** tag, such as **-1, 0**.

The stereotype also provides three means for specifying scale. The **scaleOfPrecision** tag, of type **integer**, allows you to define the number of significant digits for the value or type. Numeric values are also further refined with **fidelity** (precision of an input) and **accuracy** (degree of conformance to the specified value). Both fidelity and accuracy are usually specified as a \pm value to give a range, as in ± 2 mm.



Precision technically refers to the number of significant digits in a number, while **scale** is the number of significant digits to the right of the decimal point. The number **123.45** has a precision of 5, but a scale of 2. People usually speak of precision while meaning scale.

Another stereotype tag is **maxLatency**, using a **time[second]** value type from the SI library to allow specification of the maximum allowable latency of the data. Other metadata can be added to the stereotype as needed for your system specification.

This level of detail in the specification of quantities is important for downstream design. Requiring 2 digits of scale is far different than requiring 6 and drives the selection of hardware and algorithms with effects on system cost and required effort. These stereotypes can be applied at different levels of specification: the value property, the value type, or the unit:

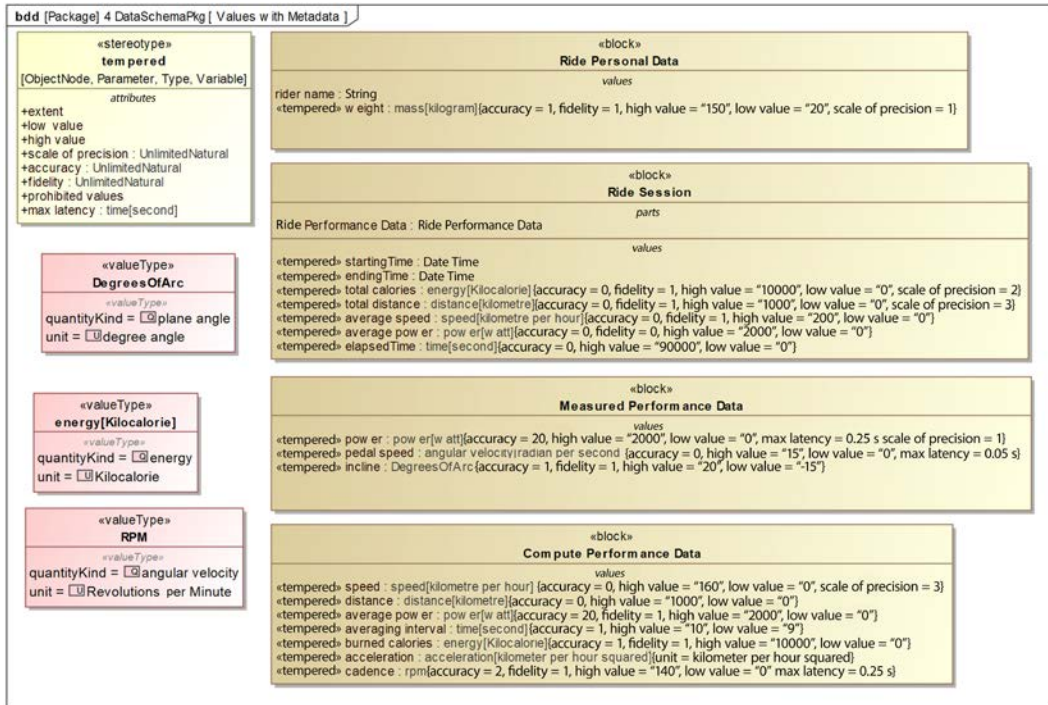


Figure 2.105: Measure performance metrics metadata

Figure 2.105 shows the application of the **tempered** stereotype to a number of value properties of relevant data blocks in the model. Different metadata may be relevant to different value properties; for example, latency is not a concern for **Ride Session::Ride Time** but is relevant for more real-time data, such as **Compute Performance Data::acceleration**.

And there you have it: a logical data schema for the values and flows specified as a part of the **Measure Performance Metrics** use case. These, along with data schema from other use cases, will be merged into the architecture in the architecture design work.

In the next chapter, we move to the topic of design. This, as we will see, is a topic rich with recipes.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>



3

Developing System Architectures

Recipes in this chapter

- Architectural trade studies
- Architectural merge
- Pattern-driven architecture
- Subsystem and component architecture
- Architectural allocation
- Creating subsystem interfaces from use case scenarios
- Specializing a reference architecture

System analysis pays attention to the required properties of a system (such as its functionality), while system design focuses on how to implement a system that implements those needs effectively. Many different designs that can realize the same functionality; system engineers must select from among the possible designs based on how well they optimize crucial system properties. The degree of optimization is determined by examining with **Measures of Effectiveness (MoE)** applied to the design. Design is all about optimization, and architecture is no different. Architecture is the integration of high-level design concerns that organize and orchestrate the overall structure and behavior of the system.

Design exists at (at least) three levels of abstraction. The highest level – the focus of this chapter – is architectural design. *Architectural design* makes choices that optimize the overall system properties at a system-wide level. The next step down, known as *collaboration design*, seeks to optimize collaborations of small design elements that collectively perform important system behaviors. Collaborative design is generally an order of magnitude smaller in scope than architectural design. Finally, *detailed design* individually optimizes those small design elements in terms of their structure or behavior.

Five critical views of architecture

The Harmony process defines six critical views of architecture, as shown in *Figure 3.1*. Each view focuses on a different aspect of the largest scale optimization concerns of the system:

- **Subsystem/Component Architecture** is about the identification of subsystems, the allocation of responsibilities to the subsystems, and the specification of their interfaces.
- **Distribution Architecture** selects the means by which distributed parts of the system interact, including middleware and communication protocols; this includes, but is not limited to, network architecture.
- **Concurrency and Resource Architecture** details the set of concurrency regions (threads and processes), how semantic elements map into those concurrency regions, how they are scheduled, and how they effectively share and manage shared resources.
- **Data Architecture** focuses on how data is managed. It includes technical means and policies for data storage, backup, retrieval, and synchronization, as well as the overall data schema itself. Beyond that, this view includes the policies for ensuring data correctness and availability.
- **Dependability Architecture** refers to large-scale design decisions that govern the ability of stakeholders to depend on the system. This is subdivided into the Four Pillars of Dependability:
 - **Safety Architecture** – the large-scale mechanisms by which the system ensures the risk of harm is acceptable
 - **Reliability Architecture** – the system-wide decisions that manage the availability of services
 - **Security Architecture** – the important design decisions that control how the system avoids, identifies, and manages attacks
 - **Privacy Architecture** – how the system protects information from spillage

- **Deployment architecture** is the allocation of responsibilities to different engineering facets, such as software, electronics, and mechanical design concerns.

This chapter focuses on the development and verification of systems architecture with some key recipes, including the following:

- Selecting from architectural alternatives with trade studies
- Merging different use cases analyses together into a system architecture
- Application of architectural design patterns
- Creating the subsystem and component architecture
- Allocation of system properties into the subsystem architecture
- Defining system interfaces

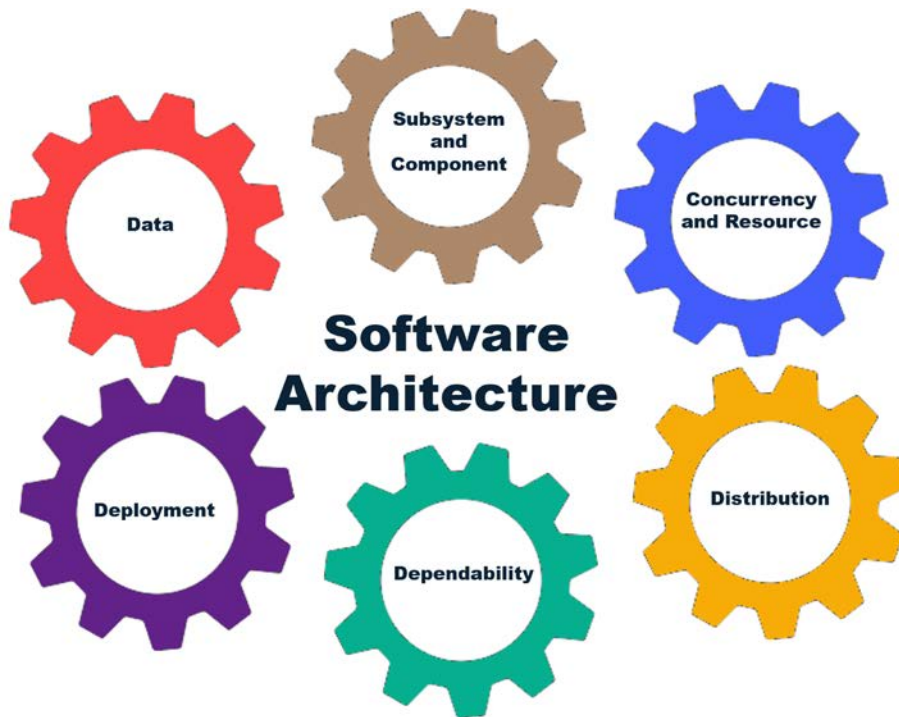


Figure 3.1: Six critical views of architecture

General architectural guidelines

As previously shown in *Figure 1.27*, good architecture:

- Is architected early

- Evolves frequently
- Is as simple as possible (but no simpler than that)
- Is based on patterns
- Integrates into project planning via Technical Work Items
- Optimizes important system properties
- Is written down (specifically, modeled)
- Is kept current

The *Architecture 0* recipe in *Chapter 1, Basics of Agile Systems Modeling*, concentrated on creating an early model of architecture so that more detailed engineering work had a structural context and to establish the feasibility of desired system capabilities early. Remember – from Chapter 1 – in the context of an agile approach, engineers establish an architectural roadmap in which architectural features that meet requirements are added incrementally over time. The recipes in this chapter will therefore be performed multiple times during the development of a product. Thus, the expectation is that architecture evolves throughout the development work as more functionality is added to the evolving system design.

This chapter provides some important recipes for architectural definition, whether it is done as Big-Design-Up-Front or in an incremental agile process.

Architectural trade studies

Trade studies are specifically concerned with the merit-based selection of approach or technology based on important concerns specific to the system development, system environment, or stakeholder needs. At a very fundamental level, trade studies are about making design choices to optimize important properties of the system at the expense of properties deemed less critical. To effectively perform trade studies, it is important to identify the things that can be optimized, the aspects subject to optimization, the MoE, and a set of alternatives to be evaluated.

Purpose

The purpose of performing an architectural trade study is to select an optimal design solution from a set of alternatives.

Inputs and preconditions

The inputs to this recipe are:

1. Functionality of concern, scoped as a set of requirements and/or use cases

- Design options capable of achieving that functionality

Outputs and postconditions

The primary output of this recipe is an evaluation of alternatives, generally with a single technical selection identified as the recommended solution. This output is often formatted as a **Decision Analysis Matrix**. This matrix is normally formatted something like *Table 3.1*:

Candidate Solutions	Solution Criteria										Weighted Total
	Power Consumption (W1= 0.3)		Recurring Cost (W2 = 0.2)		Robustness (W3 = 0.15)		Development Cost (W4 = 0.1)		Security (W5 = 0.25)		
	MoE	Score	MoE	Score	MoE	Score	MoE	Score	MoE	Score	
Gigabit Ethernet Bus	2	0.6	2.7	0.54	4	0.6	8	0.8	4	1.0	3.54
1553 Bus	3	0.9	4	0.8	10	1.5	1.5	0.15	6	1.5	4.85
CAN Bus	6	1.8	8	1.6	7	1.05	3	.3	1	0.25	5.0

Table 3.1: Example Decision Analysis Matrix

In the table, the middle columns show the optimization criteria; in this case, **Power Consumption**, **Recurring Cost**, **Robustness**, **Development Cost**, and **Security**. Each is shown with a relative weight (**W**). This weighting factor reflects the relative importance of that criterion with respect to the others. It is common, as done in this example, for the weights to be normalized so that they sum to 1.00. It is also common to normalize the MoE values as well, although that was not done in this example.

Each of those columns is divided into two. The first is the MoE value for a particular solution, followed by the MoE value times the weighting factor for that criterion. This is the weighted *Score* value in the table for that criterion. Coming up with a good MoE is a key factor in having a useful outcome for the trade study.

The last three rows in the example are different technical solutions to be evaluated. In this case, the trade study compares **Gigabit Ethernet Bus**, **1553 Avionics Bus**, and the **Control Area Network (CAN) Bus**.

The last column is the weight score for each of the solutions, which is simply the sum of the weighted scores for the solution identified in that row. The matrix is set up so that the highest value here “wins,” that is, it is determined to be the best overall solution based on the MoE and their criticalities.

How to do it

Figure 3.2 shows the basic workflow for performing an architectural trade study. This approach is useful when you can a relatively small set of alternatives (known as the “trade space”) in the evaluation. Other techniques are more appropriate when you have a very large trade space. Note that we will be using SysML parametric diagrams in a specific way to perform the trade study, and these specific techniques will be reflected in the recipe step details.

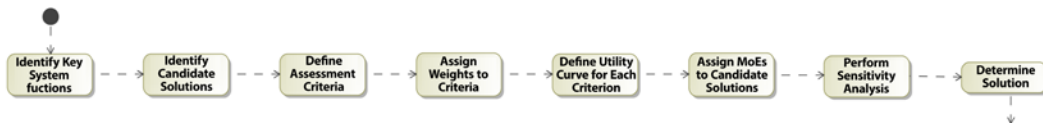


Figure 3.2: Perform trade study

Identify key system functions

Key System Functions are system functions that are important, architectural in scope, and subject to optimization in alternative ways. System functions that are neither architectural nor optimizable in alternative ways need not be considered in this recipe. To be “optimizable in alternative ways” means:

- At least one criterion of optimization can be applied to this system function
- There is more than one reasonable technical means to optimize the system function

An example of a system function would be to provide motive force for a robot arm. This could be optimized against different criteria, such as lifecycle cost, reliability, responsiveness, accuracy, or weight.

Identify candidate solutions

Candidate solutions are the technical means to achieve the system function. In the case of providing motive force for a robot arm, technical means include pneumatics, hydraulics, and electric motors. All these solutions have benefits and costs that must be considered in terms of the system context, related aspects of the system design, and stakeholder needs.

This step is often performed in two stages. First, identify all reasonable, potential technical solutions. Second, trim the list to only those that are truly options for consideration. It is not uncommon for several potential solutions to be immediately dismissed because of technical maturity issues, availability, cost, or other feasibility reasons. At the end of the step, there is usually a shortlist of three to ten potential solutions for evaluation.

In SysML, we will model the key system function as a block and will add the assessment criteria (in the next step) as value properties. The different candidate solutions will be then modeled as instance specifications of this block with different values assigned to the value property slots. Although it's not strictly necessary, we will name the block for each trade study with the name of the key function being optimized.

Define assessment criteria

The assessment criteria are the solution properties against which the goodness of the solution will be assessed.

There is a wide variety of potential evaluation criteria, including:

- Development cost
- Lifecycle cost (aka *recurring cost*)
- Requirements compliance
- Functionality, including:
 - Range of performance
 - Accuracy
- Performance (execution speed), including:
 - Worst case performance
 - Average performance
 - Predictability of performance
 - Consistency of performance
- Dependability, including:
 - Reliability
 - System safety
 - Security

- Maintainability
- Availability
- Quality
- Human factors, including:
 - Ease of use
 - Ease of training to use
 - Support for standardized work flow
 - Principle of “minimum surprise”
- Presence or use of hazardous materials
- Environmental factors and impact, including:
 - EMI
 - Chemical
 - Biological
 - Thermal
- Power required
- Project risk, including:
 - Budget risk
 - Schedule risk
 - Technical risk (technology maturity or availability)
- Operational complexity
- Engineering support (tools and training)
- Verifiability
- Certifiability
- Engineering familiarity with the approach or technology

To perform this step, a small, critical set of criteria must be selected, and then a metric must be identified for each to measure the goodness of the candidate solution with respect to that criterion.

In SysML, we will model these concerns as value properties of the block used for the trade study.

Assign weights to criteria

Not all assessment criteria are equally important. To address that, each criterion is assigned a weight, which is a measure of its relative criticality. It is common to normalize the values so that the weights sum to a standard value, such as 1.00.

Define utility curve for each criterion

A **utility curve** for an assessment criterion defines the goodness of a raw measurement value. The computed utility value for a raw measurement is none other than the MoE for that criterion. It is common to normalize the utility curves so that all return values in a set range, say 0 to 10, where 0 denotes the worst case under consideration and 10 denotes the best case.

While any curve can be used, by far the most common approach is a linear curve (a straight line). Creating a linear utility curve is simple:

- Among the selected potential solutions, identify the worst solution for this criterion and define its utility value to be 0.
- Identify the best solution for this criterion and define its utility value to be 10.
- Create a line between these two values. That is the utility curve.

The math is very straightforward. The equation for a line, given two points (x_1, y_1) and (x_2, y_2) , is simply

$$y = \frac{y_2 - y_1}{(x_2 - x_1)}x + b$$

We have special conditions, such as (worst, 0) and (best, 10), on the linear curve. This simplifies the utility curve to:

$$moe = \frac{10}{best - worst} CandidateValue + b$$

And:

$$b = -\frac{10}{best - worst} worst$$

Where:

- *best* is the value of the criterion for the best candidate solution
- *worst* is the value of the criterion for the worst candidate solution

For example, let's consider a system where our criterion is *throughput*, measured in messages processed per second. The worst candidate under consideration has a throughput of 17,000 messages/second and the best candidate has a throughput of 100,000 messages/second. Applying our last two equations provides a solution of:

$$moe = \frac{Throughput}{8300} - 170/83$$

A third candidate solution, which has a throughput of 70,000 messages/second, would then have a computed MOE score of 6.39, computed from the above equation.

Assign MoE to each candidate solution

This step applies the constructed utility curves for each criterion to each of the potential solutions. The total weighted score for each candidate solution, known as its *weighted objective function*, is the sum of each of the outputs of the utility curve for each assessment criteria times its weight:

$$Weighted\ Objective\ Function_k = \sum_j Utility_j(criterion\ value_k) * Weight_j$$

That is, for each candidate solution k , we computed its weighted objective function as the sum of the product of that solution's utility score for each criterion j times the weight of that criterion. This is easier to apply than it is to describe.

Perform sensitivity analysis

Sometimes, the MoEs for different solutions are close in value but the difference is not really significant. This can result when there is measurement error, low measurement precision, or values are reached via consensus. In such cases, a lack of precision in the values can affect the technical selection based on the trade study analysis. This issue can be examined through *sensitivity analysis*, which looks at the sensitivity of the resulting MoE to small variations in the raw values. For example, consider the precision of the measurement of message throughput in the example used earlier. Is the value exactly 70,000, or is it somewhere between 68,000 and 72,000? Would that difference affect our selection? The sensitivity analysis repeats the computation with small variations in the value and looks to see if different solutions would be selected in those cases. If so, closer examination might be warranted.

Determine solution

The recommended solution is simply the candidate solution with the highest value of its computed objective function.

Example

We've seen the recipe description in the previous section. Let's apply it to our system and consider the generation of pedal resistance in a Pegasus smart bicycle trainer.

Identify key system functions

The key system function for the example trade study is **Produce Resistance**.

Identify candidate solutions

There are a number of ways to generate resistance on a smart bicycle trainer, and they all come with pros and cons:

- **Wind turbine:** A blade turbine that turns based on the power output. Cheap, light, and resistance increases with speed but in a linear fashion, not closely related to the actual riding experience where effort increases as a function of velocity cubed.
- **Electric motor with flywheel:** An electric motor generates resistance. Expensive and potentially heavy but can produce resistance in any algorithmically defined way.
- **Hydraulic with flywheel:** Moving fluid in an enclosed volume with a programmatically controlled aperture to generate resistance. The heaviest solution considered but provides smooth resistance curves.
- **Electrohydraulic:** Combining the hydraulic approach with an electric motor to simulate inertia. This solution is available as a pre-packaged unit for easy installation.

Define assessment criteria

There are many factors to consider when selecting a technology for generating resistance:

- **Accuracy:** This criterion has to do with how closely and accurately resistance can be applied. This is very important to many serious cyclists. This is a measured boundary of error in commanded wattage versus actual wattage. As this is a measure of deviation, smaller values are better.
- **Reliability:** This is a measure of the availability of services, as determined by **Mean Time Between Failure (MTBF)**, as measured in hours. Larger numbers are better because they indicate that the system is more reliable.
- **Mass:** The weight of the system increases the cost of shipping and makes it more difficult for a home user to move and set up the system. Smaller numbers are better for this metric.
- **Parts Cost:** This is a measure of recurring cost, or cost-per-shipped system. Smaller values are better.

- Rider Feel:** This is a subjective measure of how closely the simulated resistance matches a comparable situation on a road bike. This is particularly important in the lower power generation phase of the pedal stroke as well as for simulating inertia over a longer time-frame for simulated climbing and descending. This will be determined by conducting an experiment with experienced cyclists on hand-built mock-up prototypes over a range of fixed resistance settings and then averaging the results. The scale will be from 0 (horrible) to 100 (fantastic); larger numbers are better.

For this example, *Table 3.2* summarizes the values for the criteria for the four candidate solutions:

Candidate Solution	Accuracy (\pm watt)	Reliability MTBF (hour)	Mass (kilograms)	Parts Cost (\$US)	Rider Feel (survey, out of 100)
Hydraulic	5	4,000	72	800	80
Electric	1	3,200	24	550	95
Electrohydraulic	2	3,500	69	760	92
Wind Turbine	10	6,000	13	375	15

Table 3.2: Properties of candidate solutions

For our proposed **GenerateResistance_TradeStudy** block, these properties are modeled as value properties, as shown in *Figure 3.3*. Note that we added units such as **US_dollars** and **MTBF_Hours** following the data schema recipe from the previous chapter. **Mass[kilogram]** is already provided as units in the Cameo SysML SI Units model library. I also created a block to hold the worst-case value for each criterion from all the candidate solutions and another to hold the best case. We will use these values later in the evaluation of the utility curves:

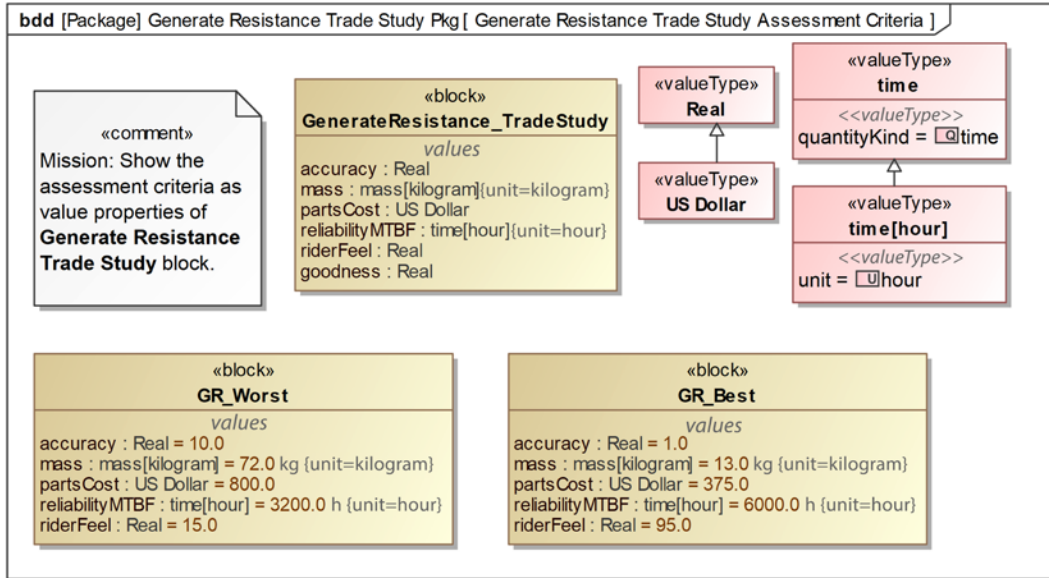


Figure 3.3: Example block and value properties for a trade study

Assign weights to criteria

In our example, we'll make the weights as follows:

- Accuracy: 0.30
- Mass: 0.05
- Reliability: 0.15
- Parts Cost: 0.10
- Rider Feel: 0.25

The weights will be reflected in a parametric constraint block (see Figure 3.4) for use in computation:

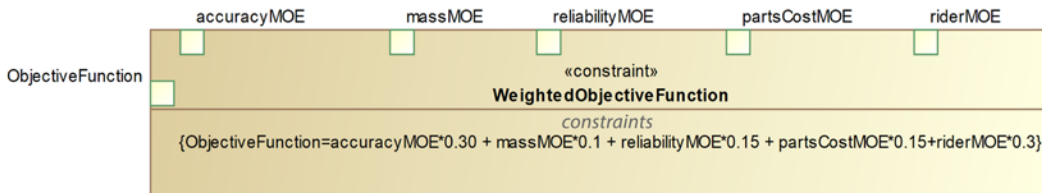


Figure 3.4: Objective function as a constraint block

This constraint block will be used to generate the overall goodness score (known as the *objective function*) of the candidate solutions.

Define utility curve for each criterion

In the approach we'll take here for determining the utility curves, we need to know the best and worst possible values for each of the criteria. As shown in *Figure 3.3*, **GR_Best** has all the value properties of **GenerateResistance_TradeStudy** but has the best values of any of the considered solutions defined as the default value of the value property. For example, **GR_Best::Accuracy** has a default value of 1 because ± 1 watt error is the best of any solution begin considered; similarly, **GR_Best::Reliability** has a default value of 6,000 because that's the best of any solution under consideration. **GR_Worst** has the worst case values; **GR_Worst::Accuracy** has a default value of 10 because a ± 10 watt error is the worst case in the trade study, and **GR_Worst::Reliability** = 3,200, the lowest MBTF of any proposed solution.

Now that we have our data, we can create the utility curves. In this example, we will use a linear curve (i.e. a straight line) for each criterion's utility curve. To assist in the computation, we will define a **LinearUtilityCurve** constraint block that constructs such as curve for us and incorporates the assumptions that the worst input should have a utility function value of 0 and the best input should have an output value of 10. This constraint block is shown in *Figure 3.5*:

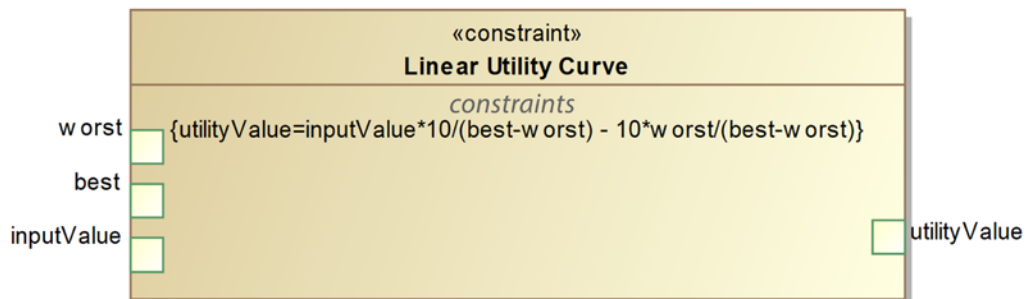


Figure 3.5: Linear utility curve

In this constraint block, the **worst** and **best** inputs are used to construct the slope and intercept of the utility curve for that criterion; the **inputValue** constraint parameter is the value of that criterion for the selection under evaluation; the resulting **utilityValue** is the value of the utility curve for the input value. Since we have five criteria, we must create five constraint properties from this constraint block, one for each utility function.

Assign MoE to each candidate solution

In our example, this results in four instance specifications that have values corresponding to these MoEs. For this purpose, let's assume that *Table 3.2* represents the raw measured or estimated values for the different criteria for the different solutions.

We then create a set of instance specifications that provide those specific values:

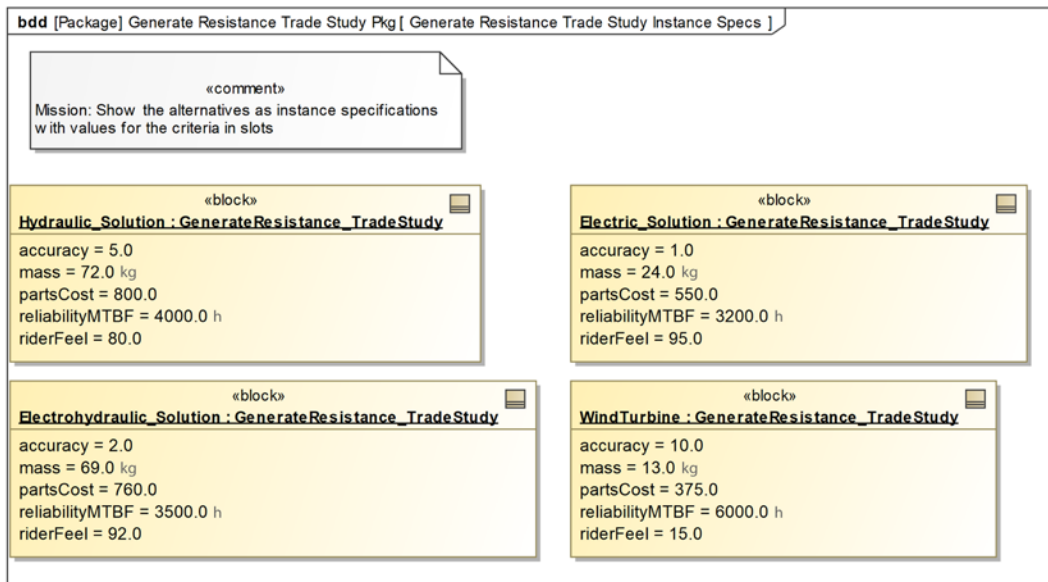


Figure 3.6: Instance specifications for trade study

We can now add the parametric diagram to the **Generate Resistance TradeStudy Pkg**; this will create a new block (with the same name as the package). Into this diagram we will drag the **GenerateResistance_TradeStudy**, **GR_Worst**, and **GR_Best** blocks, and connect the value properties to multiple occurrences of the **LinearUtilityCurve** constraint block. This results in *Figure 3.7*. While *Figure 3.7* looks complex, it's really not, since it just repeats a simple pattern multiple times. Cameo's parametric equation wizard makes connecting the constraint parameters simple:

- The **GeneralResistance_TradeStudy** block is shown with value properties representing the criteria of concern.
- In the middle, each criterion is represented by a **LinearUtilityCurve** constraint property connected to its source value property and to the corresponding input in the **itsWeightedUtilityFunction** constraint property.

- Additionally, relevant value properties for **GR_Best** and **GR_Worst** provide necessary information for the construction of each utility curve.
- The final computed **ObjectiveFunction** value then gives us the overall weighted score for a selected solution. It has a binding connector back to the **GenerateResistance_Tradestudy::goodness** value property for storing the result of the computation.

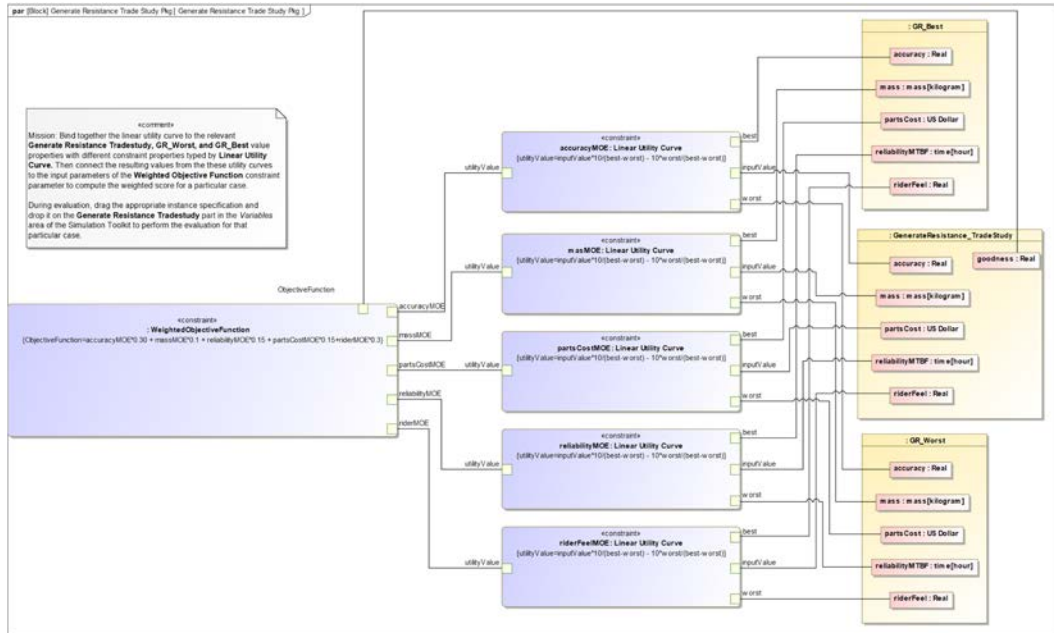


Figure 3.7: Complete parametric diagram

To perform the computations, simply compute the equation set for each instance specification. Cameo supports this computation with its **Simulation Toolkit** plugin. To do this, run the simulation toolkit; it precomputes the values. Drag each of the instance specifications from *Figure 3.6*, one at a time, drop them on the **GenerateResistance_Tradestudy** part in the **variables** area of the simulation toolkit window, and look at **WeightedObjectiveFunction.ObjectiveFunction** to get the computed goodness of the solution.

Figure 3.8 shows the output from the simulation toolkit when you drag and drop the Wind Turbine instance specification on the **:GenerateResistance_Tradestudy** part. The **goodness** value property is updated with the computed value resulting from the evaluation of the parametric diagram for this case:

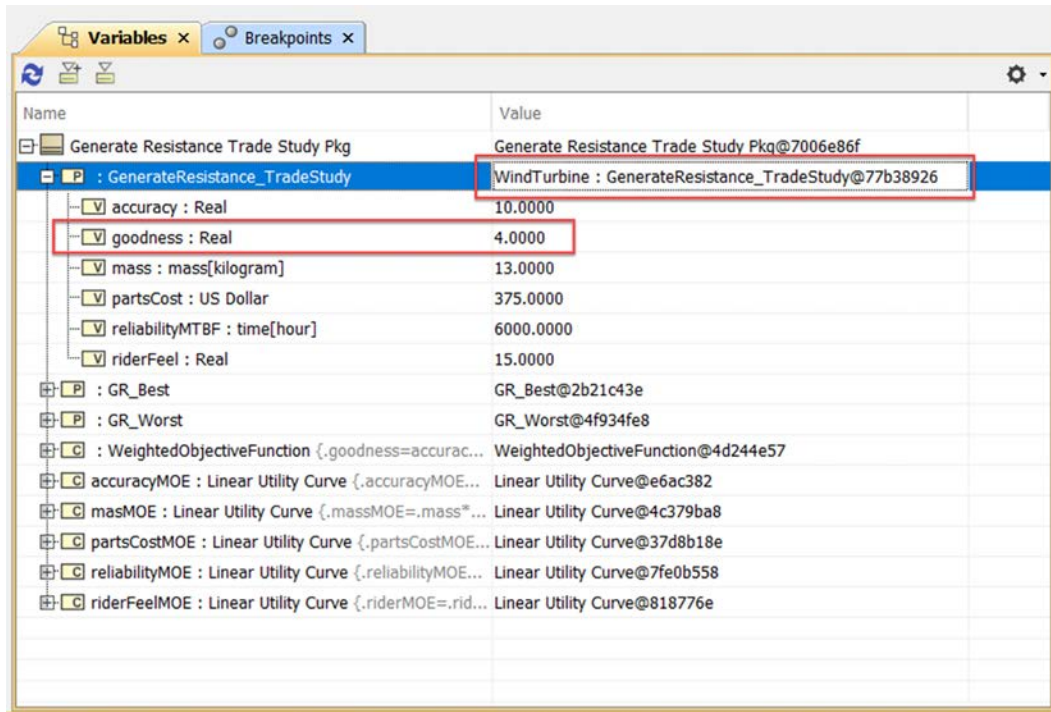


Figure 3.8: Evaluating the trade study

You can save this resulting set of instance values by selecting the part in the **Variables** area, right-clicking and selecting **Export value to > New Instance**. If you do this for all cases, you can construct the instance table in *Figure 3.9*:

#	Name	accuracy : Real	mass : mass[kilogram]	partsCost : US Dollar	reliabilityMTBF : time[hour]	riderFeel : Real	goodness : Real
1	Electric_Solution	1	24 kg	550	3200 h	95	7.6959
2	Electrohydraulic_Solution	2	69 kg	760	3500 h	92	5.9069
3	Hydraulic_Solution	5	72 kg	800	4000 h	80	4.5327
4	WindTurbine	10	13 kg	375	6000 h	15	4

Figure 3.9: Instance table for trade study

You can see that the **Electric_Solution** has the highest goodness score, so it is the winner of the trade study.

Perform sensitivity analysis

No sensitivity analysis needs to be performed on this analysis because the electric solution is a clear winner by a wide margin.

Determine solution

In this case, the electric motor wins the trade study because it has the highest computed weighted objective function value.

Architectural merge

The recipes in the previous chapter all had to do with system specifications (requirements). Those recipes create specifications in an agile way, using epics, use cases, and user stories as organizing elements. One of the key benefits of that approach is that different engineers can work on different functional aspects independently to construct viable specifications for the system. A downside is that when it comes to creating architecture, those efforts must be merged together since the system architecture must support all such specification models.

What to merge?

During functional analysis, various system properties are identified. Of these, most should end up in the system architecture, including:

- System functions
- System data
- System interfaces

Issues with merging specifications into a singular architecture

Merging specifications into an architecture sounds easy, right? Take all the features from all the use case analyses and copy them to the system block and you're done. In practice, it's not that easy. There are several cases that must be considered:

- The feature is unique to one use case
- The feature occurs in exactly the same form in multiple use cases
- The feature has different names in different use cases but is meant to be the same feature
- The feature has the same name and form in different use cases but is intended to be a semantically distinct feature
- The feature occurs in multiple use cases but is different in form:
 - Same name, different properties
 - Different name, different properties, but nevertheless still describes the same feature

In this discussion, the term **property** refers to aspects such as structuring, argument or data type, argument order, feature name, type of service (event reception or operation), and metadata such as extent, units, timeliness, dependability, and precision.

As an aside, we should **copy** the features to the system block rather than **move** or **reference** them because we want to preserve the integrity of the use case analysis data. This allows us to come back later and revisit the use case models and modify them, if the stakeholder need to change or evolve. This does mean some additional work to maintain consistency, but it can save significant time overall.

Cases 1 and 2 are trivially simple; just add the feature to the system block. The other cases require some thought. Although *trivial* might be an overstatement for interfaces since they reference the local sandbox proxies for the actual actors rather than the actors themselves, so some cleanup is required to deal with that. This recipe will address that concern.

In a traditional V lifecycle, this merge takes place a single time, but in an agile approach it will be applied repeatedly, typically once per iteration. Our approach supports both traditional and agile lifecycles.

Purpose

The purpose of the recipe is to incorporate system features identified during functional analysis into our architectural model.

Inputs and preconditions

Analysis is complete with at least two use cases, including the identification and characterization of system functions, system data, and system interfaces relevant to the use cases.

Outputs and postconditions

A system block is identified that contains the relevant system properties identified in the incorporated use cases.

How to do it

While non-trivial, the recipe for merging use case features into the architecture is at least straightforward (*Figure 3.10*).

Please note that this can be done once, as it would be in a traditional V lifecycle, or iteratively, as would be done in an agile approach:

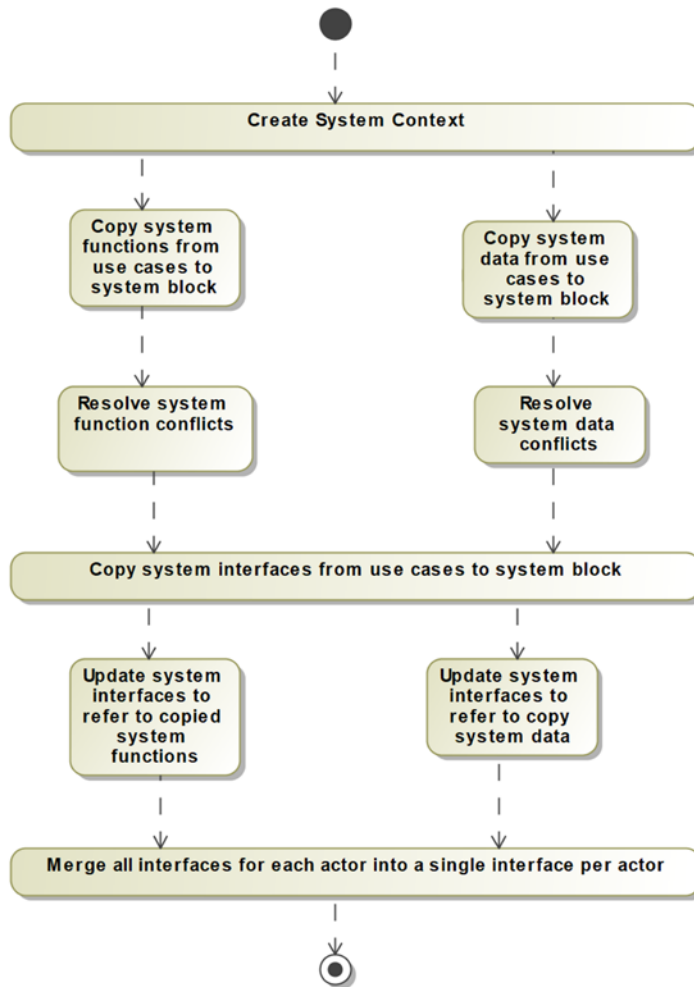


Figure 3.10: Architectural merge workflow

Create system context

The system context includes a block denoting the system of concern and its connection to the actors in the environment with which it interacts. This is normally visualized as a “context diagram,” a block definition diagram whose purpose is to show system context. We call this BDD the **system type context**. In addition, we want to show how the system block connects to the actors. This is shown as an internal block diagram and is referred to as the **system connected context**.

Copy system functions from use cases to system block

During use case analysis, use case blocks are created and elaborated. The primary reasons this is done is to 1) create high-quality requirements related to the use case, 2) identify the relevant actors for the use case, and 3) characterize system features necessary to support the use case. This latter purpose includes the identification of system data elements (represented as value properties and blocks and is detailed in the use case logical data schema), system functions (represented as actions executed in operations and signal receptions), and system behaviors (represented using activity, sequence, and/or state machines). This step copies the operations and signal receptions from the use cases to the system block. It is important to note that this must be a **deep copy**; by that, I mean that any references to types should refer to types defined in the architecture not the original types in the use case package.



Certain features may have been added to the use case block for purposes other than specification, such as to support simulation or to aid in debugging the use case analysis. Such features should be clearly identified; I recommend creating a «nonNormative» stereotype to mark such features for just this purpose. Non-normative features need not be copied to the system block, since they do not levy requirements or constraints on the system structure or behavior.

Resolve system function conflicts

As discussed earlier in the recipe description, there are likely to be at least some conflicts in the system functions coming from different use cases. These can be different operations that are meant to be the same, or the same operation meant to do different things. These cases must be resolved. For the first case – different operations meant to be the same thing – a single operation should generally be created that meets all the needs identified in the included use cases. This includes their inputs, outputs, and functionality. For the second case – the same operation meant to do different things – it is a matter of replicating the conflicting functions with different names to provide the set of required system behaviors. In both cases, the hard part is identifying which is which.

Copy system data from use cases to system block

This step copies the data elements from the various use cases to the system block, including the value properties of the use case and the data schema that defines the data relations. Remember, however, that if we just copy the operations, the input and output parameters of the functions will refer to the original model elements in the use case packages.

As a part of this step, we must update the system functions to refer to our newly copied data elements. I refer to this as a *deep copy*. As noted in the paragraph on copying functions, non-normative value properties and data need not be copied to the architecture.

Resolve system data conflicts

The same kinds of conflicts that occur with copying the system functions can also occur with system data. This step identifies and resolves those conflicts.

Copy interfaces from use cases to system block

As a part of use case analysis, interfaces between the use case and the actors are defined. They need to be copied to the architecture as well.

Update interfaces to refer to copied system functions

As with the copied functions and data, the interfaces will also refer to their original referenced elements back in the use case analysis packages. These references must be updated to point to the copied system features in the architecture, taking into account that conflicts have been resolved, so the function names and parameters list are likely to have changed.

Update interfaces to refer to copied systems data

The interfaces can refer to data either as flow properties or as parameters on the functions within the interface. This step resolves those references to the architectural copies of those data elements as updated as a solution of the “resolve conflicts” step before.

Merge all interfaces for each actor

Actors interact with multiple use cases. This means that different interfaces relating to the same actor will be copied over from the use cases to the system architecture. Commonly, the set of interfaces from the use cases to the actor are merged into a single interface between the system and that actor. However, if an interface from the system to an actor is particularly complex, it might result in multiple interfaces supporting different kinds of services.

Example

In this example, we will use some of the use cases we analyzed in *Chapter 2, System Specification*:

- Control Resistance
- Emulate Basic Gearing

- Emulate Front and Rear Gearing
- Measure Performance Metrics

Create system context

The system type context can be shown in a couple of ways. *Figure 3.11* shows one way:

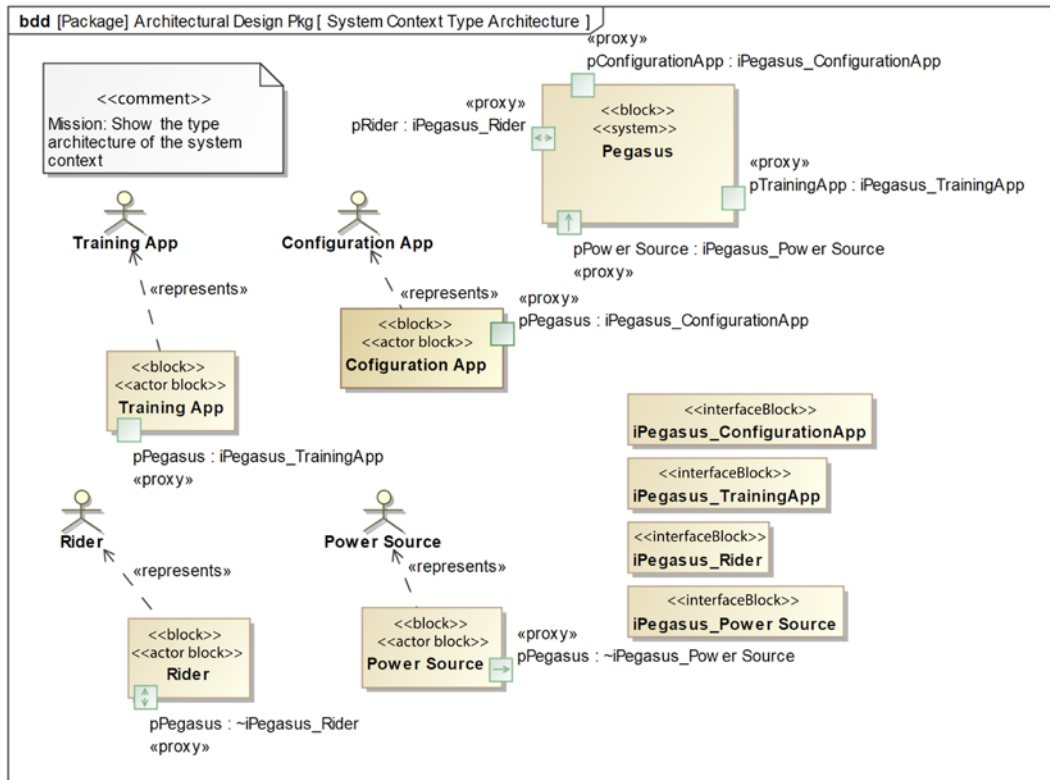


Figure 3.11: System type context as a block definition diagram

I created the stereotype «actor block» for blocks specifically representing actors because in Cameo, actors cannot have ports. The «represents» stereotype of dependency is used to explicitly relate the actor blocks and the actors. The «system» stereotype is from Cameo’s SysML profile in SysML::Non-Normative Extensions::Blocks package.

And – as I plan to use ports for actual connections – it can also be shown in an IBD, as in *Figure 3.12*.

This shows the system connected context and focuses on how the system and the actors connect to each other:

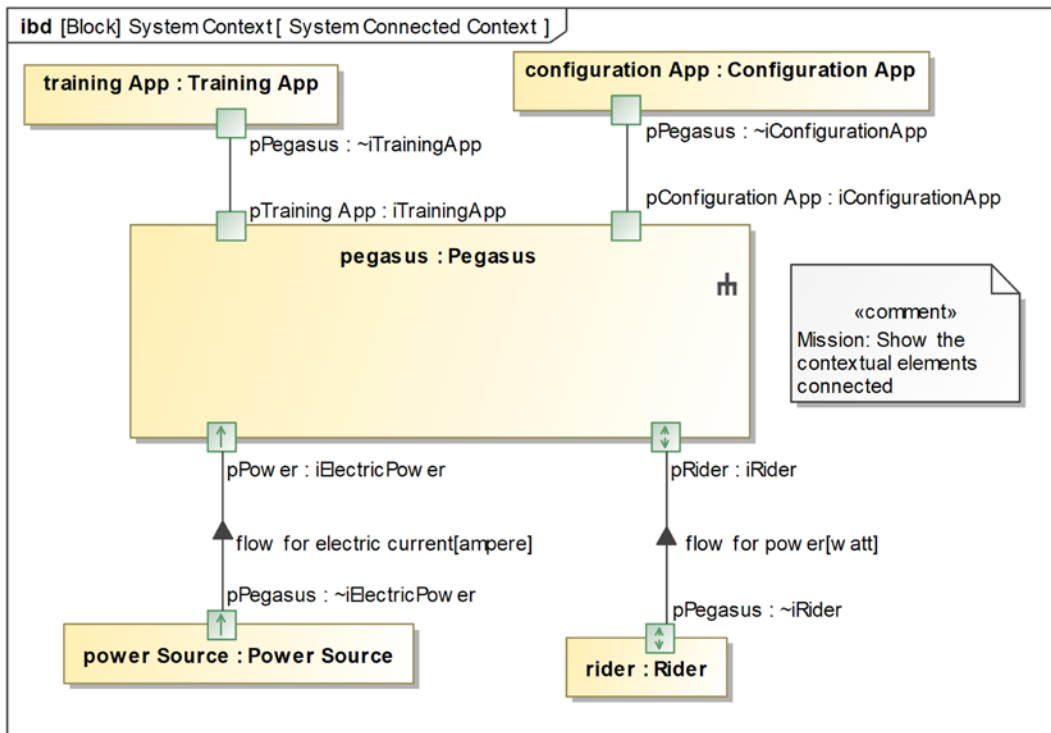


Figure 3.12: System connected context as internal block diagram

Copy system functions from use cases to systems block

To illustrate this, *Figure 3.13* shows a block definition diagram with all the included use cases (as use case blocks) with their system functions and value properties and up-to-date Pegasus block:

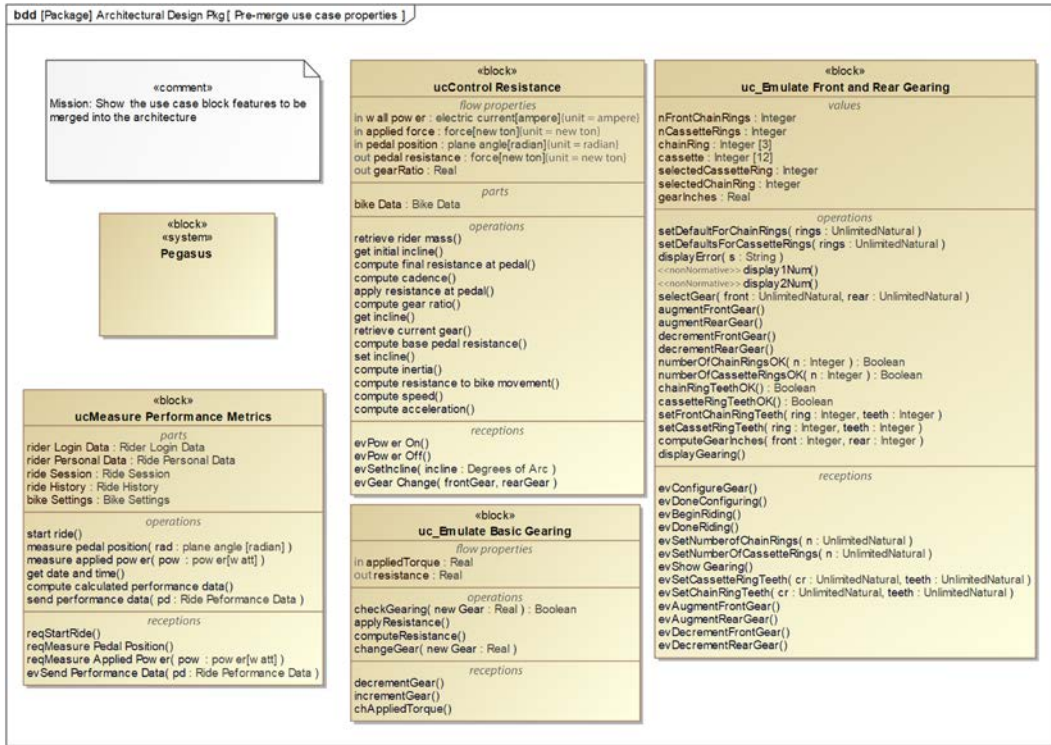


Figure 3.13: Use case block features to copy

It is important to perform a deep copy of the elements. Operations, signal receptions, flow, and value properties may reference types that are defined in the use case packages. This also includes reference and part properties owned by the copied elements. Simply copying the owning element doesn't change that. We need to update the copy to point to the copied types as well.

Similarly, signal receptions will specify a signal that they receive, which is located in the use case packages. These signal receptions must be updated to reference the new copy of the signal in the architecture.

Figure 3.14 illustrates the problem:

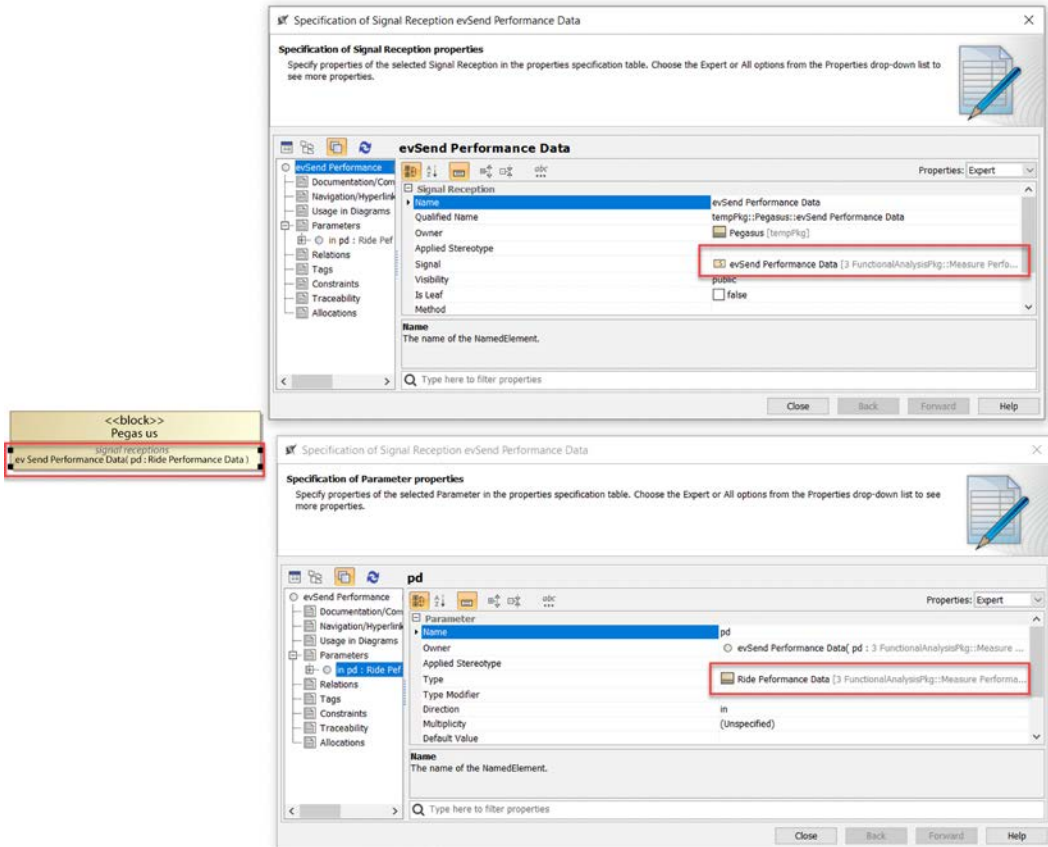


Figure 3.14: The problem with copy

The signal reception `evSendPerformanceData` is copied from the `ucMeasure Performance Metrics` block to the `Pegasus` block. If we look at the specification of the signal reception, we see that it references the signal back in the use case package `FunctionalAnalysisPkg::Measure Performance Metrics Pkg`. Further, it carries a parameter, `pd:Ride Performance Data`, and that block defining the parameter is, once again, located back in the use case package. Since we are doing an architectural merge that includes both data and signals, these references *ev* must be updated to point to those elements in the architectural packages, as shown in *Figure 3.15*:

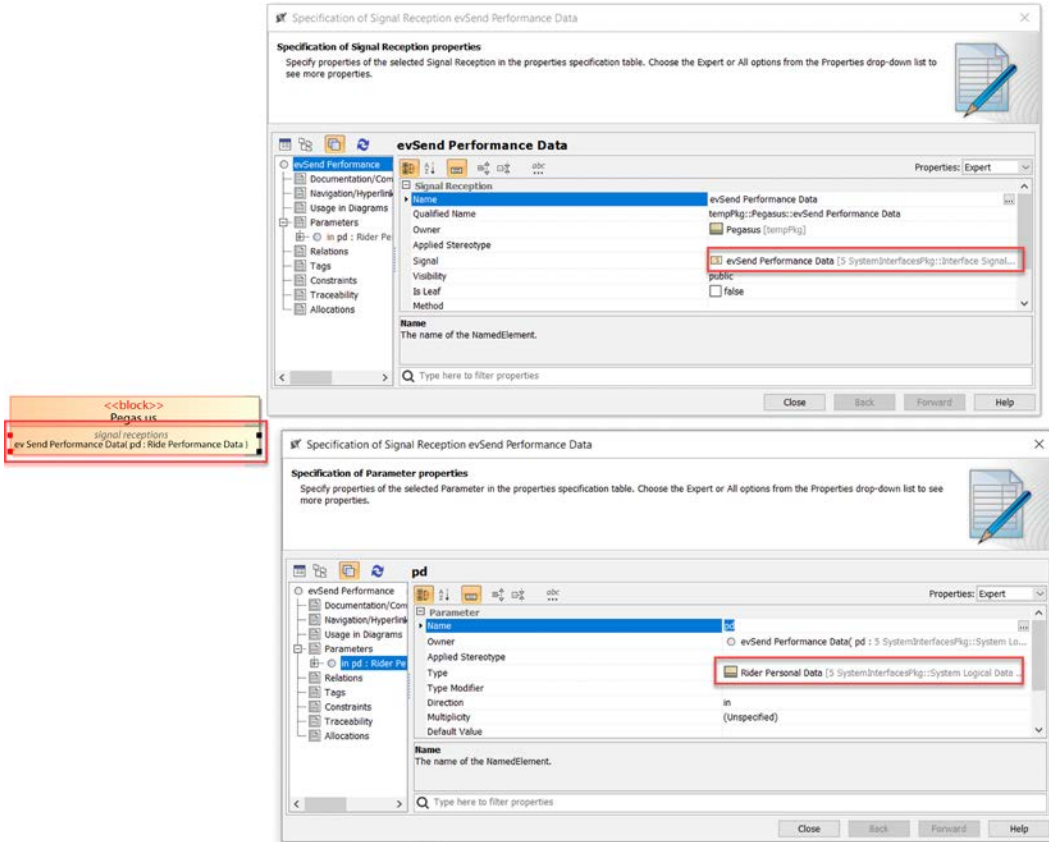


Figure 3.15: Updated copies to local referents

These copy updates aren't difficult to do, although it can be a bit time-consuming.

Resolve system function conflicts

As you can see in *Figure 3.13*, the system block must merge a large number of functions. For the most part, there is no conflict; either the system functions are unique, such as `Uc_MeasurePerformanceMetrics::send performance data()` or they do the same thing in every usage. Nevertheless, a detailed inspection uncovers a few cases that must be considered.

For example, consider `Uc_EmulateBasicGearing::changeGear(newGear: Real)` and related functions like `checkGearing(gear: Real)` and the event `ChangeGear(new gear: Real)`. These all have to do with the emulation of basic gearing. How do those functions relate to the specific front and rear gearing functions of the use case `Emulate Front and Rear Gearing`, such as `selectGear(front: Unlimited Natural, rear: Unlimited Natural)`, `augmentFrontGear()`, `augmentRearGear()`, `decrementFrontGear()`, and `decrementRearGear()`?

It could be that the `changeGear` function is called by the more specific functions that set the gearing for the front and rear chain rings, but does the training app need both the gearing (in gear-inches) and the currently selected front and rear? Gearing is a function of not only the gear ratio between the front and rear chain rings but also the wheel size, which the training app may not know. If the training app uses the current front and rear as a display option for the rider, does it still need the gearing value? Probably not, since the system itself calculates metrics such as power, speed, and incline, not the training app. In this case, it is enough to send the gearing event to the training app.

There is some subtlety in the various system functions and events. For example, the training app sends the event triggering the `Uc_MeasurePerformanceMetrics::reqStartRide()` event reception to indicate that it is ready to begin the ride and therefore ready to receive incoming data, while `Uc_EmulateFrontandRearGearing::evBeginRiding()` event reception is triggered by an event sent from the rider and not the training app. This points out just how important it is for every important system property to have a meaningful description! It would be very easy to get these events confused.

Copy system data from use cases to systems block

In addition to showing the system functions to copy, *Figure 3.13* also shows the data, as value properties, to copy as well.

Resolve system data conflicts

Just as with the system functions, the data copies over without issue, but there are a few questions to resolve. `Uc_EmulateFrontandRearGearing::gearInches: Real` is the gearing ratio computed as a function of front and rear chain rings and wheel size. This is the same as `Uc_ControlResistance::gearRatio: Real`, so they can be merged into a single feature. We will use `gearRatio: Real` for the merged value property.

Copy interfaces from use cases to systems block

A number of interfaces were produced in the merged use case analyses. To start with, we will copy not only the interfaces themselves but also the events and any data types and schema passed as arguments for those events.

Figure 3.16 shows the copied interfaces in three different browsers, each showing different kinds of elements. The one on the left shows the interface blocks copied over and exposes their operations and flow properties. The middle browser shows (most of) the events copied over. The one on the right shows the copied data elements:

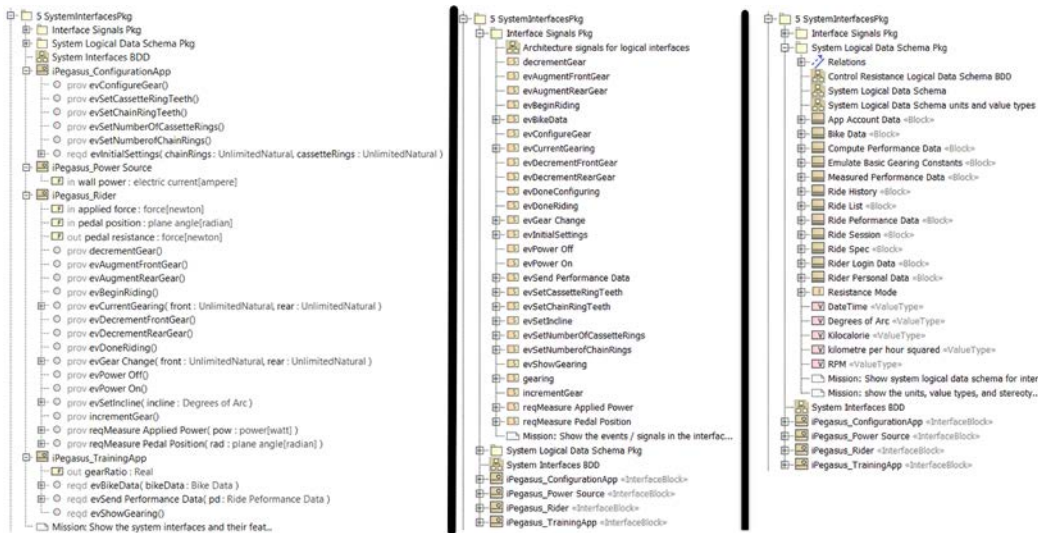


Figure 3.16: Interfaces copied to the architecture

Update interfaces to refer to copied system functions

The copied interfaces refer to events back in the originating use case analysis packages. In our use case analyses, we followed the convention that all services in these interfaces are modeled as event receptions, so we can limit our focus to ensure that the interface blocks refer to the event definitions in the architecture instead of their definition in their original source, the use case analysis packages.

Update interfaces to refer to copied systems data

The system functions and event receptions of the Pegasus system block must all refer to the copied data types, not the original data types. This is also true for any flow properties in the interface blocks.

In this particular case, there is no use for user-defined types in the interfaces, so the data references are fine without changes.

Merge all interfaces for each actor

The last step in the recipe is to create the interfaces for the actors. In the previous chapter, we created interfaces not to the actors themselves during use case analysis but rather to blocks meant to serve as local stand-ins for those actors. The naming convention for the interfaces reflects that choice. For example, the interface block `iUc_EmulateBasicGearing_aEBG_Rider` is an interface between the use case `EmulateBasicGearing` and its local proxy for the `Rider` actor, `aEBG_Rider`. In this way, we can see which interfaces to merge together. All interfaces that contain `a...Rider` should be merged into an `iPegasus_Rider` interface. A similar process is followed for each of the other actors. This is because the system architecture must merge all the interfaces referring to the same actor. At the end, we'll have an interface for each actor that specifies the interface between the system and the actor.

When it's all said and done, *Figure 3.17* shows the merged interfaces. Additional actors and interfaces may be uncovered as more use cases are elaborated, or they will be discovered as system design progresses:

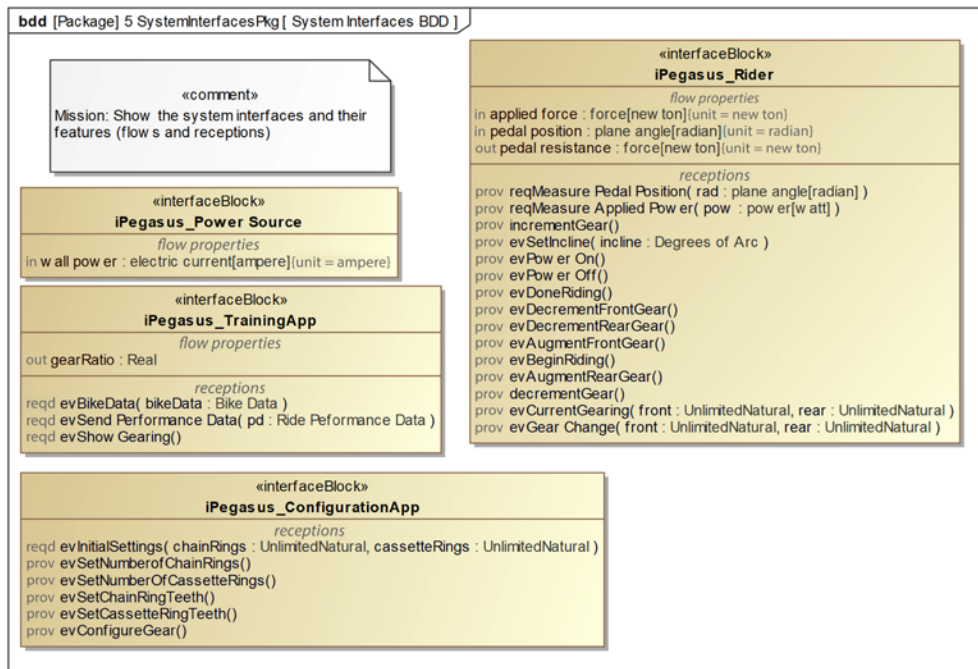


Figure 3.17: Merged system interfaces

Pattern-driven architecture

Design patterns are “generalized solutions to commonly occurring problems,” (*Design Patterns for Embedded Systems in C*, by Bruce Douglass, Ph.D. 2011). Let’s break this down.

First, a design pattern captures a design solution in a general way. That is, the aspects of the design unique to the specific problem being solved are abstracted away, leaving the generally necessary structures, roles, and behaviors. The process of identifying the underlying conceptual solution is known as **pattern mining**. This discovered abstracted solution can now be reapplied in a different design context, a process known as **pattern instantiation**. Further, while each design context has unique aspects, design patterns are appropriate for problems or concerns that reappear in many systems designs.

To orient yourself to work with design patterns, you should keep in mind two fundamental truths:

- Design is all about the optimization of important properties at the expense of others (see the previous recipe about trade studies at the start of this chapter).
- There are almost always many different solutions to a given design problem.

With many design solutions (patterns) able to address a design concern, how is one to choose? Simple – the best pattern is the one that solves the design problem in an optimal way, where “optimal” means that it maximizes the desired outcomes and minimizes the undesired ones. Typically, this means that the important **Qualities of Service (QoS)** properties should be given a higher weight than those properties that are less important. A good solution, therefore, solves the problem by providing the desired benefits at a cost we are willing to pay. This is often determined via a trade study.

Dimensions of patterns

Patterns have four key aspects, or dimensions:

- **Name:** The name provides a way to reference the pattern independent of its application in any specific design.
- **Purpose:** The purpose of the pattern identifies the design problem the pattern addresses and the necessary design context preconditions necessary for its use.
- **Solution:** The solution details the structural elements, their collaboration roles, and their singular and collective behavior.

- **Consequences:** The pattern consequences highlight the benefits and costs from the use of the pattern, focusing on the design properties particularly optimized and deoptimized through its use. This is arguably the most important dimension because it is how we will decide which pattern to deploy from the set of relevant patterns.

Pattern roles

Structural roles are fulfilled by the structural elements (blocks) in systems engineering. A **role** can be defined as the *use of an instance in a context*. Design patterns have two broad categories of roles. The first is as *glue*. Roles of this type serve to facilitate and manage the execution of the collaboration of design elements of which the pattern is a part.

The second category is as a **collaboration parameter**. Design patterns can be described as *parameterized collaborations*. The parameterized roles are placeholders that will be replaced by specific elements from your design. That is, some design elements will be substituted for these roles during pattern instantiation; design elements substituting for pattern parameters are known as **pattern arguments**. Once the pattern is instantiated, the glue roles will interact with the pattern arguments to provide the optimized design solution.

Patterns in an architectural context

Architecture is *design writ large*. Architectural design choices affect most or all of the system and all more detailed design elements must live within the confines of the architectural decisions. As we said in the lead-in to this chapter, we are focused on six key views of architecture: subsystem and component view, concurrency and resource view, distribution view, data view, dependability view, and deployment view. Each of these views of architecture has its own rich source of design patterns. A system architecture is a set of architectural structures integrated with one or more patterns in each of these views.



For further information, please see *Real-Time Design Patterns* by Bruce Douglass, Ph.D. 2011 or *Pattern-Oriented Software Architecture Volume 1: A System of Patterns* by Bushmann, Meunier, Rhnert, Sommerlad, and Stal 1996.

This recipe will provide a workflow for the identification and integration of a design pattern in general; later recipes will use this pattern to address their more specific concerns.

Purpose

The purpose of design patterns is twofold. First, design patterns capture good design solutions so that they can be reused, providing an engineering history of good design solutions. Secondly, they provide a means by which designers can reuse existing design solutions that have proven useful.

Inputs and preconditions

The fundamental precondition for the use of design patterns is that there is a design problem to be solved, and some design elements have been identified.

Outputs and postconditions

The primary output of the application of design patterns is an optimized collaboration solving the design solution.

How to do it

The use of design patterns is conceptually simple, as can be seen in *Figure 3.18*. This simple flow somewhat belies the difficulty of good design because a good design generally has several interconnected patterns working together to optimize many aspects at once:

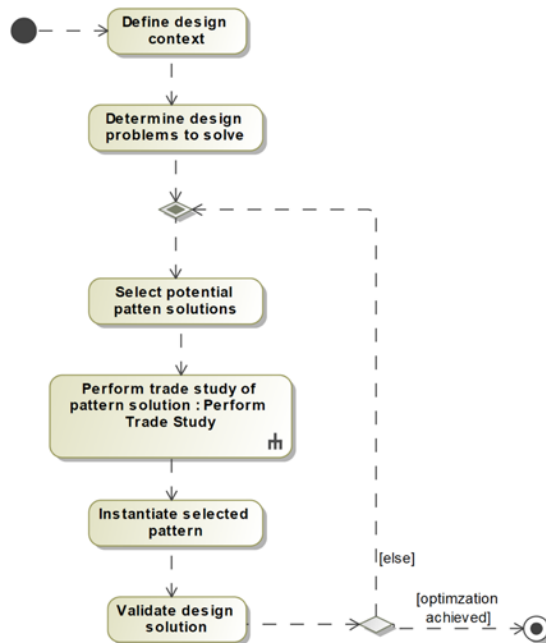


Figure 3.18: Apply design Patterns

Define design context

This step is generally a matter of defining essential elements of the design (sometimes known as the **analysis model**) without optimization. This includes the important properties including value properties, operations, state behavior, and relations. A general guideline I use is that this design context is demonstrably functionally correct, as demonstrated through execution and testing, before optimization is attempted. Optimizing too soon (i.e. before the design context is verified) generally leads to bad outcomes.

Determine design problem to solve

During the development of the design context, it is common to uncover a design issue which impacts the design qualities of services, such as performance, reusability, safety, or security. These design issues have often been solved before by other designers in many different ways.

Select potential pattern solutions

This step involves reviewing the pattern literature for solutions that other designs have found to be effective. A set of patterns is selected from the potential solution candidates based on their applicability to the design issue at hand, the similarity of purpose, and the aspects of the design that they optimize.

Perform trade study of pattern solution

Earlier in this chapter, we highlighted a recipe called *Architectural Trade Studies*. In some cases, a full-blown trade study may not be called for, but in general, this step of the recipe selects from a set of alternatives, which is what a trade study does. The observant reader will note that the icon used in *Figure 3.18* for this step is a *call behavior*, which is a formal use of the earlier recipe.

Instantiate selected pattern

Once a pattern is selected, it must be instantiated. This is largely a matter of creating the structural and behavioral elements of the pattern and making small changes to some of the existing design elements to make them proper arguments for the design pattern parameters (aka *refactoring*).

Validate design solution

Once the pattern is instantiated, it must be examined for two things. First, before the pattern was instantiated, the design collaboration worked, albeit suboptimally (*Step 1: Define design context*). We want to verify that we didn't break it so that it no longer works. Secondly, we applied the design pattern for a reason – we wanted to achieve some kind of optimization.

We must verify that we have achieved our optimization goals. If we have not, then a different pattern should be used instead or in addition to the current pattern. Patterns can be combined to provide cumulative effects in many cases.

Example

In this example, we will take a non-architectural design context. Later examples will apply the recipe to the Pegasus architecture for specific views of architecture.

Define design context

In this example, let's consider a part of the internal design that acquires measured ride data from the power and pedal position sensors. See *Figure 3.19*:

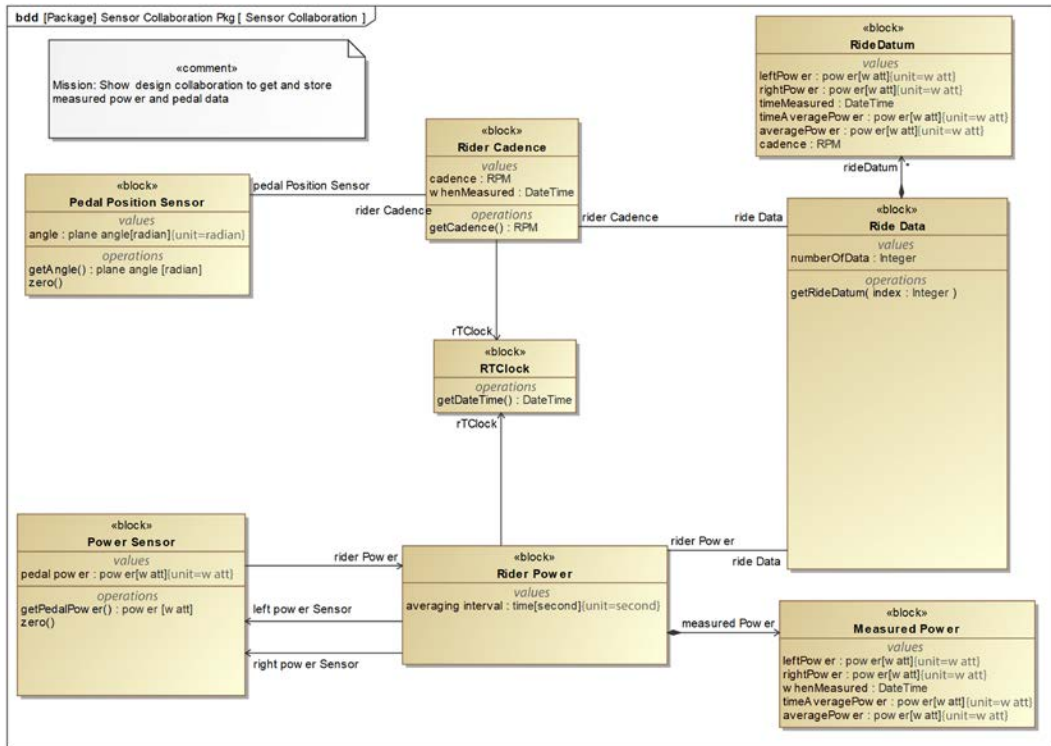


Figure 3.19: Design collaboration (pre-pattern)

The design collaboration shows three device drivers: two for rider power (left and right sensors) and one for the pedal position for determining pedal cadence.

The *Rider Power* block gets the data from the two *Power Sensor* instances and aggregates it,

along with the current time into a `Measured Power` block that includes left and right power, the average of those two values, the time-averaged power, and a time stamp. The `Rider Cadence` block gets the pedal position and, using the real-time clock, can determine cadence. The `Ride Data` block then aggregates this data into an ordered collection `Ride Datum` constituting the information about the ride. The actual design incorporates much more computed data, such as computed speed and distance, but we'll ignore that for now.

Determine design problem to solve

Some design optimization questions arise.

What's the best way to get the data from the sensors in a timely way? It must be sampled fast enough so as not to miss peak power and cadence spikes (riders love that kind of data) and the data from the two different sources must be synchronized in terms of when they are measured. On the other hand, sampling at too high a rate requires more data storage and could limit the length of rides that can be recorded and stored.

Further, note the existence of multiple clients for the real-time clock. Is having all the clients request the current time the best approach?

Select potential pattern solutions

Let's consider three design patterns that address the concern of getting data in a timely manner (from the author's *Design Patterns for Embedded Systems in C*, 2011): the *Interrupt Pattern*, *Opportunistic Polling Pattern*, and *Periodic Polling Pattern*. The first creates an interrupt driver that notifies the clients when data becomes available. The second pattern requires the client to poll the data sources when it gets a chance. The last pattern is a modification of the second pattern in which the client is notified, via a timer interrupt, when it should go poll the data.

Figure 3.20 shows the *Interrupt Polling Pattern*. It is conceptually very simple: an interrupt handler installs a set of interrupt handler functions by first saving the old address in the interrupt vector table for the selected interrupt, and then it puts the address of the desired interrupt handler in its place. Later, when that interrupt occurs, the CPU invokes the selected interrupt handler function. There are some subtleties of writing interrupt handlers, but that's the basic idea:

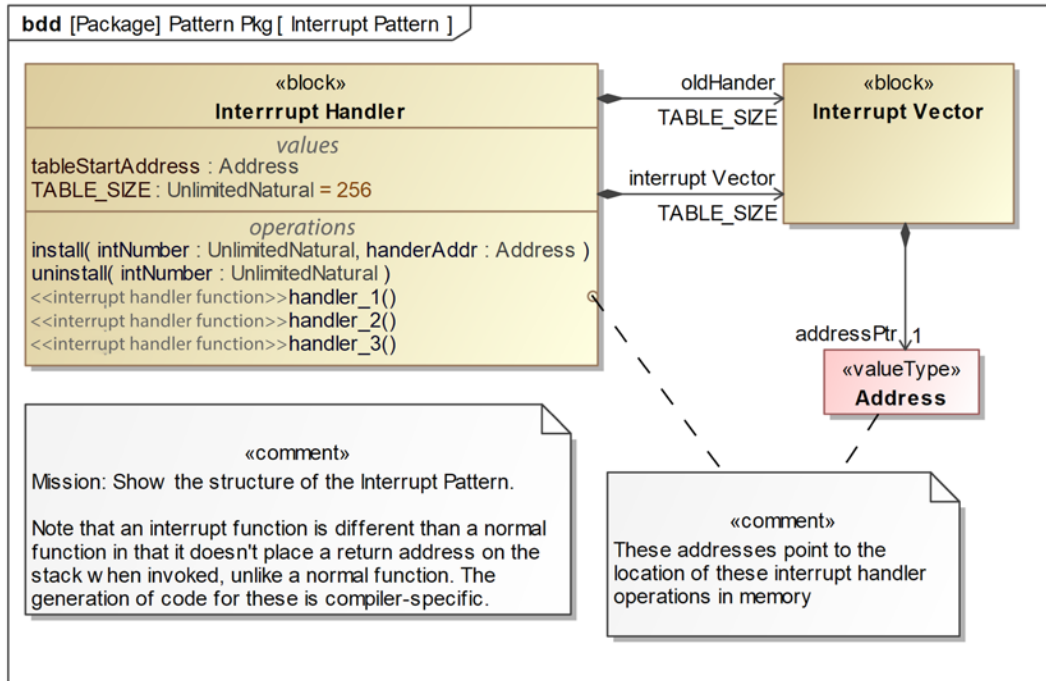


Figure 3.20: Interrupt pattern

Figure 3.21 shows the *Opportunistic Polling Pattern*. It works by having a Client, on whatever criteria it decides to use, invoke the `OpportunisticPoller::poll()` function.

The OpportunisticPoller then gets data from all the devices it polls (defined by NUM_POLL_DEVICES) and returns it to the Client. Easy peasy:

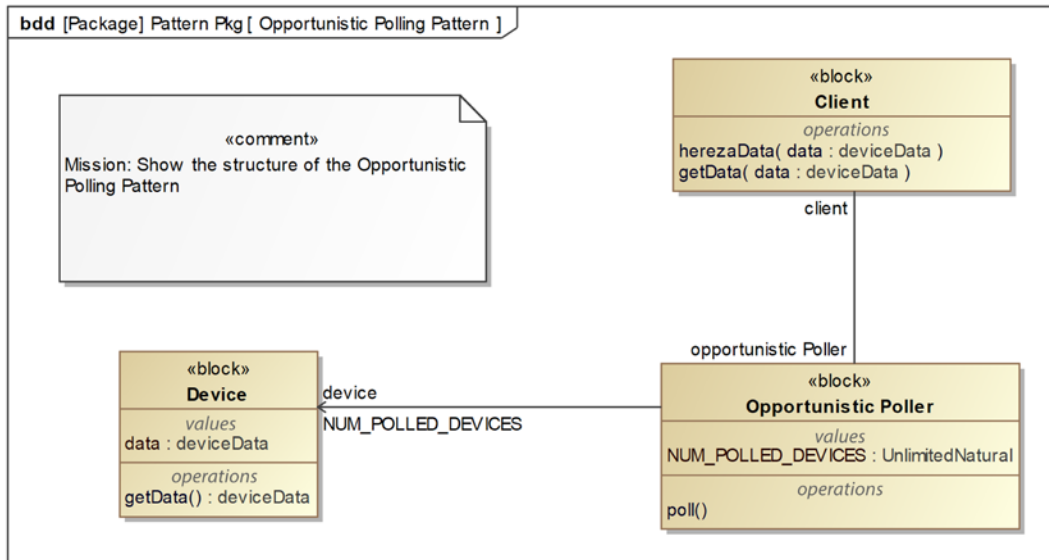


Figure 3.21: Opportunistic polling pattern

Figure 3.22 shows a specialized form of the previous pattern called the *periodic polling pattern*. In this pattern, the polling is driven by a timer; **PeriodicPoller** initializes and sets up the timer, which then invokes it with the specified period (ms are used here, but finer-grained polling can be supported with appropriate hardware resources):

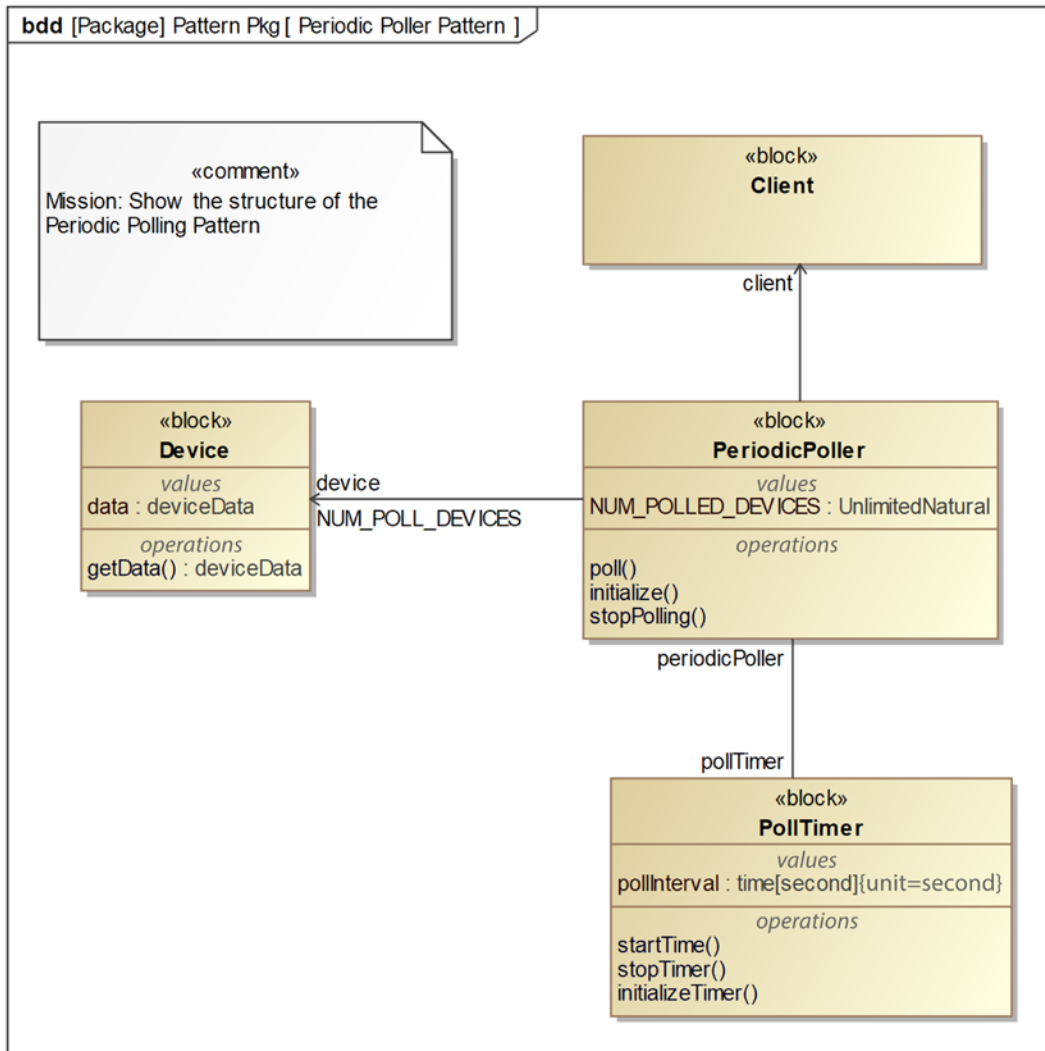


Figure 3.22: Periodic polling pattern

The consequences of the patterns are summarized in *Table 3.3*:

Pattern	Pros (Benefits)	Cons (Costs)
Interrupt	<ul style="list-style-type: none"> • Highly responsive to incoming data events 	<ul style="list-style-type: none"> • Can lead to data corruption unless interrupts are disabled during interrupt processing • Can lead to data loss if the interrupt handler takes too long • Can lead to processing starvation if data arrival is too frequent
Opportunistic Polling	<ul style="list-style-type: none"> • Simple implementation • Efficient use of processor resources since data is acquired when the processor is otherwise idle 	<ul style="list-style-type: none"> • No timeliness guarantee • Slow polling can lead to data loss • Less responsive to incoming data • Data can be corrupted if interrupts are not disabled during polling
Periodic Polling	<ul style="list-style-type: none"> • Relatively simple implementation • Period is tunable for data arrival frequency 	<ul style="list-style-type: none"> • Data can be lost if the polling period is too long • Processing starvation can occur if the frequency is too high • Handling timer interrupt must be short or it can lead to data loss • Data can be corrupted if interrupts are not disabled during polling

Table 3.3: Design patterns for timely data acquisition

Perform trade study of pattern solution

I won't detail the process of conducting the trade study but instead provide a summary in *Table 3.4*. In this trade study, the *Periodic Polling Pattern* is selected:

Pattern	Criteria						Total Weighted Score
	Simplicity ($W_1 = 0.1$)		Avoid Data Loss ($W_2 = 0.7$)		Resource Efficiency ($W_3 = 0.2$)		
	MoE	Score	MoE	Score	MoE	Score	
Interrupt	4	0.4	6	4.2	6	1.2	5.8
Opportunistic Polling	6	0.6	3	2.1	2	0.4	3.1
Periodic Polling	2	0.2	6	4.2	8	1.6	6

Table 3.4: Results of the trade study

The results in the table show that the *Periodic Polling Pattern* is selected as the best fit for our design.

Instantiate selected pattern

The next step is to instantiate the pattern. Conceptually, this is a matter of replacing the parameters of the pattern with design elements. In practice, this is often done via specialization of the parameters, so the design classes now inherit the structure and behavior required to integrate into the pattern. Additionally, there may be some small amount of reorganization (known as *refactoring*) such as the removal or modification of relations in the pre-pattern collaboration.

Figure 3.23 shows the instantiated design pattern. I've added colored shading to the pattern elements to highlight their use. The `RiderData` block subclasses the `PeriodicPoller` block; the `RiderCadence` and `RiderPower` both subclass the `Device` block. I've also elected to show the inherited features in the updated design blocks (indicated with a caret, ^). Those inherited operations will be implemented in terms of the existing functions within those specialized blocks. Further, the inherited data elements will not be used; the original data elements from the design elements will be used in their stead.

Note also that the original relations between `RideData` and `RiderCadence` and `RideData` and `RiderPower` have been deleted and subsumed by the single relation from `RideData` to `Device`; this relation applies to both `RiderCadence` and `RiderPower` since the latter two blocks are subtypes of `Device`.

In this case, NUM_POLL_DEVICES is 2. As mentioned previously, this change falls under the heading of refactoring:

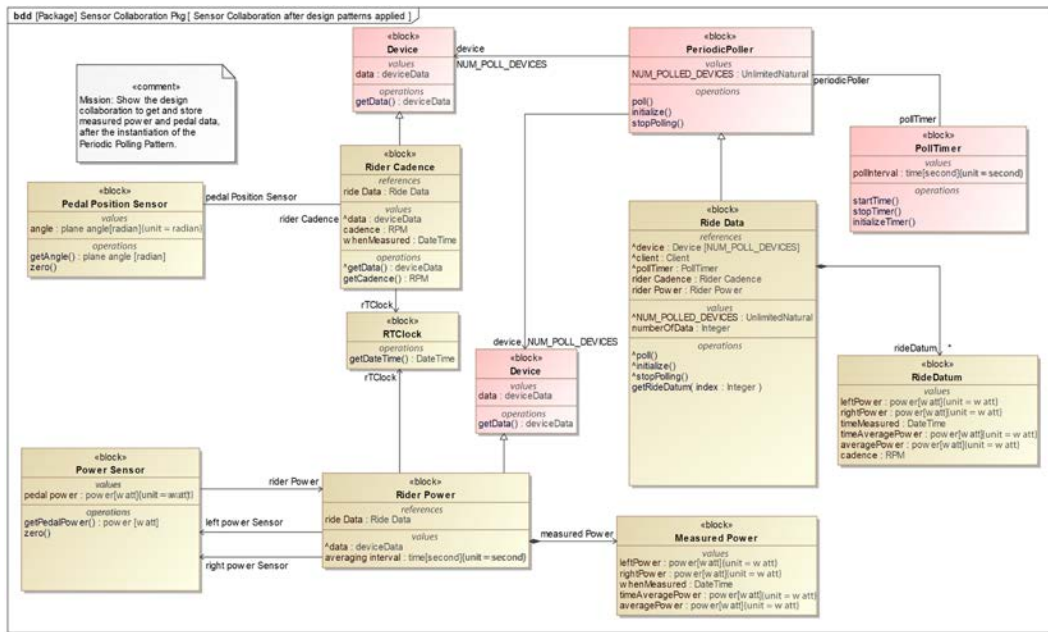


Figure 3.23: Instantiated design pattern

Validate design solution

At this point, we reapply the test cases used to verify the collaboration previously and additionally look to ensure our objectives (as stated by the trade study criteria) have been addressed.

Subsystem and component architecture

The **Subsystem** and **Component Architecture** view focuses on the identification and organization of system features into the largest-scale system elements – subsystems – their responsibilities, and their interfaces. In the *Architectural merge* recipe, we learned how the system features may be aggregated into a singular system block and to create merged interfaces and associated logical data schema. In this recipe, we'll look at how to identify subsystems, allocate functionality to those subsystems, and create subsystem-level interfaces.

So, what's a subsystem?

We'll use the definition from *Agile Systems Engineering* (see *Agile Systems Engineering* by Bruce Douglass, Ph.D., Morgan Kaufman Press, 2016):



Subsystem: An integrated interdisciplinary collection of system components that together form the largest-scale pieces of a system. It is a key element in the Subsystem and Component Architecture of the system.

Note that the definition includes the notion of a subsystem being interdisciplinary. This means that you will not define a “software subsystem,” an “electronics subsystem,” and a “mechanical subsystem,” although in a particular subsystem, one engineering discipline may dominate the design. Rather, subsystems are focused around tightly coupled requirements and coherent purpose and then implemented with some combination of engineering disciplines. The engineering contribution of an engineering discipline to a subsystem is referred to as a *facet* to distinguish it from subsystems and components. Discipline-specific facets are discussed in *Chapter 4, Handoff to Downstream Engineering*.

Modeling a subsystem in SysML

In SysML, a subsystem is just a block, although it is common to add a «Subsystem» stereotype. This can be found in the Cameo SysML library in the SysML::Non-Normative Extensions::Blocks package. It is common – although by no means required – to connect subsystems together by adding connectors between ports on the subsystems. In this book, we will use SysML proxy ports for that purpose.

More specifically, we will use proxy ports for dynamic connections, that is, connections that require the exchange of flows, such as control, energy, fluid, or data, whether discrete or continuous. For static connections, such as when mechanical pieces are bolted together, we will use associations between the blocks and direct connectors between the parts.

Block definition diagrams will be used to show the subsystem types and properties. This view is known as the system **composition architecture**. Internal Block Diagrams will show how the usages of the blocks are connected together to create an operational, running system. This latter view is sometimes known as the system connected architecture.

Choosing a subsystem architecture

It is important to remember that many subsystem architectures can achieve the same system-level functionality. The selection of a specific subsystem architecture is always an optimization decision.

Some of these optimization criteria are typically stated as “guidance” or “rules of thumb,” but they are really stating properties you’d like a good subsystem architecture to enhance. Some goals and principles for the creation of good subsystems are:

- Goals of subsystem and component architecture:
 - Reuse proven subsystem architectural approaches (patterns)
 - Support end-to-end performance needs easily
 - Minimize system recurring cost (cost per shipped system)
 - Maximize ease of maintenance
 - Minimize the cost of repair
 - Leverage team skills
 - Leverage existing technology and intellectual property (IP)
- Principles of subsystem and component architecture selection:
 - Coherence: Subsystems should have coherent intent and content
 - Contractual interaction: Subsystems should provide a small number of well-defined interfaces and services
 - Encapsulation: Tightly-coupled requirements and features should be in the same subsystem
 - Collaboration: Loosely-coupled requirements and features should be in different subsystems
 - Integrated teams: A subsystem is typically developed by a well-integrated interdisciplinary team
 - Reusability: Good subsystems can be reused in other, similar systems without requiring other contextual elements

Purpose

The purpose of creating this architectural view is to identify the subsystems and their responsibilities, features, connections, and interfaces. This provides a large-scale view of the largest-scale pieces of the system into which more detailed design work will fit.

Inputs and preconditions

The inputs include a set of requirements defining the overall system functionality. It is recommended that these requirements are organized into use cases, but other organizations are possible.

This input is a natural consequence of the recipes of *Chapter 2, System Specification*.

Additionally, the set of external actors – elements outside the system with which the system must interact – have been defined. This input is a natural consequence of the recipes of *Chapter 2, System Specification*.

Finally, an initial set of system functions, flows, and information has been identified through requirements or use case analysis and merged into a system block via the *Architectural merge* recipe or its equivalent.

Outputs and postconditions

At the end of this recipe, a set of subsystems have been identified, including their key system functions, data, and interfaces.

How to do it

The recipe workflow is shown in *Figure 3.24*:

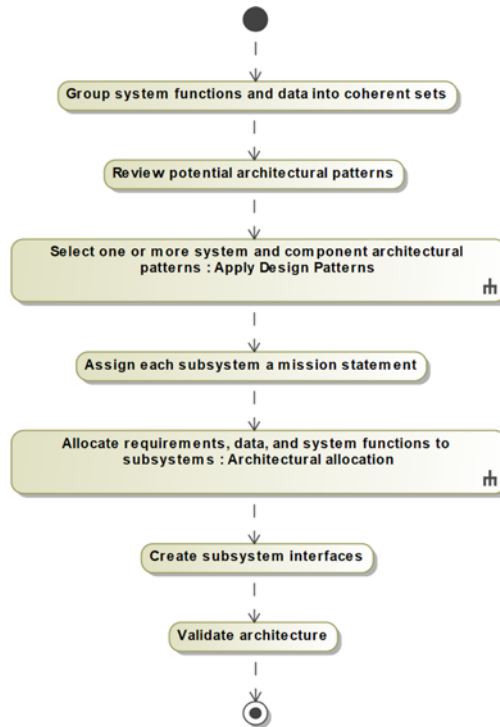


Figure 3.24: Create subsystem and component architecture

Groups systems functions and data into coherent sets

This task groups system features together by commonality. That commonality can be along different dimensions, such as a common flow source (sensors or system inputs), similar processing/transformation, similar use, reuse coherence (things that tend to be reused together), and dependability level (high-security or high-safety features tend to be grouped together).

These sets form the basis on which subsystems can be identified.

Review potential architectural patterns

As mentioned, there are many ways of organizing subsystems. The book, *Real-Time Design Patterns* by Bruce Douglass, Ph.D., Addison-Wesley Press (2003) identifies a number of popular subsystem patterns, such as the *Microkernel*, *Layered*, and *Hierarchical Control* patterns. Select one or more patterns that have the desired system properties.

Select a subsystem and component architectural pattern

This step is actually a reference to the *Pattern-driven architecture* recipe. This recipe details how to quantitatively select from a set of alternative technical approaches.

Assign each subsystem a mission statement

Once a set of subsystems is identified, each should be given a description that identifies the criteria for deciding whether or not it should host or contribute to a system capability.

Allocate requirements, data, and system features to subsystems

This step allocates system functionality, flows, and data to the subsystems. This is sometimes a little tricky and so is implemented by the next recipe in the book, *Architectural allocation*.

Create subsystem interfaces

System functionality will be provided by a collaboration of subsystems, and this requires those subsystems to communicate and coordinate. The subsystem interfaces specify the services and flows that will be used to accomplish this. They often tie in with the system-actor interfaces at one end or the other, but not necessarily. The system interfaces are created by the recipe *Specifying logical system interfaces* in the previous chapter.

Validate architecture

This step examines the resulting subsystem architecture to ensure that it can deliver the necessary system functionality and also that it achieves the optimization goals used to select the subsystem pattern.

Example

In this example, we'll create an architecture for the Pegasus system based on some of the use cases considered in previous recipes, such as Emulate Front and Rear Gearing, Control Resistance, Measure Performance Metrics, and Manually Adjust Bike Fit.

Groups systems functions and data into coherent sets

The functionality of the system can be grouped by common purpose and capability. A good way to do this is by looking at the system use cases, as we did in *Chapter 2, System Specification*. Similar to how we treat requirements and system functions, either a use case can be directly allocated to a single subsystem or it must be decomposed into subsystem-level use cases that can be so allocated. This decomposition is best represented with the «include» relation. Any subsystem-level use cases created in this way will be tagged with the «Subsystem» stereotype.

For the most part, elements added to the system block during the *Use case merge* recipe can just be moved to the appropriate subsystems. If you want to capture your reasons for the allocation, SysML provides a special kind of comment, «rationale», which you can anchor to the elements. You can even create element groups (in the right-click menu in Cameo for model elements). Element groups anchor elements and have a *criterion* property where you can add the rationale for the grouping.

To begin with, let's consider a set of use cases (Figure 3.25). Note that while this set encompasses a wide range of system capabilities, it doesn't include all capabilities. We recommend this work is done incrementally, and additional use cases – and even subsystems – can be added later as the need for them is identified:

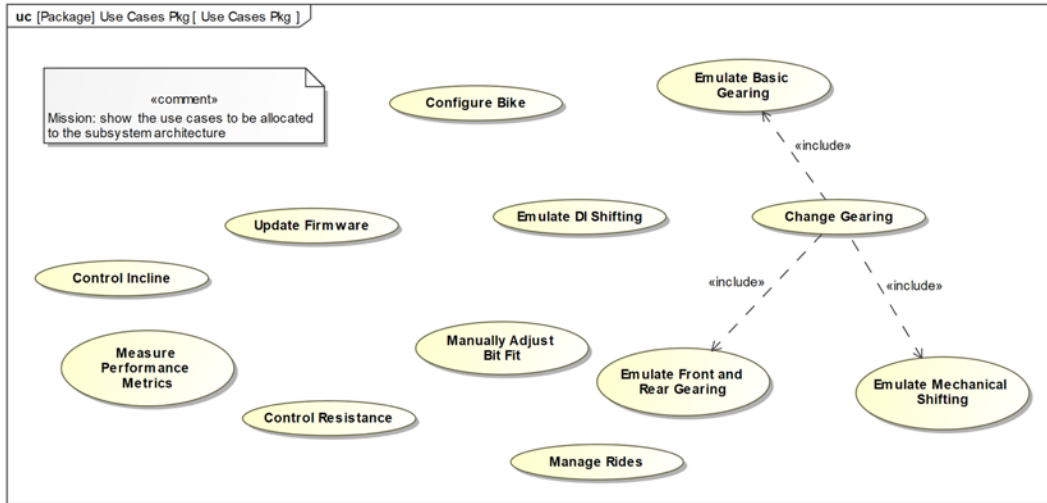


Figure 3.25: Set of use cases for subsystem allocation

Some of these use cases are likely to be allocated to a single subsystem. For example, the *Manually Adjust Bike Fit* use case might be allocated to a single Mechanical Frame. Others are likely to use multiple subsystems. For example, the **Control Incline** use case might be decomposed into subsystem-level use cases that map functionality to the **User Input** and **Mechanical Frame** subsystems. The exact decomposition will, of course, depend highly on the set of subsystems upon which we decide.

Review potential architectural patterns

There are many potential subsystem patterns from which to choose. From Buschmann et al., the *Model-View-Controller* and *Microkernel* architecture patterns seem potentially viable, although the former is more of a design collaboration-level pattern than an architectural design pattern. From the previously mentioned *Real-Time Design Patterns* book, the *Layered*, *Channel*, and *Hierarchical Control* patterns are possibilities.



See *A System of Patterns* by Buschmann, Meunier, Rohnert, Sommerlad and Stol, Wiley Press 1996 for further information.

Select a subsystem and component architectural pattern

Using the workflow from the previously presented *Pattern-driven architecture* recipe, we select the *Five-Layer Architecture Pattern* from the *Real-Time Design Patterns* book as our base pattern. We will modify it to use subsystems rather than packages, as in the original, and to replace **Abstract OS Layer** with **Abstract Common Services Layer**, as it seems more relevant to our design. See *Figure 3.26*:

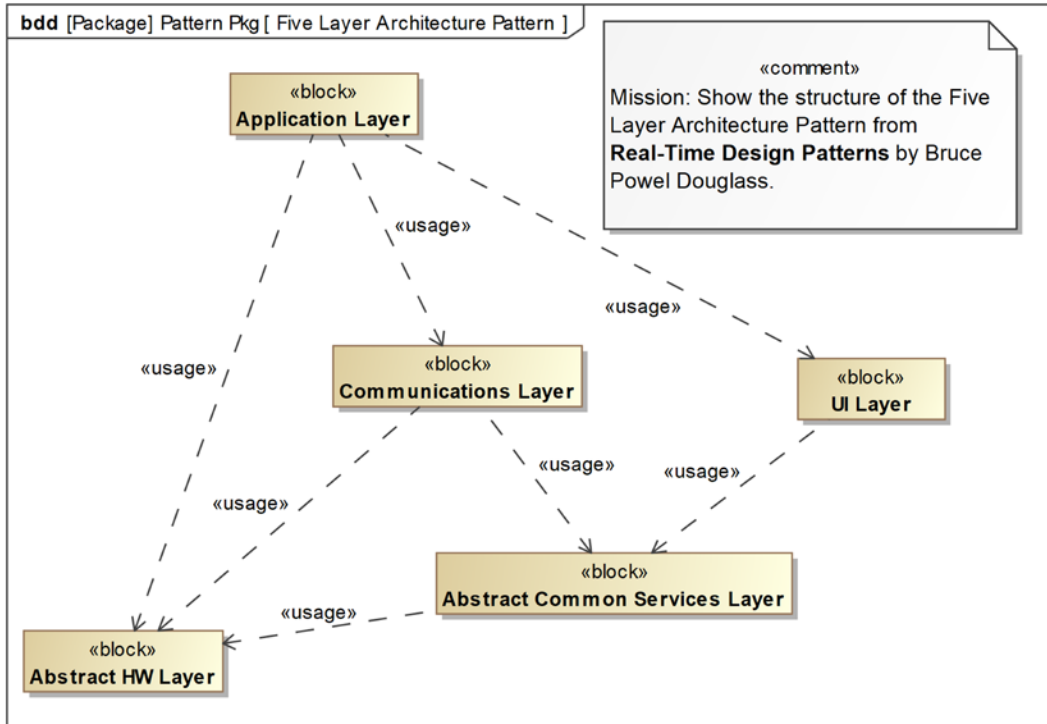


Figure 3.26: Modified Five-Layer Architecture Pattern

Each of the blocks in the pattern may have multiple instantiations for peers (such as multiple **Application Layer** elements) or may have an internal sub-subsystem structure (which we will refer to as simply “subsystems”).

Each of these blocks in the pattern has its own purpose and scope.

Application layer

This layer contains features at the application level, so system features (data and services) such as **Ride** and **Rider** metrics and properties would reside in this layer.

UI layer

This layer contains elements for the system user input. This includes the selection and display of current gearing, for example. Most display features actually reside on the third-party training apps, so the system itself has a limited UI.

Communications layer

The system requires Bluetooth and ANT+ communications protocols, setting up sessions, and so on. All those features reside largely in this layer.

Abstract common services layer

This layer contains common services such as data storage and data management and also electric power delivery.

Abstract HW layer

This layer focuses on electronic and mechanical aspects; frame adjustment, drive train, various sensors, and so on all reside here. This is likely to have peer instantiations and substructure.

The above discussion explains the pattern per se. Now, let's look at the pattern instantiation.

The set of six proposed primary subsystems is shown in *Figure 3.27* along with their high-level parts (also identified as subsystems).



In Cameo, «System» and «Subsystem» are defined in the **SysML::Non-Normative Extensions::Block** package in the SysML profile. I applied this «system» stereotype to the **Pegasus** block.

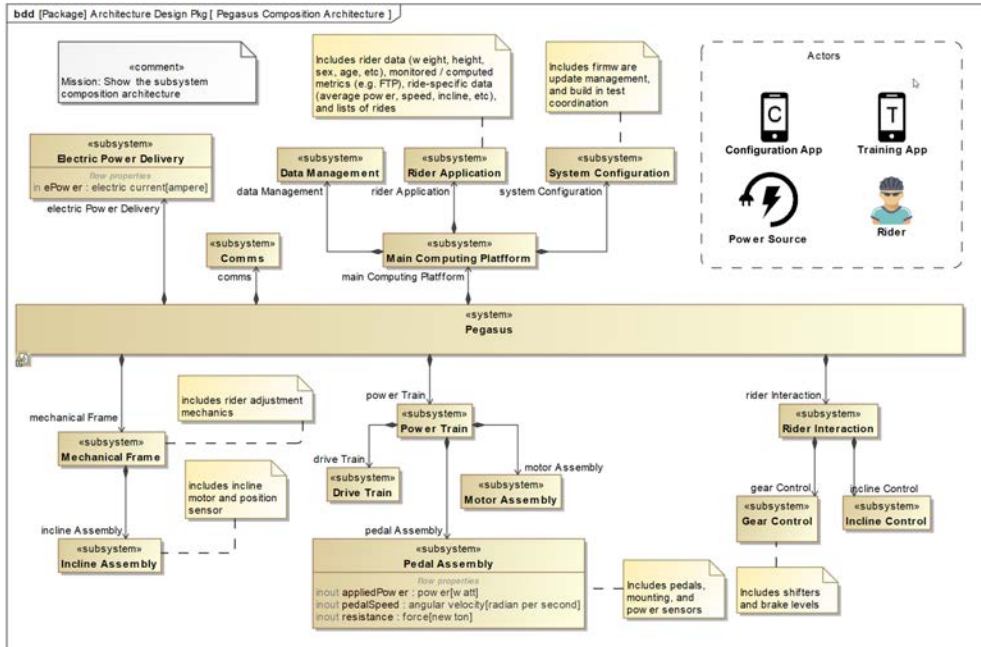


Figure 3.27: Pegasus proposed subsystems

We can create a “connected architecture” view showing how we expect the subsystems to connect. This will evolve as we elaborate the architecture and allocate functionality. *Figure 3.28* shows the connected architecture showing only the high-level subsystems. *Figure 3.29* shows the same view but with the internal structure we’ve identified so far. In both cases, I used Cameo’s *Symbol Properties* feature to change the line color and width for the electrical power connections to differentiate them. The connections are notional at this point and may change as we dive into the details of the architecture:

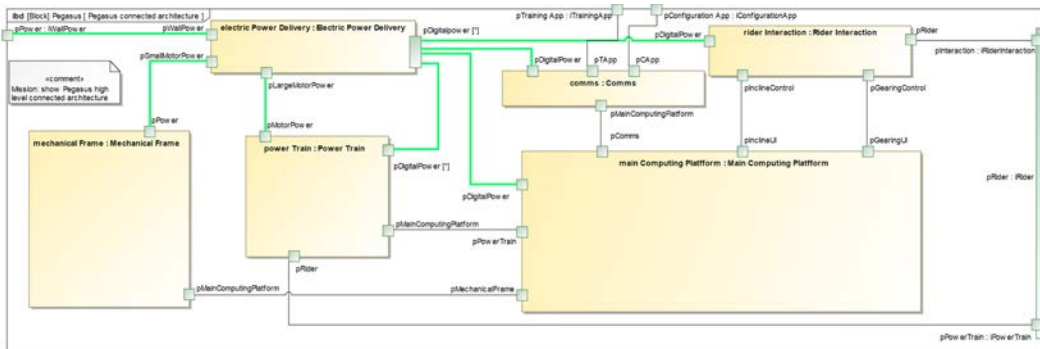


Figure 3.28: Pegasus high-level connected architecture

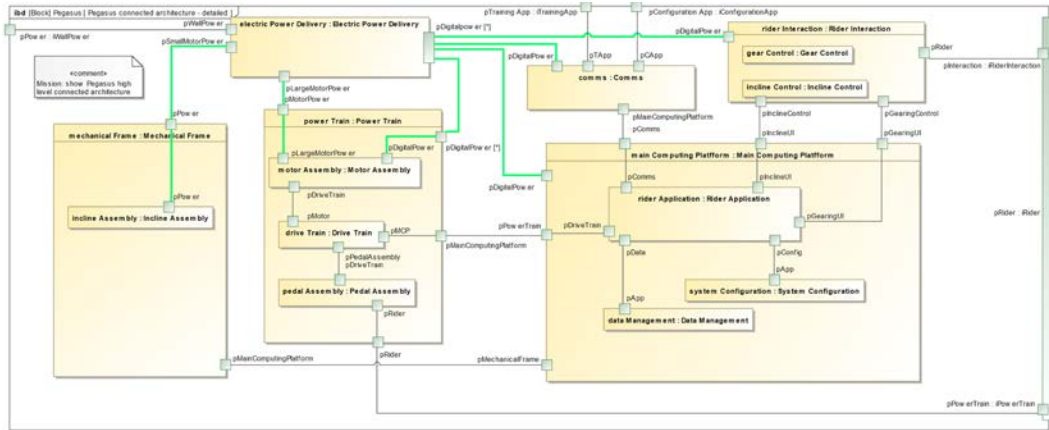


Figure 3.29: Pegasus detailed connected architecture

Assign each subsystem a mission statement

The mission statement for each subsystem is added to the **Description** field of the subsystem block. The mission statements for the high-level subsystems are provided in *Table 3.5*:

Subsystem	Mission Statement
Mechanical Frame	This subsystem provides the physical system structure and rider adjustment features – all of which are assumed to be strictly mechanical in nature – and also the inclination delivery/monitoring capability, which is planned to be motorized. This subsystem is envisioned to primarily consist of mechanical and electronic aspects.
Electric Power Delivery	This subsystem is responsible for receiving wall power and distributing managed electrical power to the other subsystems. This includes low amperage power for digital electronics, moderate amperage power for the motorized inclination capability, and high amperage power for the rider power and resistance. This subsystem design is envisioned to be dominated by electronic aspects.
Comms	This subsystem is responsible for all communications with external devices. At this time, this includes low-power Bluetooth and ANT+ communications protocols. This includes the physical layer through the network layer of the OSI protocol stack. The initial concept for this subsystem includes one or more smart communications processors to manage the communications activity. This subsystem is envisioned to consist primarily of digital electronics and software.

Main Computing Platform	The main computing platform contains the primary computing electronics and software for the system. It manages rider-level applications such as controlling and monitoring rides and sensor data, system configuration (including motor transfer function tuning and rider settings), and data management. This subsystem is envisioned to be primarily software and digital electronics hardware.
Rider Interaction	This subsystem provides the primary user interface (except for pedals) for the system, including shift and brake levers, and a display of the currently selected gearing. This system is envisioned to include mechanical, electronic, and software aspects.
Power Train	This subsystem manages the pedal assembly, the monitoring of rider power and pedal cadence input, and the creation of resistance to pedaling, under the direction of the Main Computing Platform subsystem. This system is envisioned to include mechanical, electronic, and software aspects.

Table 3.5: Subsystem missions

Allocate requirements and system features to subsystems

This task is complex enough to have its own recipe, *Architectural allocation*, later in this chapter. The example is elaborated on in that section.

Create subsystem interfaces

This task is complex enough to have its own recipe, *Creating subsystem interfaces from use case scenarios*, later in this chapter. The example is elaborated on in that section.

Architectural allocation

The recipes for functional analysis of use cases have multiple outcomes. The primary outcome is a set of high-quality requirements. The second is the identification of important system features – system functions, data, and flows. The third outcome is the identification of interfaces necessary to support the behavior outlined in the use case. This recipe focuses on allocating the first two of these to the subsystem architecture.

Purpose

The purpose is to detail the specification of the subsystems to get ready to hand off those specifications to the interdisciplinary subsystem teams for detailed design and development.

Inputs and preconditions

A set of requirements and system features have been identified, and a subsystem architecture has been created such that each subsystem has a defined mission (scope and content).

Outputs and postconditions

The primary outcome of this recipe is a specification for each subsystem that includes:

- System requirements allocated directly to the subsystem
- Subsystem requirements derived from system requirements, which are then allocated to the subsystem
- System features allocated directly to the subsystem
- Subsystem features derived from system features, which are then allocated to the subsystem

How to do it

This recipe is deceptively simple. The tasks are straightforward, although difficult to completely automate. This task can take a while to perform because there are often many requirements and features to allocate. The basic idea is that given the selected subsystem structure, either allocate a feature directly, or decompose that feature into subparts that can be so allocated. The workflow is shown in *Figure 3.30*:

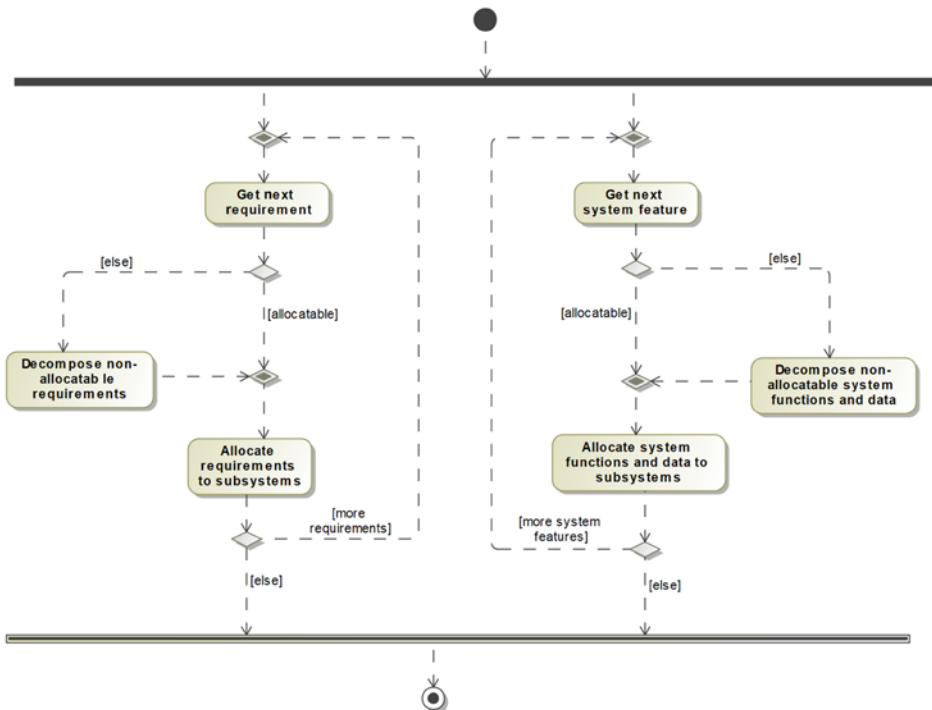


Figure 3.30: Architectural allocation

Decompose non-allocable requirements

Requirements can sometimes be directly allocated to a specific subsystem. In other cases, it is necessary to create *derived requirements* that take into account the original system requirements and the specifics of the selected subsystem architecture. In this case, create derived requirements that can be directly allocated to subsystems. Be sure to add «deriveReq» relations from the derived requirements back to their source system requirements.

Allocate system requirements

Allocate system requirements to the subsystems. In general, each requirement is allocated to a single subsystem, so in the end, the requirements allocated to each subsystem are clear. The set of allocated system requirements after this step is normally known as **subsystem requirements**.

Decompose non-allocable system functions and data

Some system features – which refer to operations, event receptions, flows, and data – can be directly allocated to a single subsystem. In practice, many cannot. When this is the case, the feature must be decomposed into subsystem-level features that trace back to their system-level source feature but can be directly allocated to a single subsystem.

Allocate system functions and data

Allocate system features to subsystems. That is, each system function now becomes a subsystem function allocated to a single subsystem, or it is decomposed to a set of subsystem functions, each of which is allocated. Similarly, system event receptions must be allocated to some subsystem and, often, this results in new subsystem event receptions being added elsewhere in the architecture to support the subsystem collaboration required to fulfill the system functionality needs. System flows and data must also be allocated to subsystems.

Example

In real system development, this recipe can take a substantial amount of time, not due to the complexity of the task but because of the large number of system requirements and features requiring allocation.

To keep the example manageable for the book format, we will focus on a subset of the requirements and system features identified in the use cases under consideration for this book:

req [Package] ReqsPkg CR Requirements				
<pre>«requirement» CR_requirement Id = "167" Text = "The system shall send pedal cadence to the associated training app, when connected."</pre>	<pre>«requirement» CR_requirement_1 Id = "168" Text = "The system shall store the rider weight for the computation of pedal resistance."</pre>	<pre>«requirement» CR_requirement_2 Id = "169" Text = "The system shall store the current gear persistently so that the last gear is retained across power cycles and resets."</pre>	<pre>«requirement» CR_requirement_3 Id = "170" Text = "The system shall monitor pedal position and speed."</pre>	<pre>«requirement» CR_requirement_4 Id = "171" Text = "The system shall compute pedal cadence."</pre>
<pre>«requirement» CR_requirement_6 Id = "172" Text = "The system shall measure force applied by the rider to the pedals."</pre>	<pre>«requirement» CR_requirement_7 Id = "173" Text = "The system shall provide time-filtering of power, supporting 0, 1-second, 3-second, and 5-second power averaging, settable by the rider."</pre>	<pre>«requirement» CR_requirement_8 Id = "174" Text = "The system shall send filtered power to the training app, if connected."</pre>	<pre>«requirement» CR_requirement_9 Id = "175" Text = "The system shall accept simulated incline from the connected training app throughout a training session."</pre>	<pre>«requirement» CR_requirement_10 Id = "176" Text = "The system shall store gear ratio from the gear setting."</pre>
<pre>«requirement» CR_requirement_11 Id = "177" Text = "The system shall accept road incline from the connected training app."</pre>	<pre>«requirement» CR_requirement_12 Id = "178" Text = "The system shall compute simulated drag based on computed rider inertia and simulated road incline."</pre>	<pre>«requirement» CR_requirement_13 Id = "179" Text = "The system shall compute simulated road speed based on computed inertia, current simulated speed and acceleration, and rider-applied force to the pedal."</pre>	<pre>«requirement» CR_requirement_14 Id = "180" Text = "The system shall compute and apply resistance to pedal movement based on simulated inertia, speed, acceleration and rider power input."</pre>	<pre>«requirement» CR_requirement_15 Id = "181" Text = "The system shall send computed speed and acceleration to the connected training app, if any."</pre>
<pre>«requirement» CR_requirement_21 Id = "182" Text = "The system shall update the training app with simulated bike speed at least 1.0 seconds"</pre>	<pre>«requirement» CR_requirement_22 Id = "183" Text = "The system shall update the training app with rider filtered power output at least 0.5 seconds"</pre>	<pre>«requirement» CR_requirement_23 Id = "184" Text = "The system shall update the training app with pedal cadence at least 1.0 seconds."</pre>	<pre>«requirement» CR_requirement_24 Id = "185" Text = "The system shall update the physics model frequently enough to provide the rider a smooth and road-line experience with respect to resistance."</pre>	<pre>«requirement» CR_requirement_25 Id = "186" Text = "The system shall send current watts per kilogram to the training app for the current power output at least every 1.0 seconds."</pre>

Figure 3.31: Requirements selected for allocation

Figure 3.31 shows the set of requirements for a single use case, Compute Resistance. In the interests of brevity, we will decompose and allocate a subset of these requirements.

Decompose non-allocable requirements

As mentioned, some requirements must be decomposed into derived requirements before allocation. This is because these requirements are partially met by different subsystems. In this case, a set of derived requirements must be created that are collectively equivalent to the original requirement but are of appropriately narrow focus so that they can be directly allocated. This is modeled with the «deriveReq» relation, going *from* the derived requirement *to* the originating requirement:

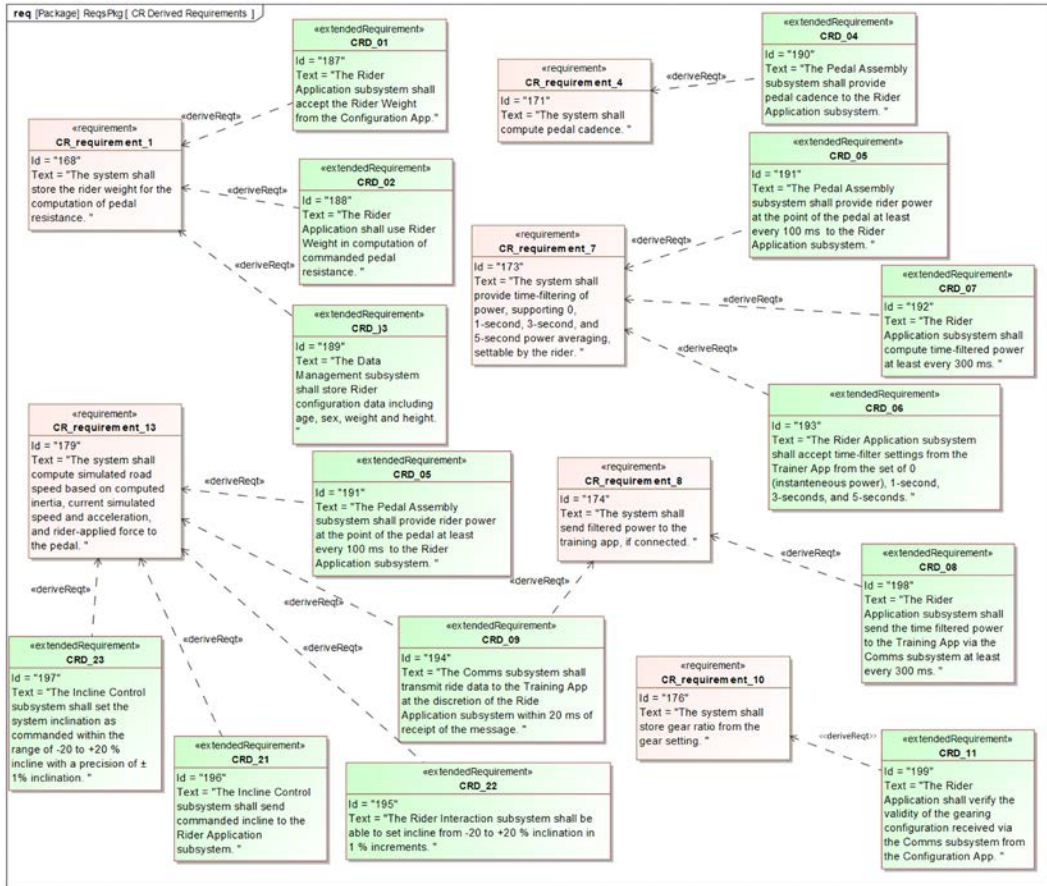


Figure 3.32: Some derived requirements

Figure 3.32 shows some of the requirements created during the derivation process. I color-coded the original system requirements and the derived requirements as a visual aid. Again, note the directions of the relations.

The figure shows a number of the system requirements decomposed into subsystem requirements. For example, **CR_requirement_7** states “The system shall provide time-filtering of power, supporting 0, 1-second, 3-second, and 5-second power averaging, settable by the rider.” This results in three derived requirements:

- **CRD_05:** The Pedal Assembly subsystem shall provide rider power at the point of the pedal at least every 100 ms to the **Rider Application** Subsystem.

- **CRD_06:** The **Rider Application** subsystem shall accept time-filter settings from the **Trainer App** from the set of 0 (instantaneous power), 1-second, 3-second, and 5-second).
- **CRD_07:** The **Rider Application** shall compute time-filtered power at least every 300 ms.

It can be cumbersome to show derived requirements diagrammatically for large numbers of derived requirements. An alternative is to construct a matrix showing originating and derived requirements. Such a matrix is shown in *Figure 3.33*:

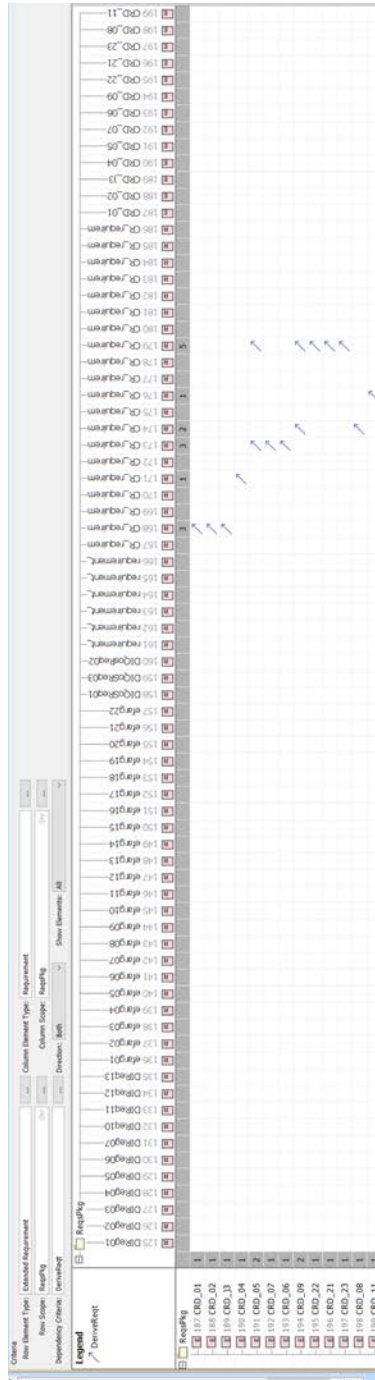


Figure 3.33: Derived requirements matrix

As we create new subsystem requirements, we need a place to put them. In this example, we'll create **SubsystemRequirementsPkg** under **RequirementsAnalysisPkg::RequirementsPkg** for this purpose. We may later decide to create a package per subsystem within **SubsystemRequirementsPkg** if the number of derived requirements grows large enough to justify it.

Allocate system requirements

Allocation means “assignment” in the sense that a subsystem is expected to implement its allocated requirements. This can be done diagrammatically, but in this case, we'll do this using an allocation matrix, as shown in *Figure 3.34*. In this matrix, the allocation relation goes *from* the subsystem *to* the requirement. The matrix itself is located in **DesignSynthesisPkg::ArchitecturalDesignPkg** in the model, but other locations are possible. The matrix layout used *Block* as the *from* element type, *Requirement* as the *to* element type, and *Allocation* as the cell element type.

The actual allocation is performed by walking through the requirements and creating an allocation relation from the subsystem to the requirement. *Figure 3.34* shows the system and derived requirements and their allocation for this subset:

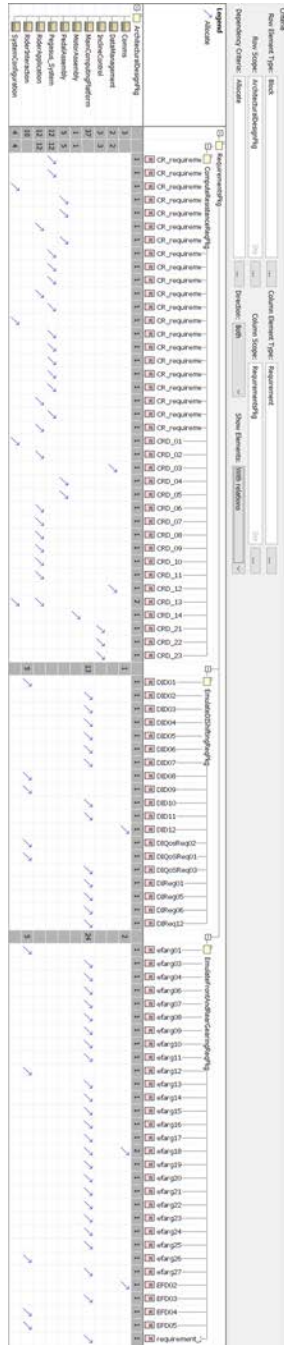


Figure 3.34: Subsystem requirements allocation matrix (subset)

Note that the matrix uses the convention that system requirements requiring decomposition are shown as allocated to the **Pegasus** block in *Figure 3.34*, while system requirements that are directly allocable are allocated directly to the subsystem. The requirements derived from the decomposed requirements are allocated to individual subsystems. In the figure, system requirements are all named **CR_requirement_<#>** (the CR indicating the **Control Resistance** use case), while the derived requirements are named **CRD_<#>**.

Also note that, for the most part, the high-level subsystems – that is, those that contain subsystems themselves – don't have allocated requirements, although their internal parts do. This is not uncommon. Thus, the **Main Computing Platform**, **Power Train**, and **Rider Interaction** subsystems don't have requirements directly allocated to them, but each contains nested subsystems that do.

Finally, note that the matrix is sparsely populated. This is because it only shows requirements from a single use case. As use cases are added, this matrix will become much more densely populated.

The matrix is shown with the subsystems as the rows because it fits better into the book format. Normally, the systems would be the columns because I prefer to have matrices with more rows than columns for readability within the modeling tool.

Allocate system functions and data

For the system features, we'll use the features we allocated to the system object during the *Architectural merge* recipe earlier in this chapter. The set of system features is shown in *Figure 3.35*. Note the Pegasus block is shown three times with different features shown for readability reasons.

The view on the left shows flow and value properties, while the views on the right show operations and event receptions:

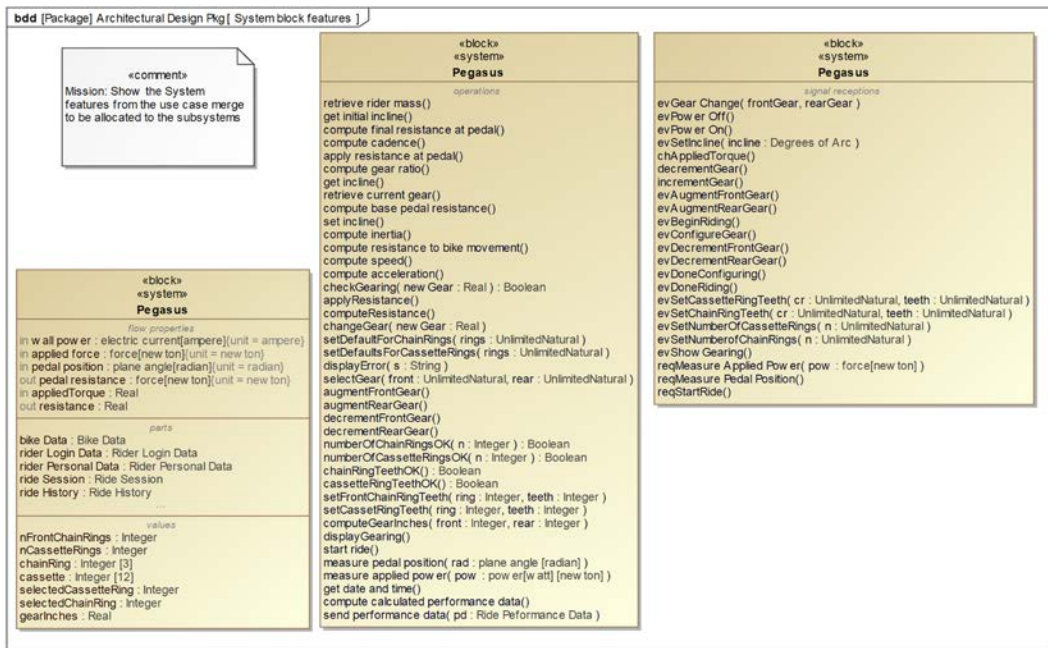


Figure 3.35: System features for allocation

The standard way of performing the allocation following the architectural merge is to *move* (in Cameo, drag and drop) the system feature to the appropriate subsystem. Any system features that remain after that initial pass are then decomposed and allocated (the last step in the recipe).

It may happen that a system feature cannot be allocated, not because it must be decomposed but because there is no appropriate subsystem. In this case, the missing subsystem must be added. It can also happen that at the end of the entire recipe, there are subsystems to which no requirements or features have been allocated. This simply means that subsystem isn't being used in the current iteration but might be in future iterations. Generally speaking, at the end of the recipe, all subsystems should have features allocated to them or serve another purpose (such as organizing and containing smaller subsystems).

Figure 3.36 shows the result of this step. Several of the subsystems have value properties, event receptions, and operations allocated to them:

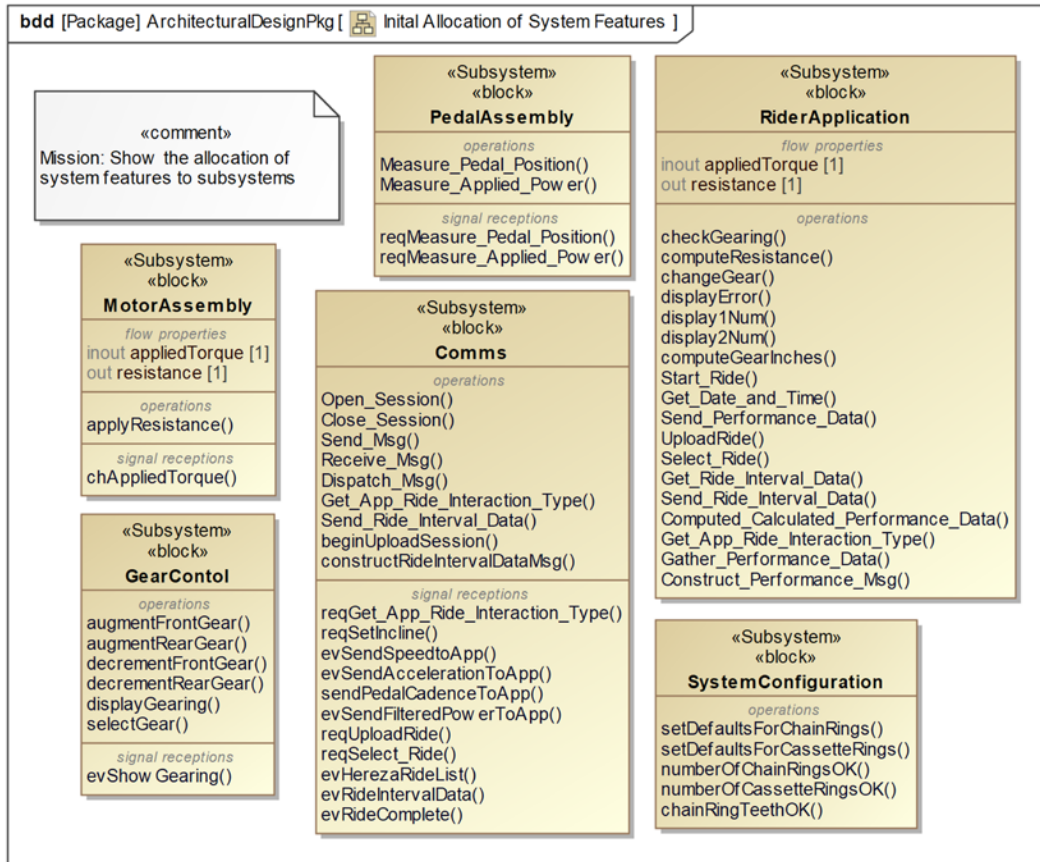


Figure 3.36: Initial allocation of system features

Decompose non-allocable system functions and data

In this example, the only system feature not directly allocated is the operation **Send_Performance_Data**. We have decomposed this service into:

- **Gather_Performance_Data** (allocate to the **Rider Application** subsystem)
- **Construct_Performance_Data_Msg** (allocate to the **Rider Application** subsystem)
- **Send_Msg** (allocate to the **Comms** subsystem)

While we're doing that, we note that we also need **Receive_Msg** and **Dispatch_Msg** system functions for the **Comms** subsystem.

To then allocate these created system features, leave the decomposed system feature in the system block and directly add the created features to the appropriate subsystem. Then add relations from the original feature to the derived features. In this case, we'll use the «derive» relation (as opposed to «deriveReq» since these are system properties, not requirements). This is shown graphically in *Figure 3.37*. If you look closely, you can see the relations not between the blocks themselves, but the operations shown:

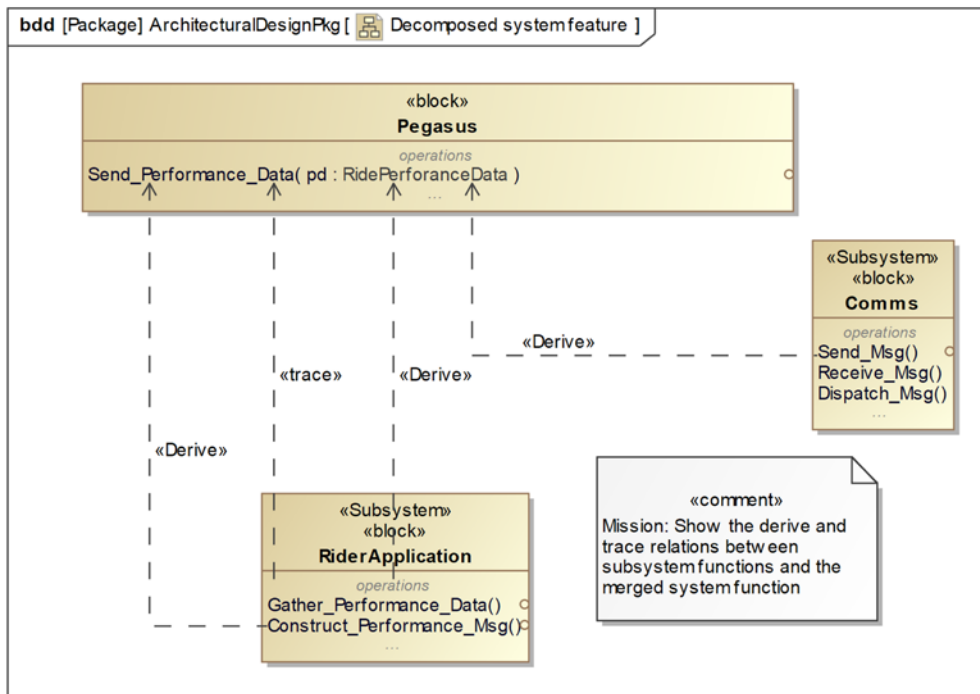


Figure 3.37: Some derived system features

An easier way to visualize these relations is with a dependency matrix, as shown in *Figure 3.38*:

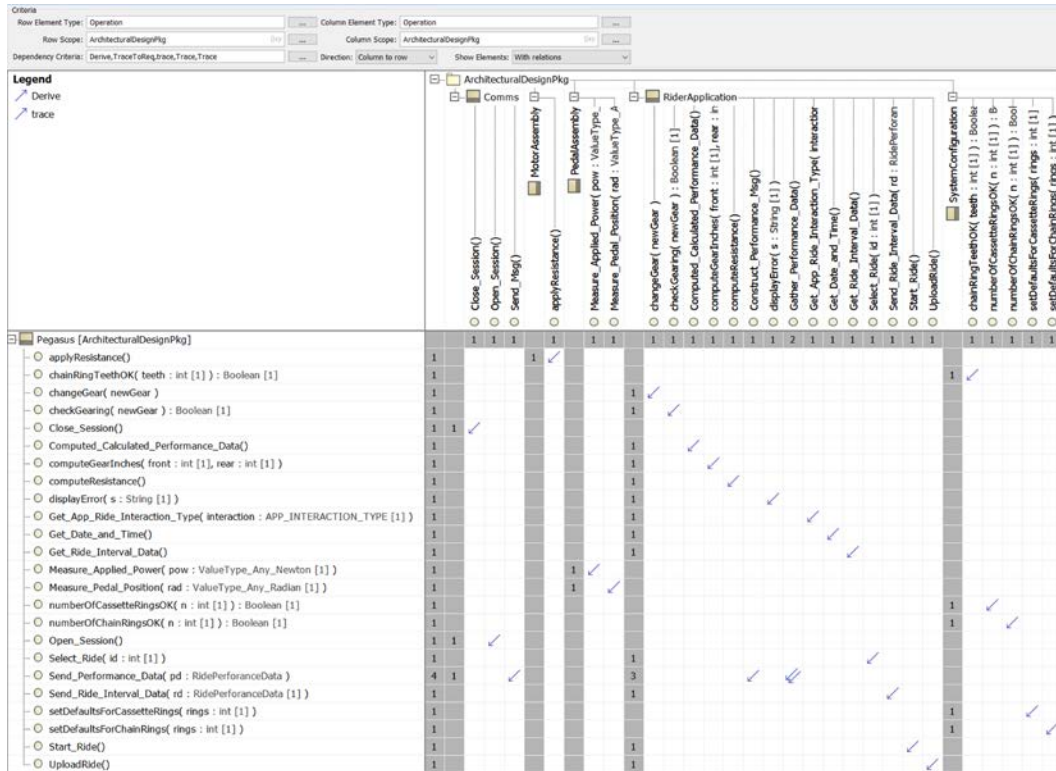


Figure 3.38: Dependency matrix showing derived relations between operations

Creating subsystem interfaces from use case scenarios

There are many methods by which subsystem interfaces can be created. For example, a common approach is to refine the black box activity diagrams from the use case analyses into so-called *white box* activity diagrams with activity partitions representing the subsystems. When control flows cross into other swim lanes, the flow or service invocation is added to the relevant subsystem interface. Another common approach is to do the same thing but use the use case sequence diagrams rather than the activity diagrams. The advantage of these approaches is that they tie back into the use case analysis. It is also possible to create the interfaces *de novo* from the allocation of system features to the subsystems.

This recipe focuses on using sequence diagrams in the creation of the system interfaces. One advantage of this approach is that this approach can leverage sequence diagrams created from the execution of the use case models that may not appear on the activity diagram. Further, many engineers use the activity diagrams as the starting point but the state machines become the normative specification of the use case; that is, the activity diagrams are not fully fleshed out but the state machines are. In such cases, this approach is superior to basing the interface definitions on the incomplete activity diagrams.

Purpose

The purpose of this recipe is to develop the interfaces between the subsystem so that the subsystem teams can design with an understanding of the flows and services they must provide to and require from other subsystems. These interfaces are still *logical* and reference essential aspects and are largely technology independent. These interfaces will be refined into physical interfaces that explicitly expose technical details in recipes discussed in the next chapter.

Inputs and preconditions

The inputs include the identified set of subsystems to which system features have been allocated. In addition, each referenced use case has a set of black box sequence diagrams showing the interaction of the use cases and the actors.

Outputs and postconditions

The primary outcome of the recipe is the set of interfaces (as interface blocks) defining the logical flows and service invocations between the subsystems and between the system and the actors. A part of that definition will be the logical data schema for flows and service parameters.

How to do it

This recipe is straightforward, although it can be a little tedious to perform by hand in many scenarios. *Figure 3.39* shows the workflow.

In any case, the outcome is a set of subsystem interfaces that support the necessary interactions for the architecture to realize the use case behaviors:

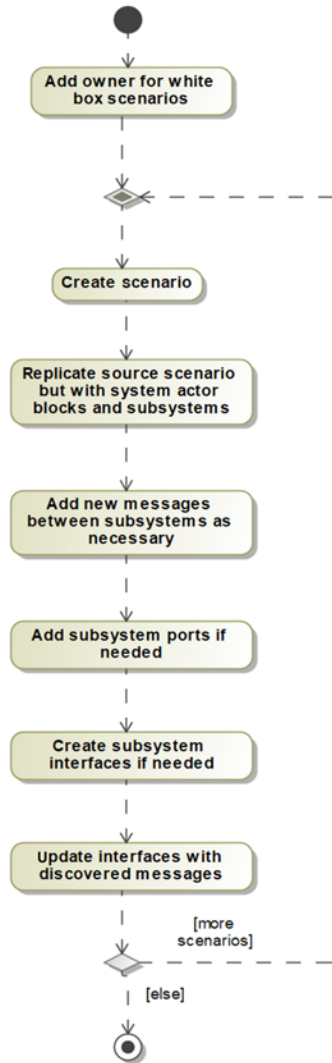


Figure 3.39: Creating subsystem interfaces from use case scenarios

Add owner to hold white box scenarios

In Cameo, the owner of a sequence diagram must be a classifier (unlike Rhapsody, which allows packages to directly own sequence diagrams).

In this case, the logical choice for the owner of the architectural white box scenarios is the architectural system context block that has the system and the architectural actor blocks as parts. This is the System Context block shown in *Figure 1.29* in the *Architecture 0* recipe.



We will use the term **scenario** to be synonymous with **interaction** in this recipe description since we have modeled each scenario of interest as a single sequence diagram in most cases.

Create scenario

Either copy an existing interaction in the browser or create an entirely new sequence diagram for the owner. If you create a new sequence diagram for the context block, Cameo will pop up a dialog asking you to select which parts you'd like to include. This will be done for each scenario in every included use case. I like to ensure the same ordering of all the lifelines for all the scenarios, so I generally prefer to copy the interaction and its own sequence diagram to retain the same ordering. This makes it easier to compare sequence diagrams later.

As a part of this step, you can add a «refine» relation from the new architectural scenario (interaction) back to the original use case scenario (interaction) it is elaborating. In Cameo, it is a simple task to create a matrix showing which architectural interactions elaborate which functional analysis interactions based on this relation.

I find it useful to add **_WB** (for “white box”) to the name of the new sequence diagram to ensure that it isn't confused with its source.

Replicate the source scenario but with the real actors and subsystems

In this step, we will recreate the messages from the actor block to the use case and from the use case to the actor block in the new scenario. What will be different is that the *architectural* actor block will be used, and the message termination inside the system will be some specific subsystem rather than the use case or use case block.

Add necessary subsystem interaction messages

This is a “magic step” in the recipe and requires the engineer to decide how the subsystem interaction should occur to realize the use case messages at the black-box level. The engineer must consider the requirements, data, and services allocated to the subsystems and create messages to support an interaction that will use them to achieve the system-level objective.

Create subsystem ports if missing

If there is at least one message between a subsystem and another, or between a subsystem and an actor, then the subsystem must have a port to support that message exchange. If the message is between a subsystem and an actor, then it will require a port on the system block. In this case, the subsystem's port will connect to the system's port, and then in turn, the system's port will connect to the actor's port. It is a good idea to set each block's **isEncapsulated** property to **true**, as this enforces the rule that a block's parts may only be accessed through the composite block's properties. This is especially true for the system block. Unfortunately, it doesn't appear to be possible to set this as a project option, so it must be done separately for each block. Once set, the Cameo rules validator will detect and report any such violations (see the **Analyze > Validation > Validate** menu).



The above is true for real interfaces that support actual messaging. When the “message” is displayed on a screen, or received by the system by the user pressing a button on the system, then that interaction is “virtual” and no real system interface needs to be created in the architecture. This is typical of user interfaces owned by the system.

Create subsystem interfaces if missing

Once the white box sequence diagram is updated, interfaces can be created wherever two subsystems exchange messages. This means that in the architecture, they will relate via a connector. I recommend this connection is done via ports typed by interfaces that support those messages. These interfaces will include any operations, event receptions, and flow properties that cross the system boundary.

Update interfaces with messages

In the previous step, we added interfaces whenever there was at least one message exchanged between a pair of subsystems or a subsystem and an actor. In this step, we will add the messages as signal receptions. We will stereotype them as «directedFeatures» in the interface blocks if we're using proxy ports.

This cycle is repeated for all functional analysis scenarios included in the architectural design.

Example

For this example, we will focus on the scenarios from two use cases: **Measure Performance Metrics** and **Control Resistance**. These scenarios show the interactions between actor blocks (acting as proxies for the actual actors) and the selected use case.

Add owner to hold white box scenarios

If the model doesn't already have a **System Context** block that has composition relations to the Pegasus and the architectural actor blocks, we will do it now. This block will own the white box interactions, which will in turn own the sequence diagrams.

Create scenario

The first scenario is **Measure Performance Metrics Black Box View 1**, which was derived from the use case activity diagram. The original sequence diagram is shown in *Figure 3.40*. We show it here as reference; we won't modify it, but we will replicate it in the upcoming steps:

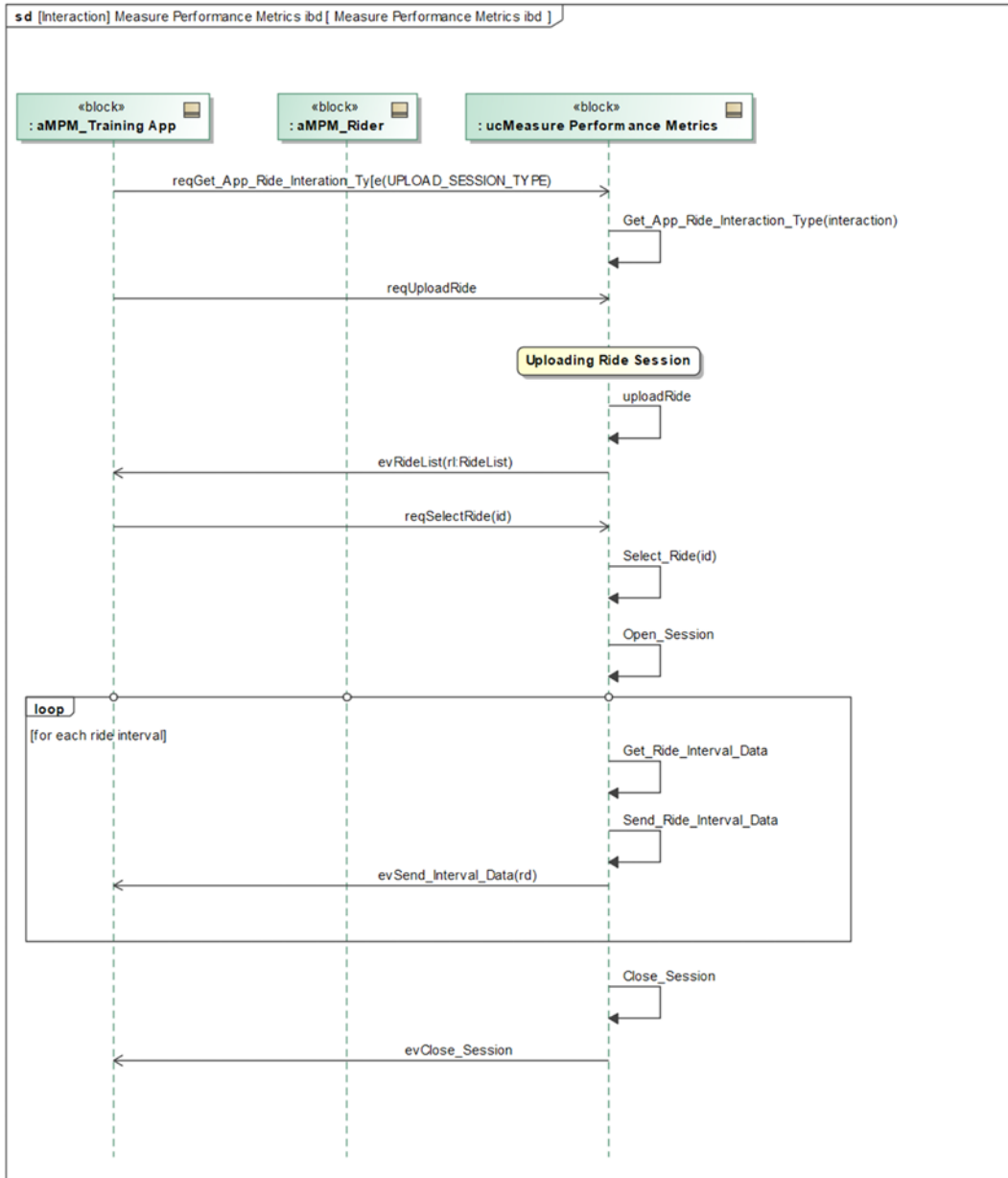


Figure 3.40: Original use case scenario

In the containment tree, also open the newly created sequence diagram showing all the subsystems as lifelines. In Cameo, you can right-click a diagram tab and select **New Vertical Group** to show diagram windows side by side.

Replicate the original scenario but with system actor blocks and subsystems

The easiest way to work is to open the new and original sequence diagrams side by side (Figure 3.41):

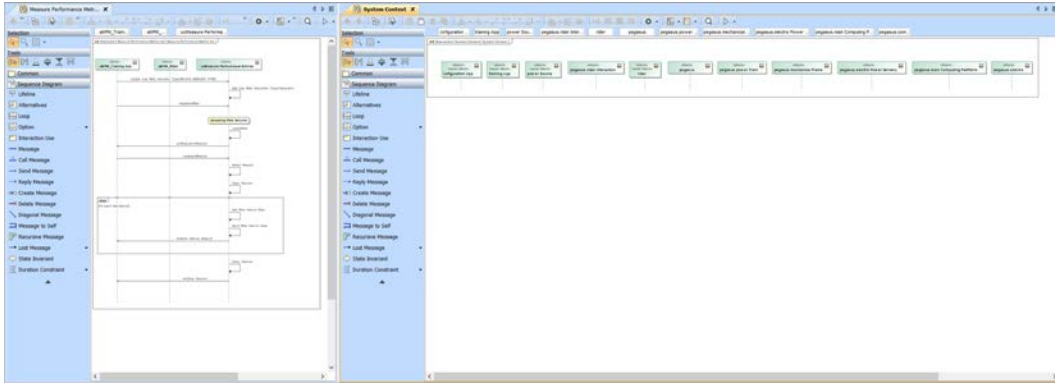


Figure 3.41: Ready to replicate messages

This step is easy; using the original scenario as guidance, add *the same messages* to the new scenario. The slightly tricky part is deciding which subsystem should be the sender or receiver lifeline. For this scenario, the lifeline receiving and sending the events to and from the **Training App** lifeline will be the **Comms** subsystem. Deciding where the “messages to self” go is only a little harder. If the services have to do with communications per se, then they will be put on the **Comm** subsystem lifeline. In this case, all the other messages to self will be put on the **Main Computing Platform** lifeline.

The sequence diagram now looks like this (Figure 3.42):

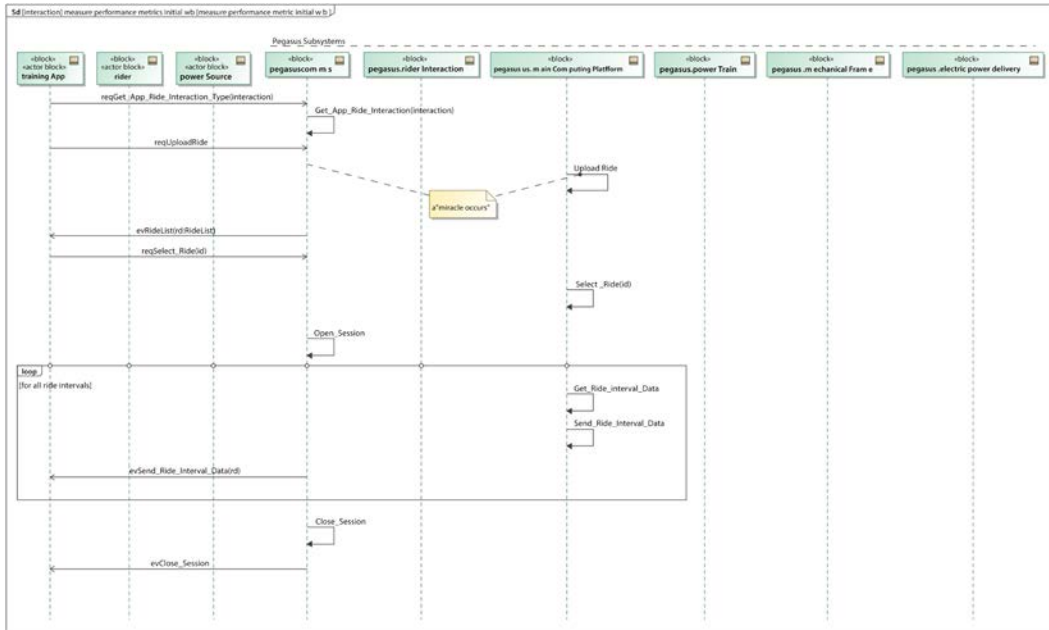


Figure 3.42: Scenario replicated

You will note that a miracle occurs between the **Comm** subsystem receiving a message and the **Main Computing Platform** doing something. That is because we are missing the messages between those subsystems; their identification and elaboration are outcomes of this recipe.

Add necessary subsystem interaction messages

As we go down the sequence, we need to add messages between the subsystems so that we can coordinate their actions and collectively perform the overall system interaction with the actors. It also happens that we may discover discrepancies in the original interaction, such as missing messages to or from an actor or data that must be passed with an existing message. Since we are defining the actual logical messages within the architecture, we will address those defects here.

The result of this step is a white box sequence diagram that shows the interaction of the subsystems with other subsystems and the actors. Some of these messages will be realized by the operations allocated to the subsystem during the *Architectural allocation* recipe earlier in this chapter. Others will be new.

Figure 3.43 shows the previous sequence diagram elaborated with the messages between the subsystems to complete the architectural flow. If we then repeat this for all the scenarios under consideration, we can use that to form the basis of the architectural interfaces:

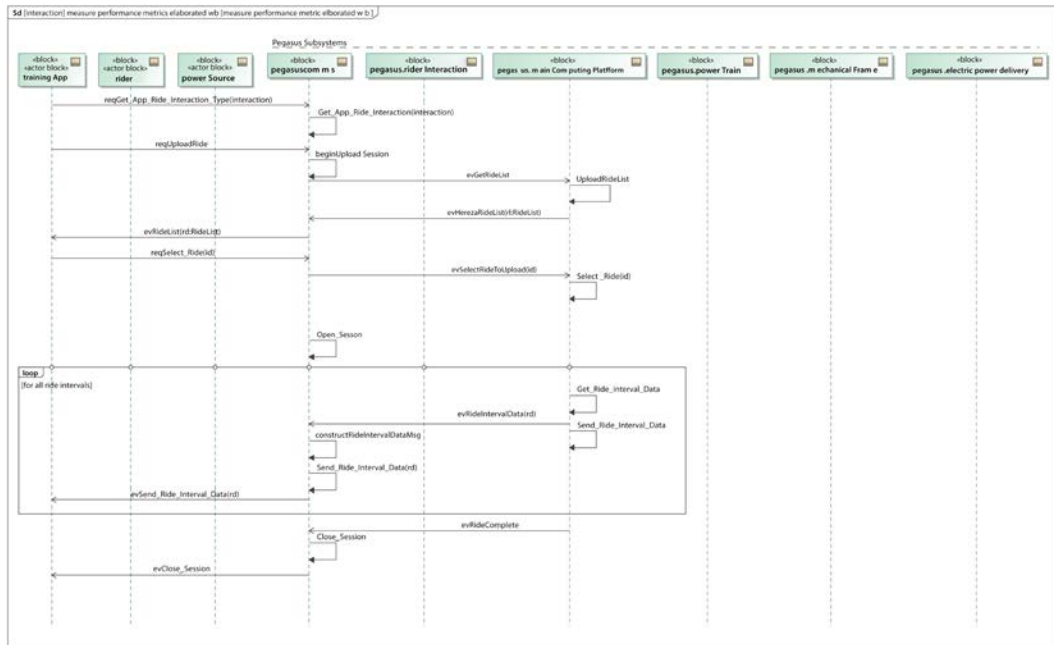


Figure 3.43: White-box scenario measure performance metrics

Repeating that elaboration process for the other three scenarios – Control Resistance, Process Pedal Inputs, and Execute Physical Model – results in Figure 3.44, Figure 3.45, and Figure 3.47:

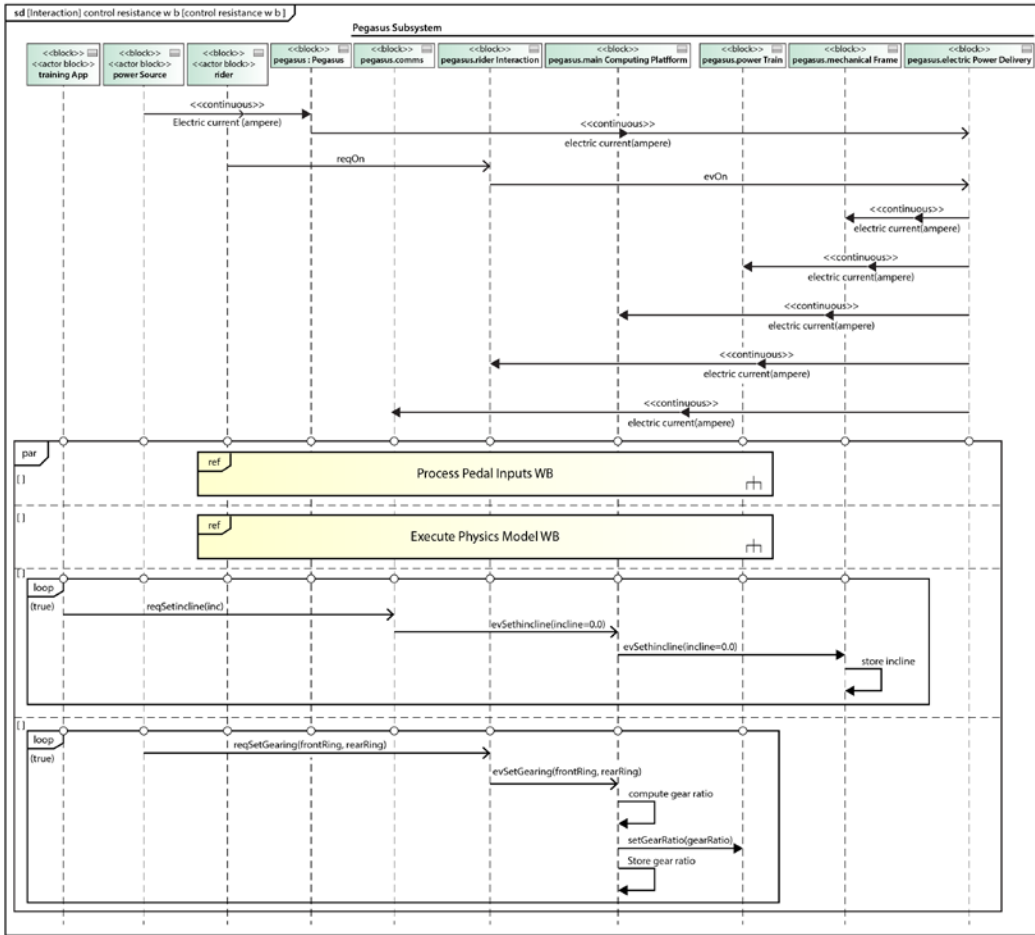


Figure 3.44: White-box scenario control resistance

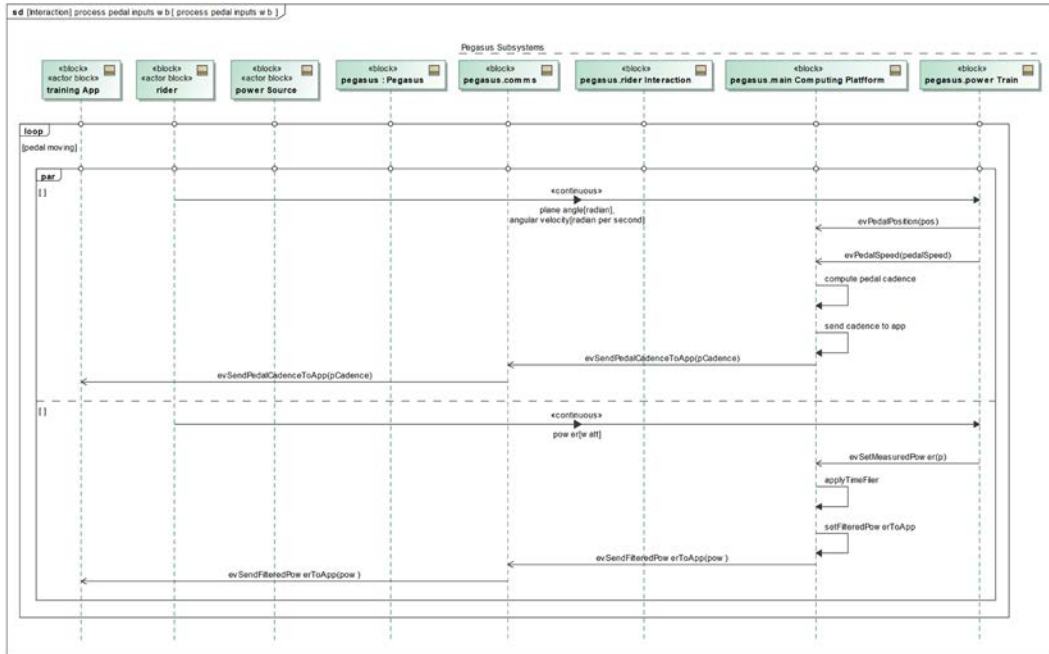


Figure 3.45: White-box scenario process pedal inputs

Note the item flows on a few of the messages in *Figure 3.45*. This is done in Cameo by selecting the message and clicking on the **item flow manager** icon in the popup quick toolbar. However, in Cameo, this can only be added when the flow item is already defined in the model as occurring between the elements represented by the lifelines. One natural place to do that is in the internal block diagram, as shown in *Figure 3.46*:

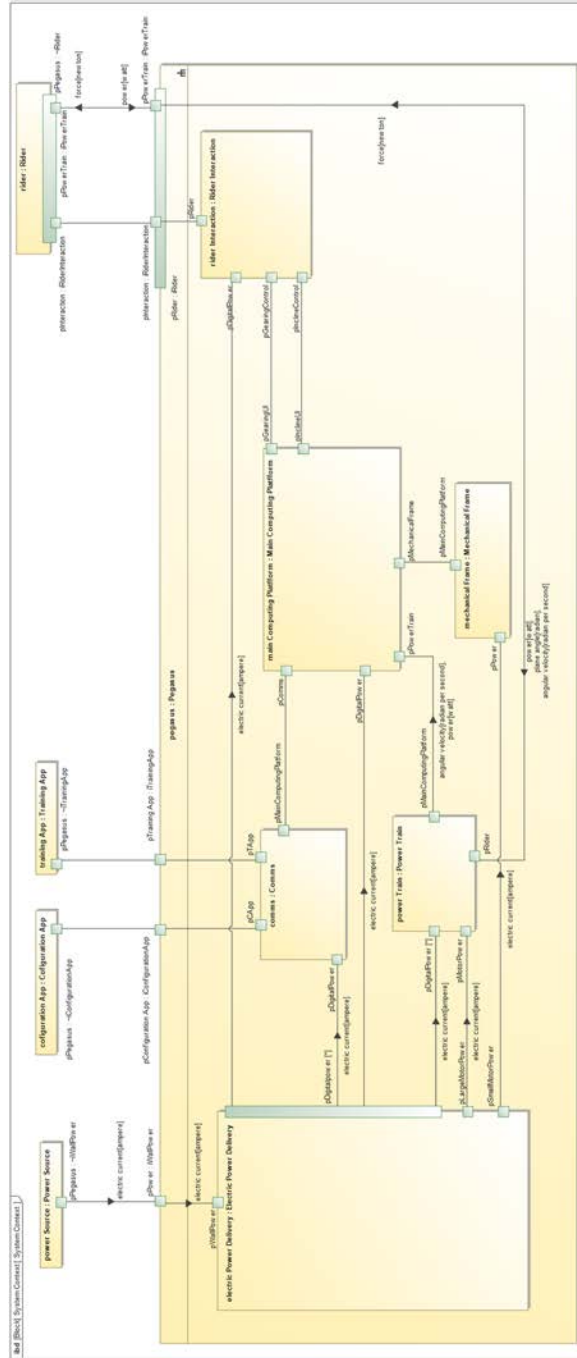


Figure 3.46: IBD with flow items

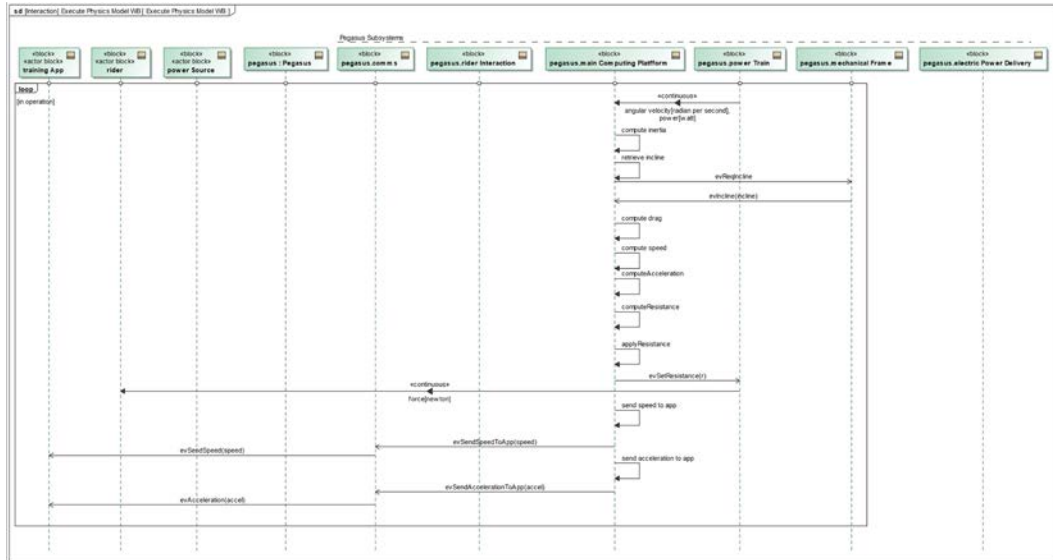


Figure 3.47: White-box scenario execute physics model

Create subsystem ports if missing

Wherever at least one message exists between elements in the white box scenarios, there must be a port pair supporting that message exchange. The ports necessary to support the white box scenarios are shown in *Figure 3.46*.

Ports are named with a **p** followed by the name of the target element to which they will connect. The **Comms** port that will connect to the **Main Computing Platform** subsystem is thus named **Comms.pMainComputingPlatform**. The ports for delivery of power from the **Electric Power Delivery** subsystems are flow ports because they each only deliver a singular flow without the invocation of services.

Create subsystem interfaces if missing

In the elaboration of the white box scenarios in the example, we have identified interactions between a number of subsystems and added port pairs to support the messaging.

Interface blocks are named with an **i** followed by the name of the unconjugated user of the interface block, an underscore, and the name of the conjugated side of the connection. Consider the interface block is named **iMainComputingPlatform_Comms**; its naming indicates that the **MainComputingPlatform.pComms** port uses the interface block in its normal form but the **Comms.pMainComputingPlatform** port uses the interface block in conjugated form.

Update interfaces with messages

This step adds the messages sent across the ports to the interface blocks. These messages are the messages on the white box sequence diagrams. Note that, by convention, we are using asynchronous messages between the subsystems and between the system and the actors to define these logical interfaces. Since these are proxy ports, SysML requires us to stereotype these messages in the interface blocks as «directedFeatures». Once this stereotype is applied, the **feature direction** property appears and the message direction can be specified as **provide**, **required**, or **providedrequired**.

In this example, *Figure 3.48* shows interface blocks and their services identified from the scenarios examined in this example:

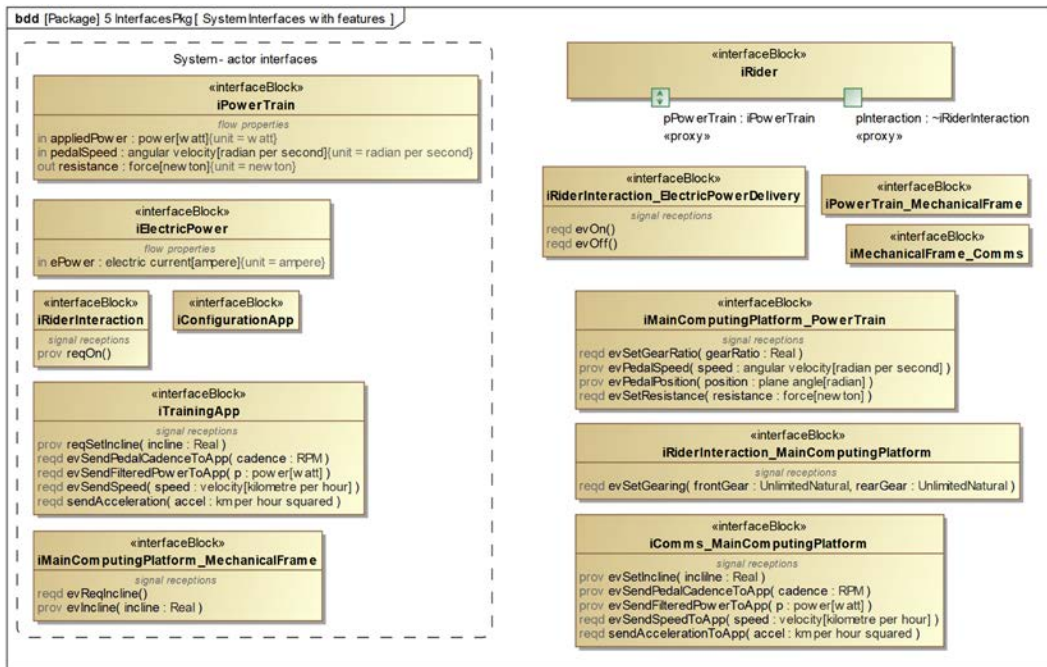


Figure 3.48: Derived subsystem interface blocks

Specializing a reference architecture

In this recipe, we will discuss the first of two approaches for using a reference architecture.

What is a reference architecture?

In *Chapter 1, Basics of Agile Systems Modeling*, we defined architecture as “the set of strategic design optimization decisions for the system.” The use of the word *strategic* here is important; these are design decisions that affect most or all subsystems and impact the overall performance and structure of the system. In the previous discussions, we went on to identify the six key views of architecture: the subsystem and component view, the concurrency and resource view, the distribution view, the data view, the dependability view, and the deployment view.

In the *Pattern-driven architecture* recipe in this chapter, we talked about how an architecture is an instantiation of patterns. In fact, a systems architecture is an integration of one or more patterns in each of the architectural viewpoints.

A reference architecture is an extension of this concept. A reference architecture is a *pattern writ large*; it is a combination of a set of patterns to define a master pattern for the architecture of a type of system, such as perhaps an aircraft, satellite, or medical ventilator. That means that a reference architecture generally cannot be used as-is. It must be instantiated for use to create specific system designs. In this recipe, I will call that process *specializing the reference architecture*.

Why would I use a reference architecture?

There are six main reasons for using a reference architecture.

Frame of reference

A reference architecture provides a frame of reference to get an overview of a system’s structure, organization, functionality, and behavior. Because the principles of the system’s architecture are defined by the reference architecture to which it complies, it becomes easier to understand how a system is put together and how to effectively maintain, update, and enhance the system.

Interoperability

Systems are almost never standalone islands of capability. They fit into both an engineering and an operational context. In the engineering context, we have common development tools, plug-and-play components, and other common elements to assist in design and development. In the operational context, we have concerns about the use, maintenance, and sustainment of the system in operation – including the *dev sec ops* culture, automation, and integration of the system into enterprise architectures.

Interoperability is also important when you have a portfolio of related systems. You will gain economy-of-scale advantages with common usable elements and designs across a portfolio. Reference architectures are a way of specifying that commonality.

Reuse of effective architectures

The vast majority of systems are rehashing or slight improvements on the existing design. We have decades of experience developing avionics and medical systems, for example. As we gain industrial experience with such systems, we learn which architectures are effective in practice (and not just in theory) and which are not. Complying with a reference architecture enables gaining these benefits with relatively little effort.

Benchmarking

Comparing metrics and outcomes of different systems is made easier when they are specializations compliant with a common reference architecture.

Regulatory compliance

In regulated industries, such as aerospace, automotive, and medical, gaining regulatory approval is crucial to a system's success. Regulators often show a preference for architectures that have previously demonstrated that they are safe, secure, and effective. Reference architectures can provide that assurance.

Owning the baseline

In the **US Department of Defense (DoD)**, a common concern is the ownership of the baseline technology and architecture. The DoD doesn't develop the vast majority of systems that it owns but has them developed under an acquisitions process. One of the historical problems with the acquisitions approach is that the **Original Equipment Manufacturers (OEMs)** generally retain ownership of the system design as their intellectual property. This has the downside of *vendor lock*. The DoD can call for the development of a system – such as the F-35 aircraft – and purchase hundreds of millions of dollars worth of systems, but they do not own the design. As such, when it becomes necessary to modernize or update the system, the DoD cannot simply go through an open acquisitions process but is limited to working with the vendor who owns the design. This has led to the **Modular Open Systems Approach (MOSA)**. See <https://ac.cto.mil/mosa/>

Using a reference architecture

A reference architecture is a collection of interlocking patterns for a kind of system or a set of systems in a kind of operational environment. Put another way, a reference architecture is a generalized set of abstractions that collectively organize a system at a strategic level. However, you don't manufacture a reference architecture; you manufacture a specific architecture. The use of a reference architecture involves the specification of a specific system that complies with the reference architecture and instantiates the patterns that constitute it.

Purpose

The purpose of this recipe is to create a specific system design that conforms to a reference architecture. This recipe is particularly relevant whether you want to develop a new system *type* or simply a system *instance*.

Inputs and preconditions

The system's primary capabilities have been identified and analyzed, along with a significant set of system requirements.

Outputs and postconditions

A reference architecture is selected and a specific system architecture is defined, especially with respect to conformance to the selected reference architecture.

How to do it

Figure 3.49 shows the workflow for this recipe:

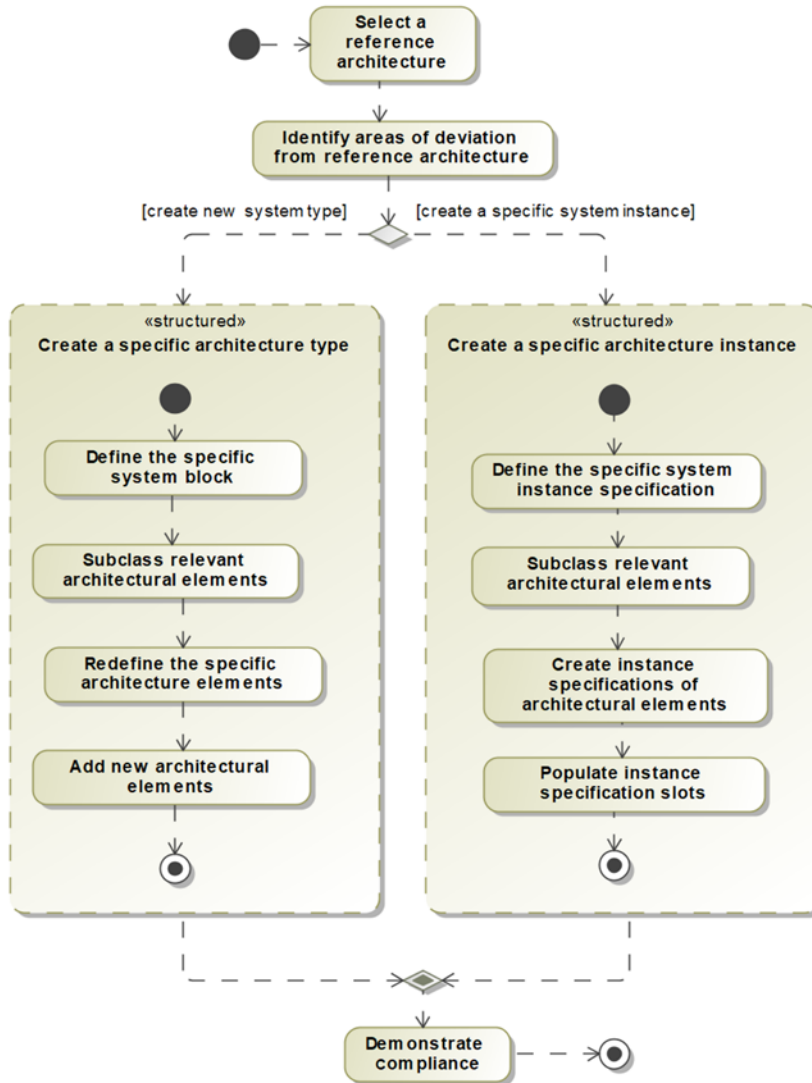


Figure 3.49: Specializing a reference architecture

Select a reference architecture

In your industry, it is likely that reference architectures exist and are competing with each other. It may also be that there is a single reference architecture provided to you. It may even be that one of your engineering tasks is to develop a reference architecture.

In any case, this first step is to select the reference architecture with which your system must conform. This architecture should have, at a minimum:

- Definition of the operational context in which conformant systems exist both as type definitions (BDD) and connected architectures (IBD)
- Standardized and possibly generic interfaces to actors in the operational context
- Standard capabilities of a system that is conformant, generally represented as system use cases
- Composition architecture of the system, including identification of the subsystems relevant to conformance, and the multiplicities of those relations
- Type architecture of the system, including the key properties of the subsystems, such as ports, relations, key system data, and key system functions allocated to the subsystems
- Connected architecture shows how the subsystems connect (IBD)
- Standardized and possibly generic interfaces between the subsystems

Identify areas of deviation from reference architecture

There are several different ways that a specific architecture differentiates itself from the reference architecture to which it conforms:

- Specific subtypes of identified subsystems or components
- Specific implementation technologies
- Specific multiplicities of contributing design elements
- Addition of new capabilities
- Addition of new subsystem and design elements to support new capabilities
- Addition of greater levels of detail over what is specified in the reference architecture

Create a specific architecture type

This structured action contains the following sequence of actions. This sub-workflow is appropriate when you want to create a new type of system or when you want to add a fundamentally new capability to the specific architecture.

Define the specific system block

In this recipe, one of the outcomes is a new system type that is still conformant to the reference architecture. The new system block is that type.

Subclass relevant architectural elements

When developing specific architectures, it is very common to subclass an architectural element to redefine how a behavior is performed (also known as **polymorphic behavior**) or to extend the functionality of the element. In SysML, this is done by subclassing the element.

Redefine the specific architecture elements

Once the subclasses are defined, they must be referenced in the specific architecture at exactly the same point in which their super-classes are used in the reference architecture. This is done with a redefinition of the element in the specialized system.

Add new architectural elements

Of course, a specialized system may do additional things beyond the capabilities of the reference architecture. An E3 Sentry (AWACS) is a specific kind of aircraft and may be compliant with an aircraft reference architecture, but it is also a **Command-and-Control Battle Management (C2BM)** platform as well. Additional architectural elements must be added to support additional capabilities of the specific system.

Create a specific architecture instance

This structured action contains the following nested actions. This sub-workflow is used when you 1) don't need to create a new type of reference architecture system and 2) you don't need to add fundamentally new elements or capabilities to the specific architecture. It is always possible to refine and extend existing parts and capabilities in an instance, but it is difficult to add entirely new ones.

Define the specific system instance specification

Another approach to creating a specialized architecture is to create a specific instance, represented with an instance specification. This is most useful when you needn't create a new type of system but merely wish to create a specific instance of the architecture. In this step, create an instance specification of the reference system.

Subclass relevant architectural elements

If the elements from the reference architecture are used as is, you can create instance specifications of those elements to plug them into the slots of the system instance specification. You may, however, want to put in specialized forms of those elements, so in this step, you may subclass some of the architectural elements.

Create instance specification of architectural elements

The system instance specification has slots for all its part properties. In this step, create an instance specification for every subsystem you want contained within the system instance.

Populate instance specification slots

In this step, you add the instance specifications of the architectural elements into the slots of the system instance specification.

Demonstrate compliance

The last step in this recipe is to demonstrate how the specific system complies with the reference architecture. This may be a simple matter if relations formally represent compliance when it is direct usage in the specific architecture or the subclassing of elements defined in the reference architecture. It is a bit more work when you don't have the model of the reference architecture or when the reference architecture is notional or not formally defined.

Example

Select a reference architecture

For this example, we'll start with a reference architecture for a generic **Exercise Bike**. The reference composition architecture for this system is shown in *Figure 3.50*:

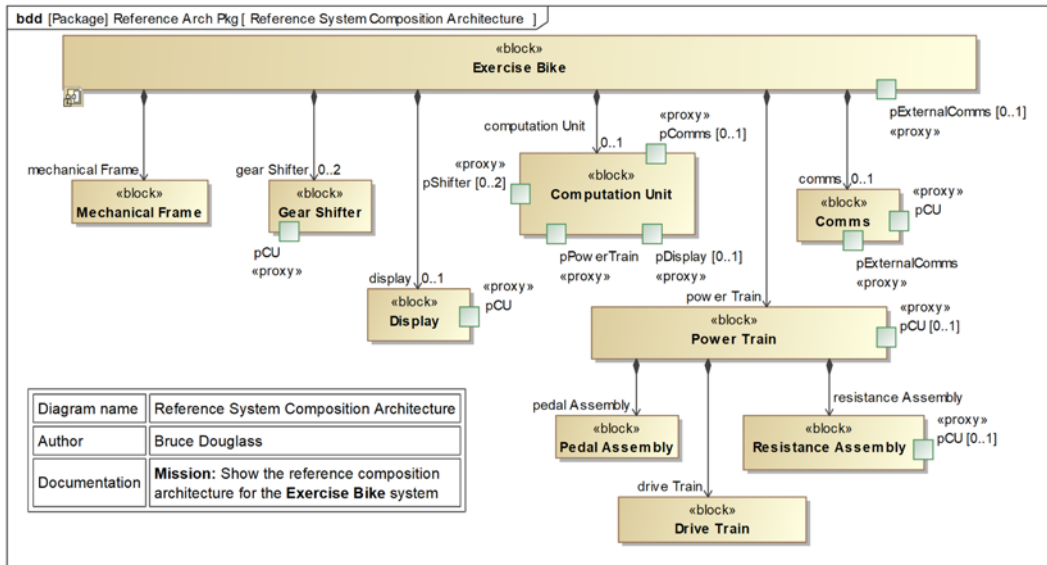


Figure 3.50: Exercise bike reference composition architecture

This system has a **Mechanical Frame**, zero to two **Gear Shifters**, and a **Power Train** subsystem that contains an internal **Pedal Assembly**, **Drive Train**, and **Resistance Assembly**. If the system has at least one **Gear Shifter**, then it will also have a **Computation Unit** to change the resistance offered at the pedal assembly. Optionally, if the system has at least one **Gear Shifter** and a **Computation Unit**, it may also optionally have a **Display** and may also provide information via a **Comms** unit.

Figure 3.51 shows how these parts connect in the reference architecture:

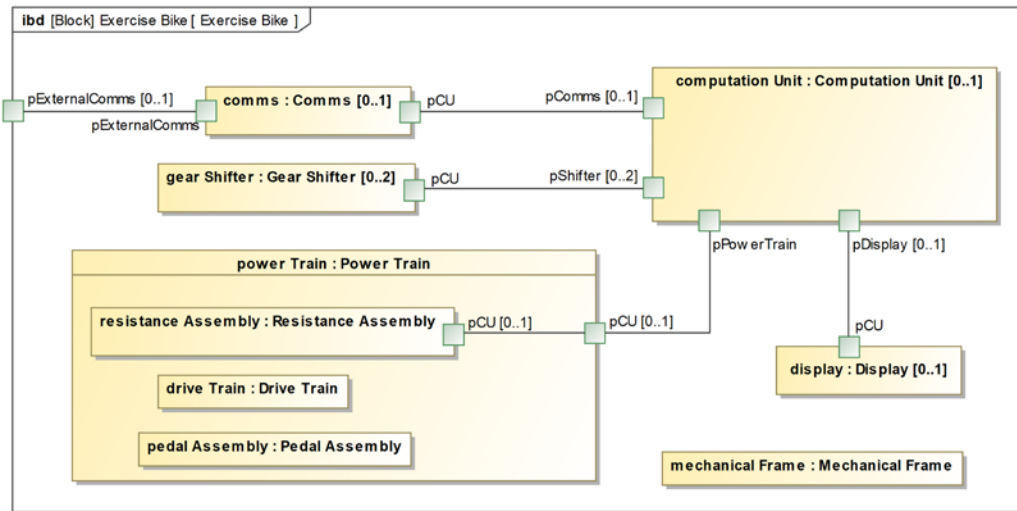


Figure 3.51: Exercise bike reference connected architecture

One can imagine different specific architectures that could conform to this reference architecture. The simplest such system would have only the required subsystems of a mechanical frame and a power train.

Indeed, I trained on wind turbine trainers in the 1980s that worked just like that. More elaborate specific systems might add a dual shifter (simulating front chain rings and a rear cassette) and a computation unit that modifies the resistance, which might be provided hydrostatically by pushing fluid through variable valves or by varying resistance electronically with a motor. An even more elaborate system that still conforms to this reference architecture might add both a display of information about resistance and gearing, and even send it out through the (optional) communication system. Each of these variants can vary not only the multiplicity of elements (say, from 0 to 1) but also their type and implementation. A display could be done with simple LEDs, a low-resolution black and white screen, a high-resolution color display, or even a virtual reality headset. Communications can be similarly varied to support different kinds of networks and communication standards. Even with such a simple reference architecture, one can easily envision many different specific architectures derived from it.

Identify areas of deviation from the reference architecture

Let's envision two systems, one for each of the different paths through the workflow. For the left workflow in *Figure 3.49* (**Create a specific architecture type**), let's develop an exercise bike that has a single shifter, a computational unit, and a nice large color display.

Redefine the specific architectural elements

This is the secret sauce of this recipe. You must redefine the architecture with the subclassed elements and with the specific multiplicity in the **LORE** system.

To do this in Cameo, first, expose the inherited parts for the **LORE** symbol on the diagram with the **Symbol Properties** dialog. Then, right-click the part property in the block symbol you want to redefine and select **Refactor > Redefine To**. This pops up a dialog from which you can select the subclass you want. If you only want to change the multiplicity, use this same technique but select the original type, then open the specification dialog for the part and change the multiplicity. *Figure 3.53* shows what that looks like for redefining the **display** part:

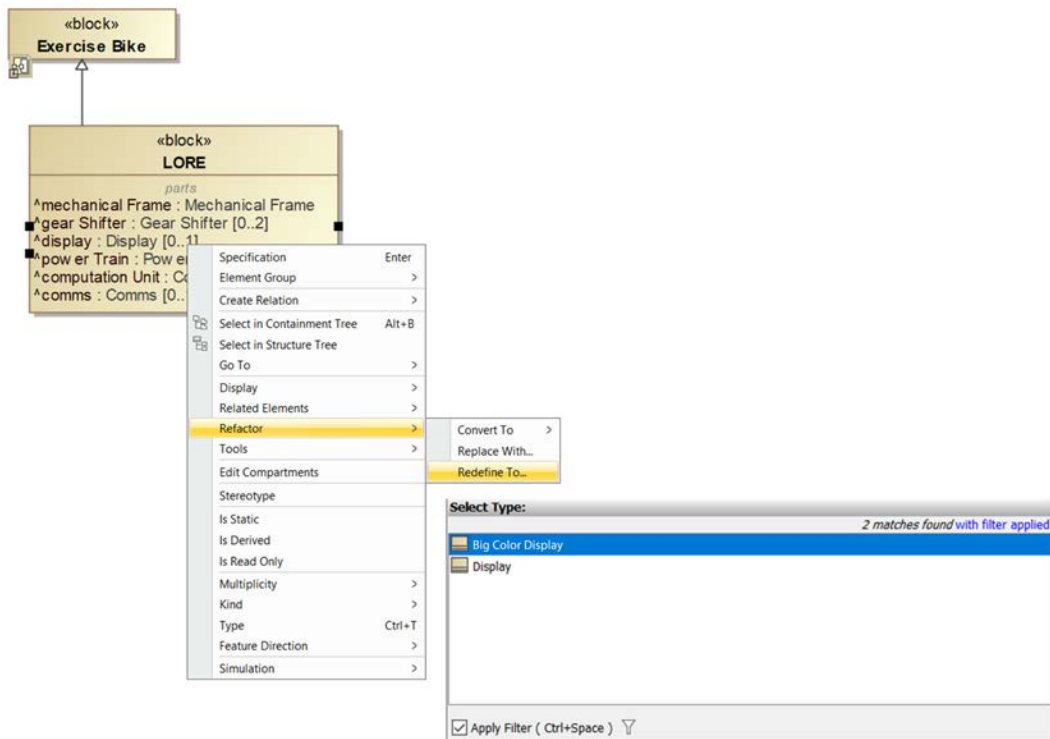


Figure 3.53: Redefining part properties in Cameo

The result of this redefines the parts inherited in the **LORE** block to the new subclasses and then part property multiplicity can be changed in their specifications dialog as needed. The redefinition of the port multiplicity, such as **LORE::pExternalComms**, must be done in the specification dialog for the owning block.

Once this is done, you have the parts redefined in the **LORE** system as shown in *Figure 3.54*:

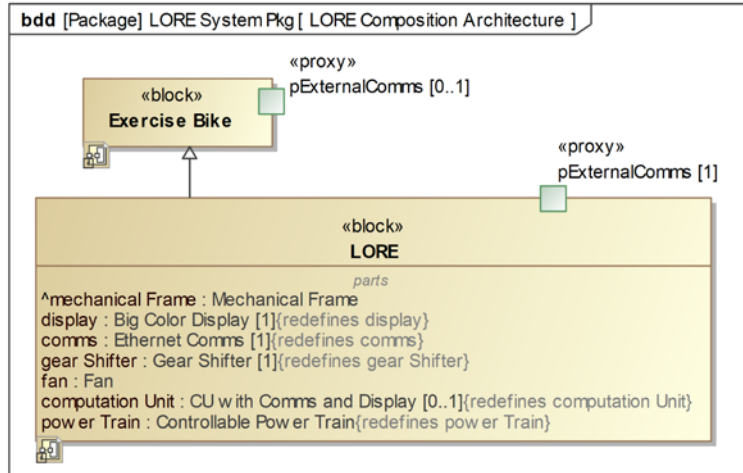


Figure 3.54: LORE redefined

Add new architectural elements

Now we can add the **Fan** to the **LORE** block with a composition relation. We can see in *Figure 3.55* the connected architecture of the **LORE** system with all its specialized parts and specific multiplicities:

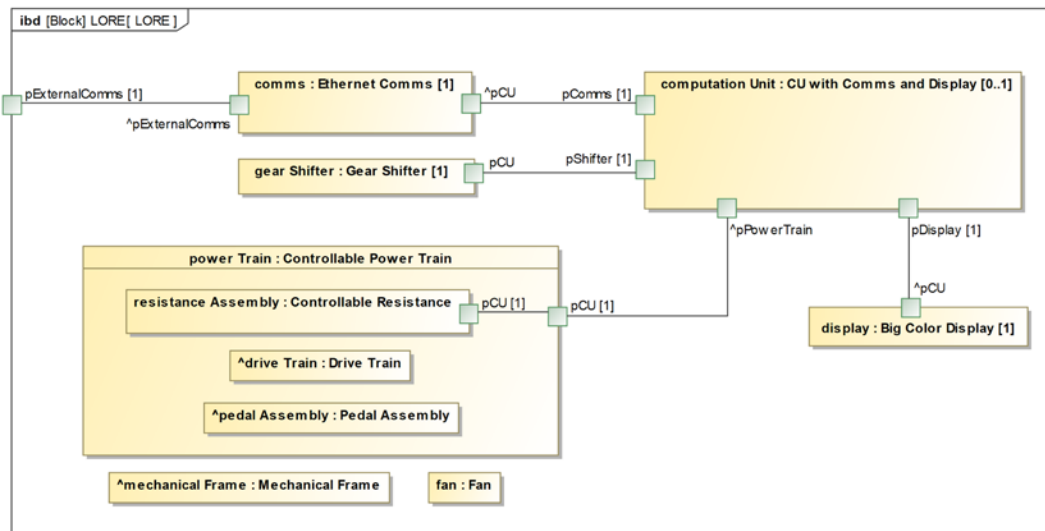


Figure 3.55: LORE redefined connected architecture

Let's now create the right workflow and create our specific instance of the **Cycling God** system. Remember that this system has two shifters and supports the BLE communications protocol.

Define the specific system instance specification

In this step, we add an instance specification named **Cycling God** of the **Exercise bike** block.

Subclass relevant architectural elements

We only really need to subclass **Comms** to **BLE Comms** because we want different behavior in the included element, that is, support for the BLE protocol.

Create instance specifications of architectural elements

The instance specifications for the architectural element of the **Cycling God** instance are shown in *Figure 3.56*:

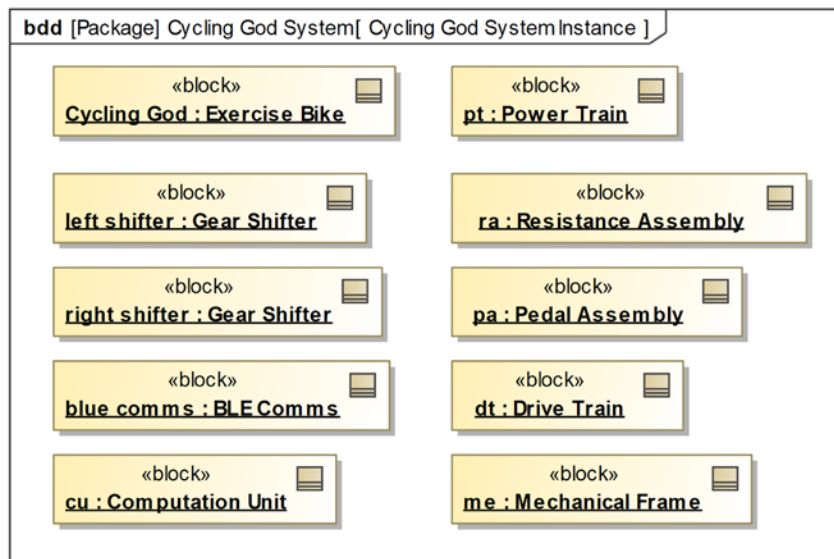


Figure 3.56: Set of instance specifications

Populate instance specification slots

The next step is to populate the slots in the instance specifications. The **Power Train** has three instance slots, so they must be populated with the **ra**, **pa**, and **dt** instances. The **Exercise Bike::Gear Shifter** slot must hold both the left and right shifter instances.

The resulting instance specifications with populated slots are shown in *Figure 3.57*. Note that the names of the instances are given in the slots:

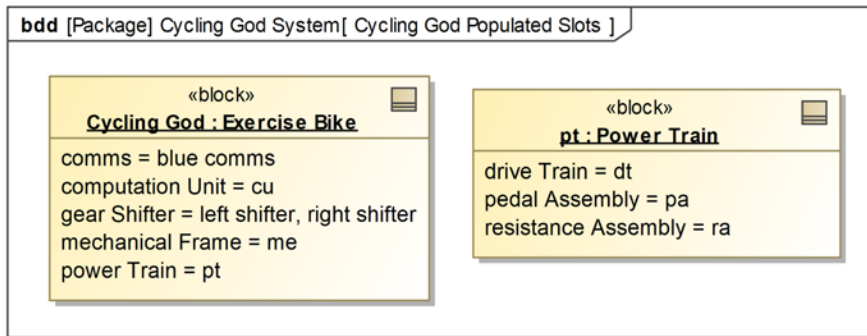


Figure 3.57: Populated instance specifications

In this latter example, we minimized the use of subclasses. Had we wanted to, we could have specialized the multiplicities of the ports and parts with subclassing and redefinition.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>



4

Handoff to Downstream Engineering

Recipes in this chapter

- Preparation for Handoff
- Federating Models for Handoff
- Logical to Physical Interfaces
- Deployment Architecture I: Allocation to Engineering Facets
- Deployment Architecture II: Interdisciplinary Interfaces

The purpose of the Handoff to Downstream Engineering recipes is to:

- Refine the system engineering data to a form usable by downstream engineers
- Create separate models to hold the prepared engineering data in a convenient organizational format (known as *model federation*)
- For each subsystem, work with downstream engineering teams to create a deployment architecture and allocate system engineering data into that architecture

It is crucial to understand that the handoff is a *process* and not an *event*. There is a non-trivial amount of work to do to achieve the above objectives. As with other activities in the Harmony aMBSE process, this can be done once but is recommended to take place many times in an iterative, incremental fashion. It isn't necessarily difficult work, but it is crucial work for project success.

The refinement of the system's engineering data is necessary because, up to this point, the focus primarily has been on its conceptual nature and logical properties. What is needed by the downstream teams are the physical properties of the system – along with the allocated requirements – so that they may design and construct the physical subsystems.

Activities for the handoff to downstream engineering

At a high level, creating the logical system architecture includes the identification of subsystems as types (blocks), connecting them up (the connected architecture), allocating requirements and system features to the subsystems, and specifying the logical interfaces between the architectural elements. Although the subsystems are “physical,” the services and flows defined in the interfaces are almost entirely logical and do not have the physical realization detail necessary for the subsystem teams. One of the key activities in the handoff workflows will be to add this level of detail so that the resulting subsystem implementations created by different subsystem teams can physically connect to one another.

For this reason, the architectural specifications must now be elaborated to include physical realization detail. For example, a logical interface service between a RADAR and a targeting system might be modeled as an event `evGetRadarTrack(rt: RadarTrack)`, in which the message is modeled as an asynchronous event and the data is modeled as the logical information required regarding the radar track. This allows us to construct an executing, computable model of the logical properties of the interaction between the **targeting** and **RADAR** subsystems. However, this logical service might actually be implemented as a **1553** bus message with a specific bit format, and it is crucial that both subsystems agree on its structure and physical properties for the correct implementation of both subsystems. The actual format of the message, including the format of the data held within the message, must be specified to enable these two different development teams to work together. That is the primary task of the process step *Handoff to Downstream Engineering*.

Other relevant tasks include establishing a single source of truth for the physical interface and shared physical data schema in terms of a referenced specification model, the large-scale decomposition of the subsystems into engineering facets, the allocation of requirements to those facets, and the specification of interdisciplinary interfaces. A *facet*, you may recall, is the term used for the design contribution from a single engineering discipline, such as software, electronics, mechanics, and hydraulics.

The recipes in this chapter are devoted to those activities.

Once fully defined, the handoff data can be used in a couple of different ways. For many organizations, it is used as the starting point for the downstream implementors/designers. In other cases, it can be incorporated into **Request for Proposals (RFPs)** and bids in an acquisition process or for source selection if **Commercial Off-the-Shelf (COTS)** is to be used.

Starting point for the examples

In this chapter, we will focus on the requirements and allocations from a number of use case analyses and architectural work done earlier in the book. Specifically, we'll focus on requirements from the use cases **Compute Resistance, Emulate Front and Rear Gearing, and Emulate DI Shifting**. Because not all subsystems are touched by these requirements, we will focus this incremental handoff on the following subsystems: **Comms, Rider Interaction, Mechanical Frame, Main Computing Platform, and Power Train**. The other subsystems will ultimately be elaborated and added to the system in future iterations. For brevity, we will not consider the subsystems nested with those subsystems but focus solely on the high-level subsystems.

The relevant system-connected architecture is shown in the internal block diagram of *Figure 4.1*:

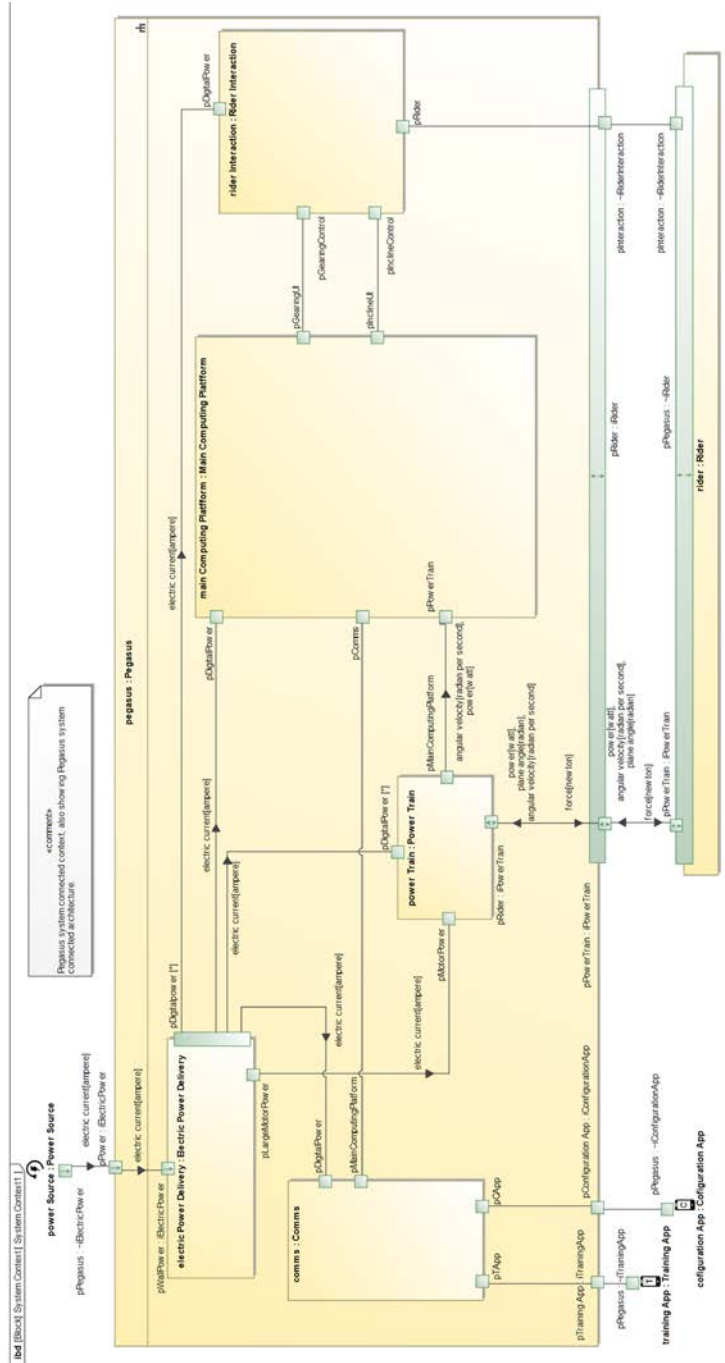


Figure 4.1: Connected architecture for incremental handoff

System requirements have either been allocated or derived and allocated. *Figure 4.2* shows some of the requirements with their specification text:

#	△ Name	Text
1	R 125 DIReg01	In DI Shifting mode, if the UP button is pressed and the system is already in the highest possible gear, then the system shall audibly beep and keep the current gearing.
2	R 126 DIReg02	The system shall provide shifting with a DI Shifting mode that enables the DI shifting buttons and disables the shifting levers.
3	R 127 DIReg03	In DI Shifting mode, the UP button shall shift into the next highest possible gearing from the selected gear set, as measured in gear inches.
4	R 128 DIReg04	In DI Shifting mode, the UP button shall shift into the next highest possible gearing from the selected gear set, as measured in gear inches.
5	R 129 DIReg05	In DI Shifting mode, if the DOWN button is pressed and the system is already in the lowest possible gear, then the system shall audibly beep and keep the current gearing.
6	R 130 DIReg06	In DI Shifting mode, when an upshift requires changing the chain ring, the system shall progress to the next largest gearing, as measured by gear inches.
7	R 131 DIReg07	In DI Shifting mode, when a downshift requires changing the chain ring, the system shall progress to the next smallest gearing, as measured by gear inches.
8	R 132 DIReq10	The system shall enter DI Shifting Mode by selecting that option in the Configuration App.
9	R 133 DIReq11	Once DI Shifting mode is selected, this selection shall persist across resets, power resets, and software updates.
10	R 134 DIReq12	Mechanical shifting shall be the default on initial start up or after a factory-settings reset.
11	R 135 DIReq13	The system shall leave DI Shifting mode when the user selects the Mechanical Shifting option in the Configuration App.
12	R 136 efarg01	The system shall notify the rider of the current number of chain rings and cassette rings on start up.
13	R 137 efarg02	The system shall accept a rider command to enter a mode to configure the gearing.
14	R 138 efarg03	The system shall accept a rider command to set up from 1 to 3 front chain rings, inclusive.
15	R 139 efarg04	The default number of chain rings shall be 2.
16	R 140 efarg05	The rider shall be able to decrement the cassette ring from a higher (smaller number of teeth) to the next lower (larger number of teeth) gear until the largest cassette ring is reached.
17	R 141 efarg06	The system shall accept a rider command to set up from 10-12 cassette rings, inclusive.
18	R 142 efarg07	The default number of cassette rings shall be 12.
19	R 143 efarg08	The system shall accept a rider command to set any chain ring to have from 20 to 70 teeth.

Figure 4.2: Pegasus system and subsystem requirements

Figure 4.3 shows how they are satisfied by subsystems. A nice feature of Cameo tables is that it identifies not only direct satisfies relations between subsystems and requirements, but also relations implied by the composition architecture (i.e. a part having a satisfy relation implies a satisfy relation to the element owning that part):

Criteria

Row Element Type: Requirement Column Element Type: Block

Row Scope: ReqsPkg Column Scope: Architecture Design Pkg

Dependency Criteria: Satisfy,Satisfy (Implied) Direction: Both Show Elements: All

Legend

→ Satisfy

→ Satisfy (Implied)

Architecture Design Pkg

		Configuration App	Comms	Data Management	Drive Train	Electric Power Del	Gear Control	Incline Control	Main Computing P	Motor Assembly	Pedal Assembly	Pegasus	Power Source	Power Train	Rider	Rider Application	Rider Interaction	System Configur	System Context	Training App
R 176 CR_requirement_10	3								→			→								→
R 177 CR_requirement_11	4		→						→			→								→
R 178 CR_requirement_12	3								→			→								→
R 179 CR_requirement_13	3								→			→								→
R 180 CR_requirement_14	3								→			→								→
R 181 CR_requirement_15	4		→						→			→								→
R 182 CR_requirement_21	4		→						→			→								→
R 183 CR_requirement_22	4		→						→			→								→
R 184 CR_requirement_23	4		→						→			→								→
R 185 CR_requirement_24	4								→			→		→						→
R 186 CR_requirement_25	4		→						→			→								→
E 187 CRD_01	3								→			→								→
E 188 CRD_02	3								→			→								→
E 189 CRD_03	3								→			→								→
E 190 CRD_04	3								→			→								→
E 191 CRD_05	3								→			→								→
E 192 CRD_07	3								→			→		→						→
E 193 CRD_06	3								→			→								→
E 194 CRD_09	3		→						→			→								→
E 195 CRD_22																				
E 196 CRD_21																				
E 197 CRD_23																				
E 198 CRD_08	3		→						→			→								→
E 199 CRD_11	4		→						→			→								→

Figure 4.3: Requirements - subsystem satisfy matrix

Based on the white box scenarios, the interface blocks for the ports have been elaborated, as shown in the block definition diagram in *Figure 4.4*. Note that the interfaces include signals sent from one element to another but also flow properties, indicating flows between elements not invoking behaviors directly:

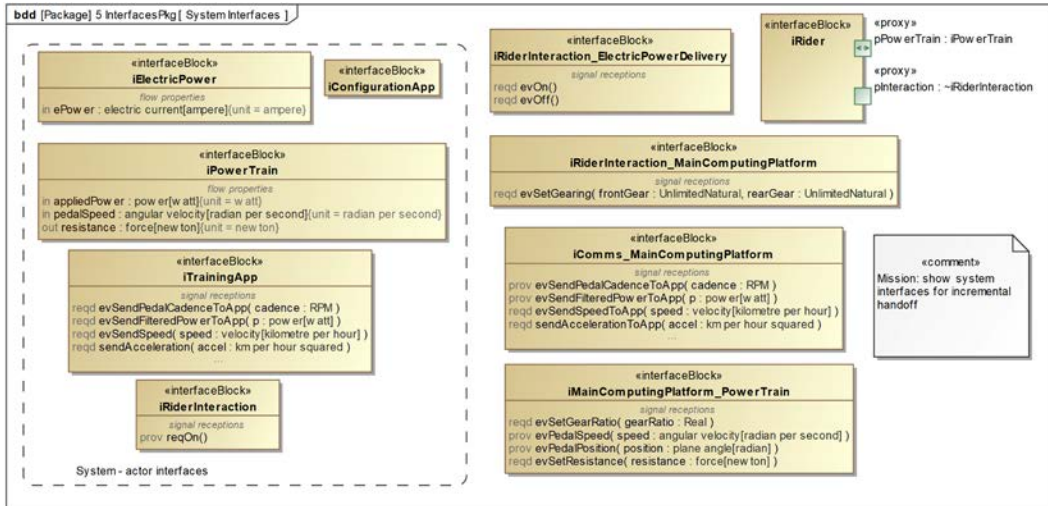


Figure 4.4: Subsystem interface blocks for incremental handoff

The logical data schema for data passed in the services has been partially elaborated. The information is collected into the subpackage **InterfacesPkg::LogicalDataSchemaPkg**. Figure 4.5 shows the scalar value types, units, and quantity kinds. Named constants are shown as read-only value properties within a block.

Figure 4.6 shows the logical data schema that uses those types to represent the information within the architecture. The figure also shows the «tempered» stereotype, which is used to specify important value type metadata such as extent (the allowable range of values), lowest and highest values, latency, explicitly prohibited values, and precision, accuracy, and fidelity.

As a side note, some of the metadata can be specified in constraints, such as a constraint on a **power** value property using OCL: $\{(self \geq 0) \text{ and } (self \leq 2000)\}$. However, Cameo doesn't seem to evaluate constraints owned by value properties for a violation during simulation (one of the key benefits of modeling subranges that way), and can only evaluate them when assigned to blocks. In such a case, a block **Measured Performance Data** could constrain the value property **power** with $\{(power \geq 0) \text{ and } (power \leq 2000)\}$. For this reason, we decided that using the stereotype tags to represent the subrange metadata was a simpler solution.

I want to emphasize again that this is not the entire, final architecture; rather, it is the architecture for a particular iteration. The final, complete architecture will contain many more elements than this:

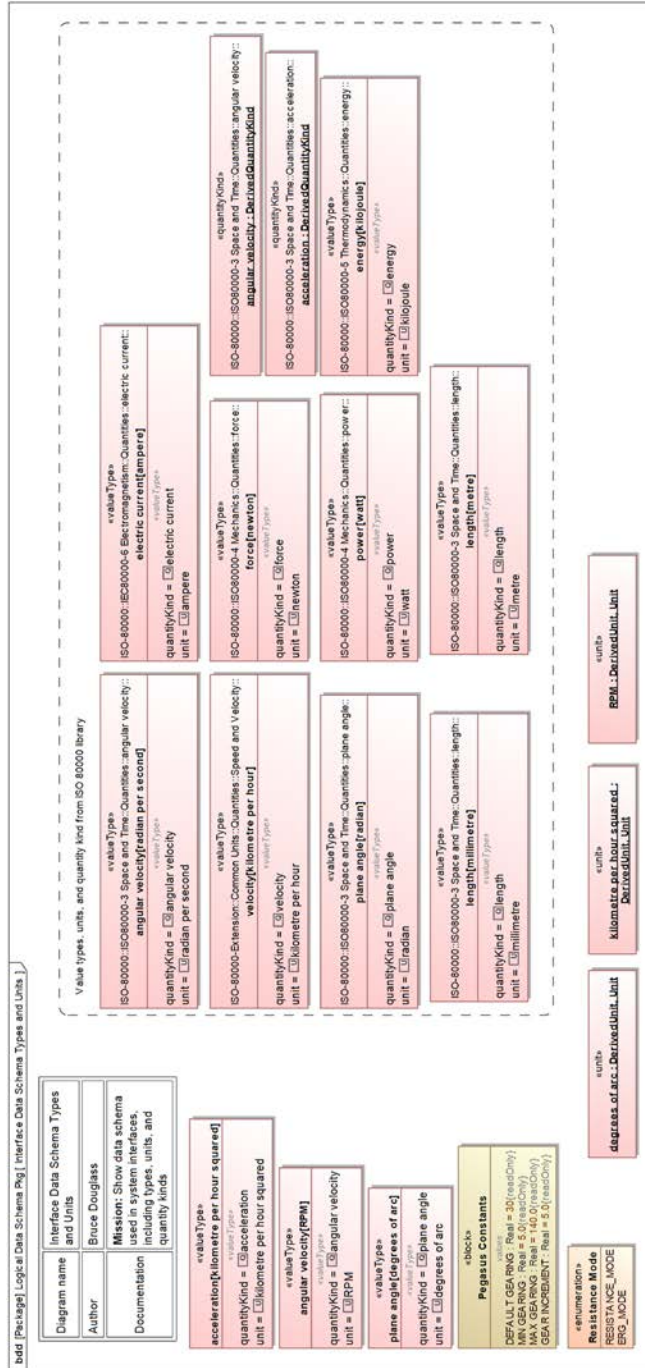


Figure 4.5: Scalar value types, units, and quantity kinds

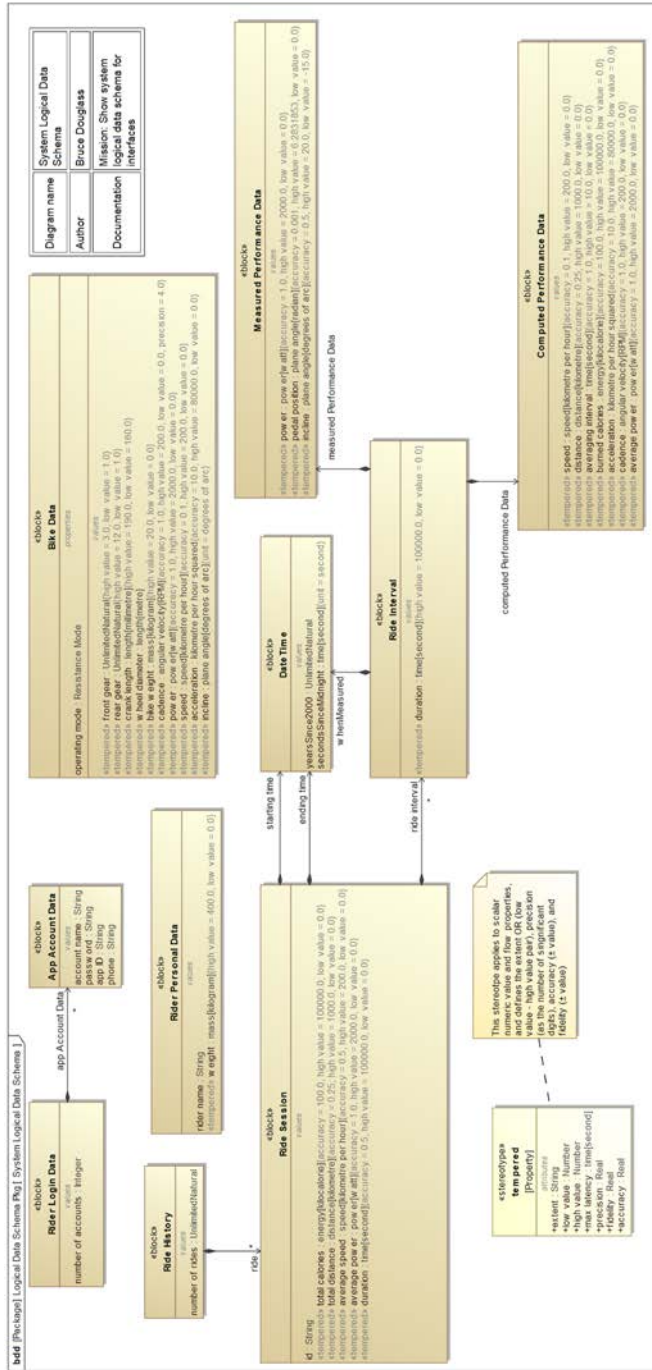


Figure 4.6: Blocks and logical data schema

Preparation for Handoff

Purpose

The purpose of this recipe is to facilitate the handoff from systems engineering to downstream engineering. This will consist of a review of the handoff specifications for adequacy and organizing the information to ease the handoff activities. Ideally, each subsystem team needs access to system model information in specific and well-defined locations, and they only see information necessary for them to perform their work.

Inputs and preconditions

The precondition for this recipe is that the architecture is defined well enough to be handed off to subsystem teams for design and development. This means that:

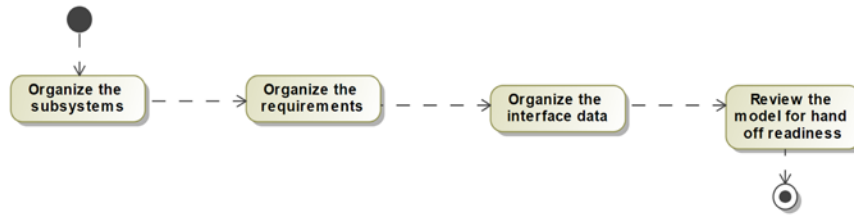
- The system requirements for the handoff are stable and fit for purpose. This doesn't necessarily mean that the requirements are complete, however. If an iterative development process is followed, the requirements need only be complete *for the purpose of this increment*.
- The subsystem architecture is defined including how it connects to other subsystems and the system actors. As in the previous point, not all subsystems need to be specified, nor must the included subsystems be fully specified in a given iteration for an incremental engineering process. It is enough that the subsystem architecture is specified well enough to meet the development needs of the current iteration.
- The subsystem requirements have been derived from the system requirements and allocated to the subsystems.
- The logical subsystem interfaces are defined so that the subsystems can collectively meet the system needs. This includes not only the logical specification of the services but also the logical data schema for the information those services carry. In addition, flows not directly related to services must also be logically defined as a part of the subsystem interfaces.

Outputs and postconditions

The resulting condition of the model is that it is organized well to facilitate the handoff workflows. Subsystem requirements and structures are accessible from external models with minimal interaction between the system and subsystem models.

How to do it

Figure 4.7 shows the workflow to prepare the handoff. While straightforward, doing a good job at this task means that the subsequent work for the handoff will be much easier:



Organize the subsystems

The next recipe, *Federating Models for Handoff*, will create a model for each subsystem. Each subsystem should reach into the system model as efficiently as possible to retrieve its – and only its – specification details. This is not to hide information about other subsystems but is instead meant to simplify the process of retrieving specification information that it needs. We must organize each subsystem into its own package so that the subsystem team can add this package to their model to serve as their starting architectural specification.

Organize the requirements

At this point, all the requirements relevant to the current handoff should be allocated to the subsystems. This means that the subsystems have relations identifying the requirements they must meet. These may be either «allocate» or «satisfy» relations, owned by the subsystem blocks and targeting the appropriate subsystems. This step involves locating the requirements so that they can be useably accessed by the downstream engineers and provided in summary views (tables and matrices). The package holding the requirements, which might be organized into subpackages, will be referenced by the individual subsystem models during and after the handoff.

Organize the interface data

One of the things put into the **Shared Model** (in recipes later in this chapter) is the physical interface specifications and the physical data schema for the data used within those interfaces. The logical interfaces and related schema must be organized within the systems engineering model so that it can easily be referenced by the **Shared Model**. This means putting the interfaces, interface blocks, and data definition elements – such as blocks, value types, quantity kinds, and units – in a package so that the **Shared Model** can easily reference it.

Review model for handoff readiness

After all the information is readied, it should be reviewed for completeness, correctness, and appropriateness for the downstream handoff workflow to occur. Participants in the review should be not only the systems engineers creating the information but also the subsystem engineers responsible for accepting it.

To review the readiness of the systems model for handoff, we look at the parts of the model that participate in the process:

- Subsystems:
 - Are the subsystems each nestled within their own packages?
 - Does each subsystem have ports defined with correct interfaces or interface blocks and with the proper conjugation?
 - Is the set of all subsystems complete with respect to the purpose of the handoff (e.g. does it support the functionality required of this specific iteration)?
 - Does each subsystem package have a reference to the requirements allocated to it?
- Requirements:
 - Are the requirements all located within a single package (with possible subpackages) for reference?
 - Are all requirements relevant to this handoff either directly allocated to subsystems or decomposed and their derived requirements allocated?
- Interfaces:
 - Is all the interface data located within a single package (which may have subpackages)?
 - Are all the interfaces or interface blocks detailed with the flow properties, operations, and event receptions?
 - Is the directionality of each interface feature set properly?
 - Are the properties of the services (operations and event receptions) complete, including data passed via those services?
 - Is the logical interface data schema fully, properly, and consistently defined for blocks, value types, quantity kinds, and units?

Example

The starting point for the example of this recipe is the detail shown at the front of this chapter. In this recipe, we will organize and refactor this information to facilitate the other handoff activities.

Organize the subsystems

Here, we will create a package within the **Architecture Pkg::Architectural Design Pkg** for each subsystem and relocate the subsystem block into that package. This will support later import into the subsystem model. These packages will be nested within the **Architecture Pkg::Architectural Design Pkg**. See *Figure 4.8*:

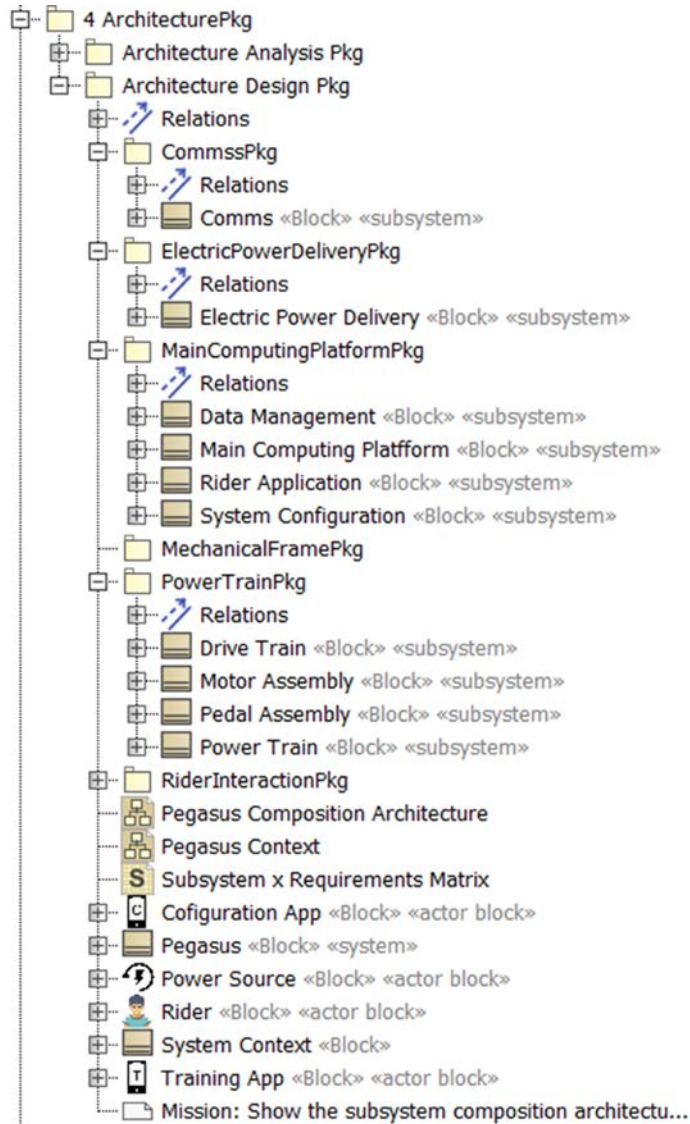


Figure 4.8: Subsystem organization in the system model

Organize the requirements

The requirements are held within the **Requirements Analysis Pkg :: Requirements Pkg**. While it is possible to redistribute the requirements into a package per subsystem based on those relations, it isn't always practical to do so. Minimally, two summary views are needed. First, a table of the requirements showing the name and specification text of all requirements must be provided. This should be placed in the **Requirements Analysis Pkg::Requirements Pkg** so the subsystem teams can access it easily (as shown in *Figure 4.2* and *Figure 4.3*) and use it to determine their requirements.

Secondly, matrix views showing the allocation relations between the subsystems and the allocated requirements are needed (as shown in *Figure 4.3*). Since the subsystems are the owners of the «allocate» and «satisfy» relations, it makes sense for these matrices to be in the **Architecture Pkg::Architectural Design Pkg**. The overall matrix can be put in that package directly. *Figure 4.9* shows the package organization of the requirements in the model:

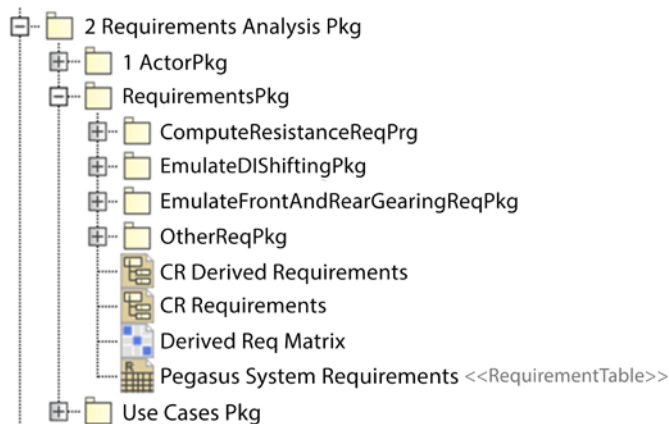


Figure 4.9: Requirements organization

In addition, each subsystem package contains an allocation matrix just showing the requirements allocated to that specific subsystem. *Figure 4.10* shows the example for the **Comms** subsystem. The Cameo feature **Show Elements > With relations** is used to just show the requirements allocated to the **Comms** subsystem:

Criteria			
Row Element Type:	Requirement	Column Element Type:	Block
Row Scope:	RequirementsPkg	Column Scope:	CommsPkg
Dependency Criteria:	Satisfy,Satisfy (Implied)	Direction:	Both
		Show Elements:	With relations
Legend		CommsPkg	
↗ Satisfy		Comms	
RequirementsPkg		15	
ComputeResistanceReqPrg		11	
R 174 CR_requirement_8		1	↗
R 175 CR_requirement_9		1	↗
R 177 CR_requirement_11		1	↗
R 181 CR_requirement_15		1	↗
R 182 CR_requirement_21		1	↗
R 183 CR_requirement_22		1	↗
R 184 CR_requirement_23		1	↗
R 186 CR_requirement_25		1	↗
E 194 CRD_09		1	↗
E 198 CRD_08		1	↗
E 199 CRD_11		1	↗
EmulateDIShiftingPkg		2	
R 132 DIRReq10		1	↗
R 135 DIRReq13		1	↗
EmulateFrontAndRearGearingReqPkg		2	
R 137 efarg02		1	↗
R 153 efarg18		1	↗

Figure 4.10: Comms subsystem allocated requirements

Organize the interface data

In the canonical model organization used in the example, the logical interfaces are located in the systems engineering model **InterfacesPkg**. These may be either *Interfaces* or *Interface Blocks* depending on whether standard or proxy ports are used. In addition, the package contains the specifications of the logical data passed via those interfaces, whether that data is expressed as service-independent flow properties or arguments for services. All this information should be organized for reference by the **Shared Model**, defined in the next recipe. The **Shared Model** will define physical interfaces and an associated data schema that represents the logical interfaces and data schema in the systems model. Those physical interfaces will then be made available to the subsystem teams via the model federation defined in the next recipe.

The information for this example is already shown in *Figure 4.4*, *Figure 4.5* and *Figure 4.6*. The organization of the interface data is shown in *Figure 4.11*. Although the figure only shows the services for a single interface block, note that we have followed the convention of using event receptions for all logical service specifications in the systems engineering model. These will be changed into a physical schema in later recipes in this chapter:

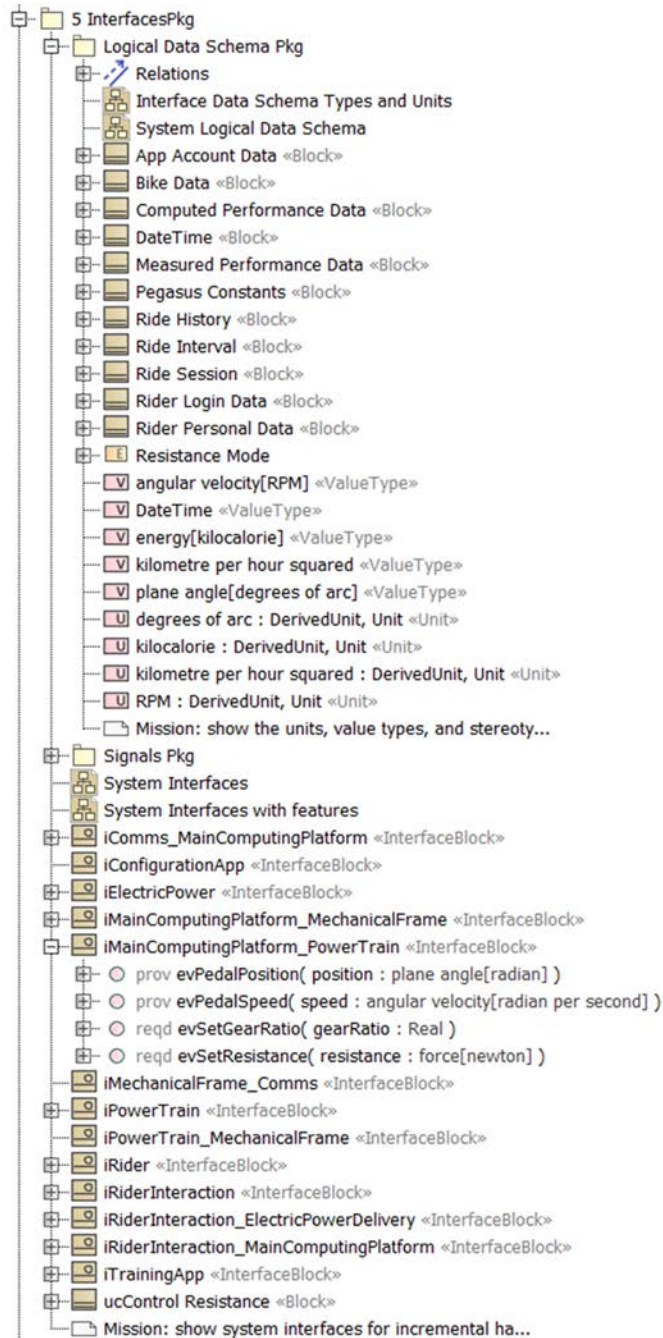


Figure 4.11: Organization of interface data

Review model for handoff readiness

In this example, the model is set up and ready for handoff. See the *Effective Reviews and Walkthroughs* recipe in *Chapter 5, Demonstration of Meeting Needs*, for the workflow for performing the review.

Federating Models for Handoff

In SysML, a *model* is not the same as a *project*. A model is a special kind of package used to organize information for a particular modeling purpose. It may contain many nested packages and even other nested models. What we normally call the “systems engineering model” normally contains at least two models: a specification model (containing requirements, use cases, and functional analyses) and an architecture model (identifying subsystems and their properties, relations, and interfaces). There may be additional models for various analyses, such as trade studies or safety analyses.

In many cases, all these models are contained within a single SysML *project*. A project can contain one or more models, but a model can also be split across multiple projects. This is often done when the model is being worked on by different, geographically distributed teams. Projects and models are related but are distinct concepts. Many people just squint and say *model = project* and while that isn’t wrong, neither is it the entire story.

Downstream engineering is different in that many separate, independent models will be created. These models will interact with each other in specific and well-defined ways. A set of such independent, yet connected models is called a **federation**, and the process of creating the models and their linkages is called a **model federation**.

One of the key ideas in a model federation is the notion of a *single source of truth*. This concept means that while there may be multiple sources for engineering data, each specific datum is situated in a single, well-defined location known as the datum’s **authoritative source**. When a value is needed, the authoritative source for that data is referenced.

This means that, for the most part, data is not *copied* from model to model, which can lead to questions such as “The value reported for this datum is different if I look at different sources – so which is correct?” Instead, data is *referenced*.

To share packages from a project in Cameo, the source project must identify the packages allowed to be shared. This is done by right-clicking a package in the Cameo containment tree and selecting **Project Usages > Share Packages**. This pops up the **Share Packages** dialog, which allows the user to select which packages they will allow to be shared with other projects.

Sharing packages is how models are federated.

In the Cameo tool, model federations are constructed with the **File > Use Project > Use Local Project** feature (or **File > Use Project > Use Server Project** for projects stored on Teamwork Cloud). This feature can add data – usually packages with their contained elements – to other models. This feature loads in packages and model elements from other models or projects, provided they have been designated by that project owner as sharable. Once the **Use Project** dialog pops up, you can select the desired reference project and see the sharable packages. You can access them as read-only or read-write. In the former case, you can see the used model elements and create references to them. In the latter case, you can also create bidirectional relations with those elements and modify them – this can lead to synchronization issues later, so we recommend:

Always make packages in a federation read-only.

If the element in the source project must be changed, simply open the source project and make the edits.

While many models might be federated, including CAD models, PID control models, environment simulation models, and implementation models, we will focus on a core set of models in the recipe. The models in our federation will be:

- Systems engineering model:
 - This is the model we've been working with so far
- Shared Model:
 - This model contains elements used by more than one subsystem model
 - Our focus here will be the system and subsystem physical interfaces, including the physical data schema for information passed in those interfaces
- Subsystem model [*]:
 - A separate model per subsystem is created to hold its detailed design and implementation

Purpose

The purpose of this recipe is to provide workspaces to support both the handoff process itself and the downstream engineering work to follow.

Inputs and preconditions

The starting point for this recipe is the system engineering model with engineering data necessary to perform the handoff to downstream engineering, including the set of identified subsystems. This information is expected to be organized to facilitate the referencing of requirements, architectural elements, and interface data for the linking together of the models in the federation.

Outputs and postconditions

At the end of the recipe, a set of models are constructed with limited and well-defined points of interaction. Each model is created with a standard canonical model organization supportive of its purpose.

How to do it

The steps involved in the recipe are shown in *Figure 4.12*:

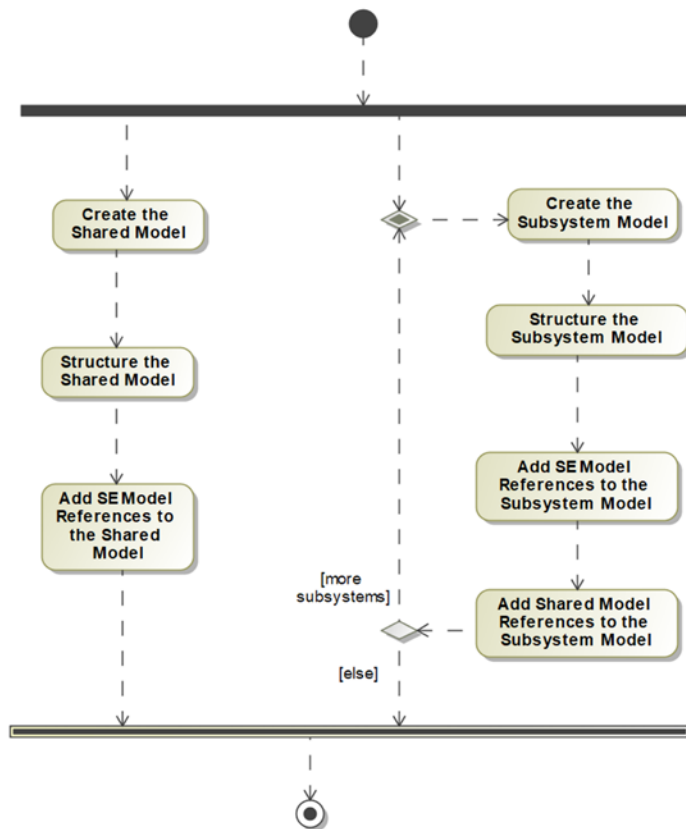


Figure 4.12: Create model federation

Create the Shared Model

Create a new model, named the **Shared Model**. This is normally located in a folder tree underneath the folder containing the system engineering model, but it needn't necessarily be so.

Structure the Shared Model

This model has a recommended initial structure containing the package **Physical Interfaces Pkg**, which will be referenced by the **Subsystem Models**. This package will hold the specification of the interfaces and has two nested packages, one for the physical data schema and the other for related stereotypes. The stereotypes are commonly used to specify metadata to indicate sub-ranges, extent, and precision, and this information is important for the subsystem teams to use those data successfully. The subsystems need a common definition for interfaces and they get it by referencing this package.

It should be noted that it is common to add an additional **DomainsPkg** later in the downstream engineering process to hold design elements common to multiple subsystems, but the identification of such elements is beyond the scope of the handoff activity.

Add SE Model references to the Shared Model

There are a small number of **SE Model** packages that must be referenced in the Shared Model. Notably, this includes:

- A **Requirements Pkg** package
- An **Interfaces Pkg** package
- A **Shared Common Pkg** package, which may contain a **Common Stereotypes** profile

The first two are discussed in some detail in the previous recipe. The **Shared Common** package includes stereotypes and other reusable elements that may be necessary to properly interpret the elements in the referenced packages.

In Cameo, adding these references is done with the **File > Use Project** feature. If you select the menu item, the **Use Project** dialog will open and you can navigate to the systems engineering model you want to reference.

Once there, you can select the packages it has shared (Figure 4.13):

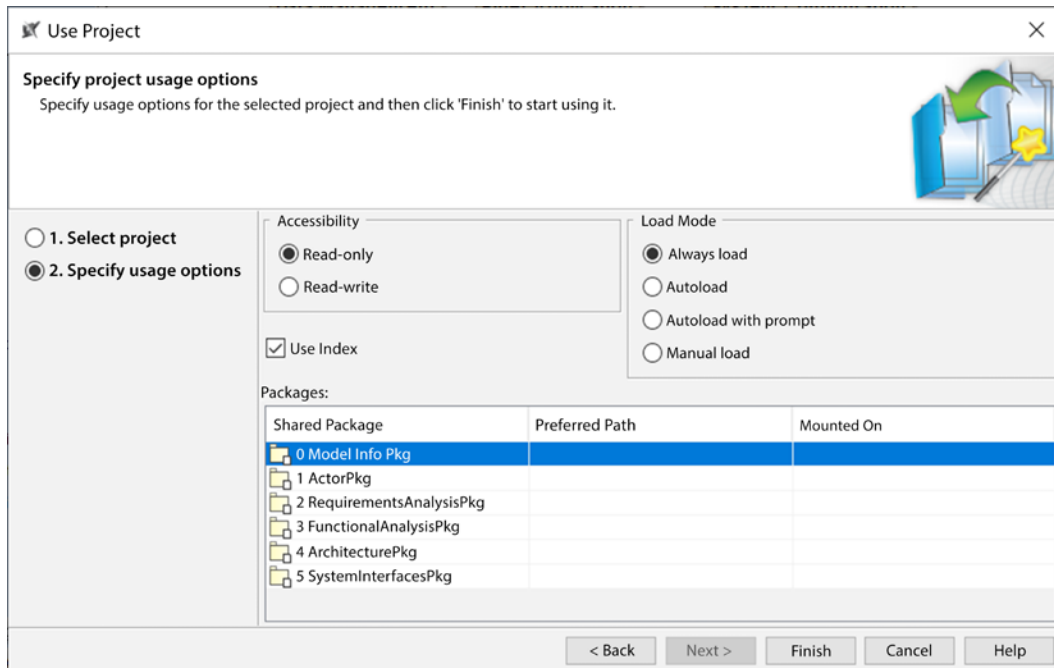


Figure 4.13: Use Project dialog

It is highly recommended that you use the packages as **Read-only**. This will prevent synchronization problems later. If you need to modify the source package, open up the system engineering model and edit it there.

Create the subsystem model

It is assumed that each subsystem is a well-defined, mostly independent entity and will generally be developed by a separate interdisciplinary engineering team. For this reason, each subsystem will be further developed in its own model. In this step, a separate model is created for the subsystem.

This step is repeated for each subsystem.

Structure the subsystem model

A standard canonical structure is created for the subsystem. It consists of two packages, although it is expected that the subsystem team will elaborate on the package structure during the design and implementation of that subsystem. These packages are:

- A **Subsystem Spec Pkg** package:

- This package is intended to hold any requirements and requirements analysis elements the team finds necessary. Further recipes will discuss the creation of refined requirements for the included engineering disciplines and their facets.
- **A Deployment Pkg package:**
 - This package contains what is known as the *deployment architecture* for the subsystem. This is a creation of engineering-discipline-specific facets, which are contributions to the subsystem design from a single engineering discipline. The responsibilities and connections of these facets define the deployment architecture.

This step is repeated for each subsystem.

Add SE Model references to the subsystem model

Each subsystem model has two important references in the **SE Model**:

- **Subsystem package:**
 - In the previous recipe, each subsystem was located in its own package in the **SE Model**, along with its properties and a matrix identifying the requirements it must meet. The package has the original name of the subsystem package in the **SE Model**.
- **Requirements Pkg package:**
 - This is a reference to the entire set of requirements, but the matrix in the subsystem package identifies which of those requirements are relevant to the design of the subsystem.

While in almost all cases, it is preferable to add these other model packages to the subsystem model by reference, it is common to add the subsystem package *by value* (copy). The reason is that in an iterative development process, systems engineering will continue to modify the **SE Model** for the next iteration while the downstream engineering teams are elaborating the subsystem design for the current iteration. As such, the subsystem teams don't want to see new updates to their specification model until the next iteration. By adding the subsystem spec package by copy, it can be updated at a time of the subsystem teams choosing and **SE Model** updates won't be reflected in the subsystem model until the subsystem model explicitly updates the reference. Any relations from other subsystem model elements to the elements in the subsystem spec package will be automatically updated because the reloaded elements will retain the same GUID. However, any changes made to the copied elements will be lost.

Insulation from changes made by system engineers can also be handled by a configuration management tool that manages baselined versions of the referenced models, so there are multiple potential solutions to this issue.

This step is repeated for each subsystem.

Add Shared Model references to the subsystem model

The relevant package to reference in the Shared Model is **Physical Interfaces Pkg**. This will contain the interface definitions and related data used by the subsystems, including the physical data schema.

This step is repeated for each subsystem.

Example

In this example, we will be creating a Shared Model and a subsystem model for subsystems **Comms**, **Electrical Power Delivery**, **Main Computing Platform**, **Mechanical Frame**, **Power Train**, and **Rider Interaction**.

Create the Shared Model

We will put all these models, including the Shared Model, in a folder located in the folder containing the **SE Model**. This folder is named **Subsystem Model** and contains a nested folder, **Shared-Model**.

Structure the Shared Model

It is straightforward to add **Physical Interfaces Pkg** and its nested packages.

Add SE Model references to the Shared Model

It is similarly straightforward to add the references back to the common, requirements, and logical interface packages in the systems engineering model. When this step is complete, the initial structure is as shown in *Figure 4.14*. The grayed-out package names are the shared/referenced packages from the systems engineering model and the darker package names are the ones that are new and owned by the **Shared Model**:

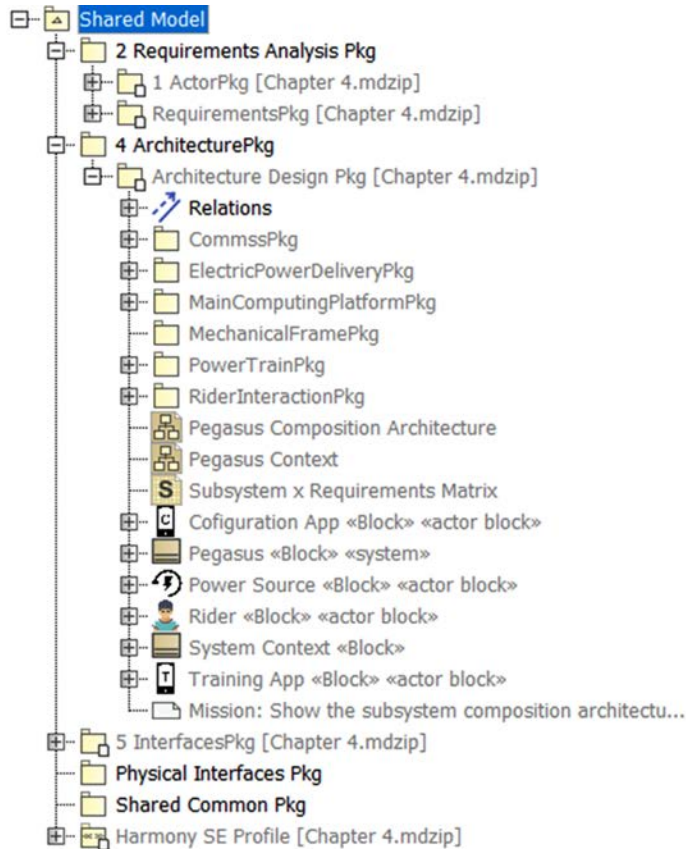


Figure 4.14: Initial structure of the Shared Model

Create the subsystem model

For every subsystem, we will create a separate subsystem model. If we are practicing incremental or agile development, the handoff recipes will be performed multiple times during the project. In this case, it isn't necessary that all of the subsystem will be involved in every increment of the system, so subsystem models only need to be created for the subsystems that participate in the current iteration.

In this example, those subsystems are **Comms**, **Electric Power Delivery**, **Main Computing Platform**, **Mechanical Frame**, **Power Train**, and **Rider Interaction**.

Structure the subsystem model

Although the subsystem models may evolve differently depending on the needs and engineering disciplines involved, they all start life with a common structure. This structure consists of a **Subsystem Spec Pkg** and **Deployment Pkg** packages. If software engineering is involved in subsystem development, then additional packages to support software development are added as well. While it's not strictly a part of the handoff activity per se, I've added a typical initial structure for software development in the **Main Computing Platform** model, shown in *Figure 4.15*.

Add SE Model references to the subsystem model

Each subsystem model must have access to its specification from the **SE Model**; as previously mentioned this can be done by value (copy) or by reference (project usage). In the previous recipe, we organized the SE Model to simplify this step; the information for a specific subsystem is all located in one package nested within the **Architecture Design Pkg**. In our example **SE Model**, the names of these subsystem packages are simply the name of the subsystem followed by the suffix **Pkg**. For example, the package containing the **Main Computing Platform** subsystem is **Main Computing Platform Pkg**.

In addition, each subsystem model must reference the **Requirements Pkg** package so that it can easily locate the requirements it must satisfy.

Add Shared Model references to the subsystem model

Finally, each subsystem needs to reference the common physical interface specifications held in the **Shared Model**:

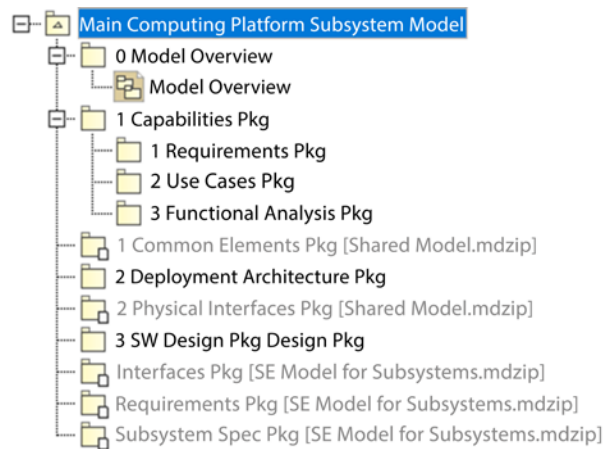


Figure 4.15: Main computing subsystem model organization

Logical to Physical Interfaces

The previous chapters develop interfaces from the functional analysis (*Chapter 2, System Specification*) and the architecture (*Chapter 3, Developing Systems Architecture*). These are all logical interfaces that are defined by a set of logical services or flows. These logical interfaces characterize their logical properties – extent, precision, timeliness, and so on – as metadata on those features. In this book, all services in the logical interface are represented as events that possibly carry information or as flow properties. In this recipe, that information is elaborated in a physical data schema, refining their physical properties.

The subsystem teams need physical interface specifications, since they are designing and implementing physical systems that will connect in the real world. We must refine the logical interfaces to include their implementation detail – including the physical realization of the data – so that the subsystems can be properly designed and be ensured to properly connect and collaborate in actual use.

For example, a logical service specifying a command to enter into configuration mode such as `evSetMode(CONFIGURATION_MODE)` might be established between an actor, such as the **Configuration App**, and the system. The physical interface might be implemented as a Bluetooth message carrying the commanded mode in a specific bit format. The bit format for the message that sends the command is the physical realization of the logical service.

A common system engineering work product is an **Interface Control Document (ICD)**. While I am loath to create a document per se, the goal itself is laudable – create a specification of the system interfaces and their metadata and present it in a way that can be used by a variety of stakeholders. I talk about creating model-based ICDs both on my website (https://www.bruce-douglass.com/_files/ugd/21dc4f_7bf30517b9ca46c59c21eddedf18663c.pdf) and on my YouTube channel (<https://www.youtube.com/watch?v=9pzjMkhFIoM&t=23s>). What should an ICD contain?

An ICD should identify:

- Systems and their connections and connection points
- Interfaces, which in turn contain:
 - Service specifications, including:
 - Service name
 - Service parameters, including their direction (in, out, or inout)
 - Service direction (e.g. provided vs required)

- Description of functionality
- Preconditions, post-conditions, constraints, and invariants
- Flow specifications, including:
 - Flow type
- Data schema for arguments and flows:
 - Type model
 - Metadata, including extent (set of valid values), precision, accuracy, fidelity, timeliness, and other constraints

In SysML models, this means that the primary element will be *Interface Blocks* and their owned features.

Representing an ICD

In SysML, ICD can be visualized as a set of diagrams, tables, matrices, or (in a best case scenario) a combination of all three. One of the useful mission types for block definition diagrams that I use is creating an **Interface Diagram**, which is nothing other than a BDD whose purpose is to visualize some set of interfaces, their owned features, and related metadata. I normally create one such diagram for each subsystem, showing the ports and all the interfaces specifying those ports. It is also easy to create an **Interface Specification Table** (in Cameo, usage of a **Generic Table**) that exposes the same information but in tabular form. An advantage of the tabular format is the ease with which the information can be exported to spreadsheets.

Purpose

The purpose of this recipe is to create physical interfaces and physical data schema and store them in the **Shared Model** so that subsystem teams have a single source of truth for the definition of actor, system, and subsystem interfaces.

Note that any interfaces defined *within* a subsystem are out of scope for this recipe and for the **Shared Model**; this recipe only addresses interfaces between subsystems, the system, and the actors, or between the subsystems and the actors.

Inputs and preconditions

The inputs for the recipe include the logical interfaces and the logical data schema. Preconditions include the construction of the **Shared Model** complete with references to the logical interfaces and data schema in the **SE Model**.

Outputs and postconditions

The outputs from this recipe include both the physical interface specifications and the physical data schema organized for easy import into the subsystem models.

How to do it

The workflow for the recipe is shown in *Figure 4.16*:

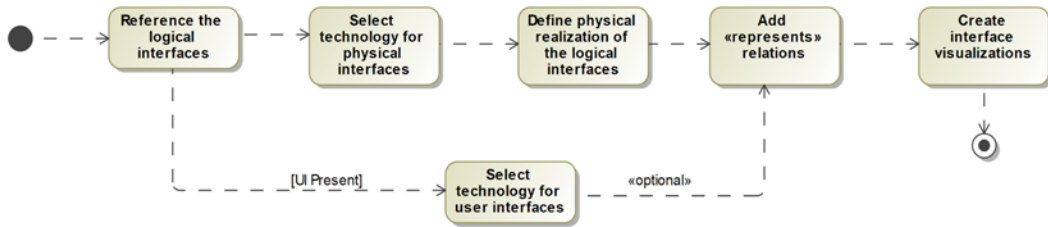


Figure 4.16: Define physical interfaces

Reference the logical interfaces

The physical interfaces we create must satisfy the needs of the already-defined logical interfaces. To start this recipe, we must review and understand the logical interfaces.

Select technology for physical interfaces

Different technological solutions are available for the implementation of physical interfaces. Even for something like a mechanical power transfer interface, different solutions are possible, such as friction, gearing, direct drive, and so on. For electronic interfaces, voltage, wattage, amperage, and phase must be specified. However, software interfaces – or combined electro-software interfaces – are where most of the interface complexity of modern systems reside.

In the previous recipes, we specified a logical interface using a combination of flow properties (mostly for mechanical and electronic interfaces) and signal events (for discrete and software interfaces). This frees us from early concern over technical detail while allowing us to specify the intent and logical content of interfaces. Once we are ready to direct the downstream subsystem teams this missing detail must be added.

Flow properties with mechanical or electronic realization may be implemented as energy or matter flows, or – if informational in nature – perhaps using a software middleware such as **Data Distribution Service (DDS)**. Events are often implemented as direct software interfaces or as messages using a communication protocol.

The latter case is an electro-software interface, but the electronics aspect is dictated by a standard in the physical layer of the protocol definition and while the software messages are constrained by the defined upper layers of the protocol stack. This is assuming that your realization uses a pre-defined communications standard. If you define your own, then you get the joy of specifying those details for your system.

In many cases, the selection of interface technology will be driven by external systems that already exist; in such cases, you must often select from technology solutions those external systems already provide. In other cases, the selection will involve performing trade studies (see the *Architectural Trade Studies* recipe from the previous chapter) to determine the best technical solutions.

Select technology for use interfaces

User interfaces (UIs) are a special case for interface definition. Logical interfaces defining UIs are *virtual* in the sense that our system will not be sending flows or services directly to the human user. Instead, a UI provides sensors and actuators manipulated or used by human users to provide the information specified in the logical interfaces. These sensors and actuators are part of the system design. Inputs from users in a UI may be via touchscreens, keyboards, buttons, motion sensors, or other such devices. Outputs are perceivable by one or more human sensors, typically vision, hearing, or touch. The interactions required of the UI are often defined by a human factors group that does interaction workflow analysis to determine how users should best interact with the system. The physical UI specification details the externally-accessible interactions although not necessarily the internal technical detail. For example, the human factors group may determine that selection buttons should be used for inputs and small displays for output, but the subsystem design team may be charged with the responsibility of selecting the specific kind of buttons (membrane vs tactile) and displays (LED, LCD, plasma, or CRT).

Define the physical realization of the logical interfaces

Once the technology of the physical interfaces is decided, the use of the technology for the interfaces must be defined. This is straightforward for most mechanical and electronic interfaces, but many choices might exist for software. For example, TCP/IP might be a selected technology choice to support an interface. The electronic interfaces are well defined in the physical layer specification. However, the software messages at the application layer of the protocol are defined in terms of datagrams. The interfaces must define the internal structuring of information within the datagram structure.

The basic approach here is to define the message packet or datagram internal data structure, including message identifier and data fields, so that the different subsystems sending or receiving the messages can properly create or interpret the messages. Be sure to add a dependency relation from the physical packet or datagram structure to the logical service being realized. I create a stereotype of dependency named «represents» for this purpose. The relations can best be visualized in matrix form.

Add «represents» relations

It is a good idea to add navigable links from the physical interfaces back to their logical counterparts. This can be done as a «trace» relation, but I prefer to add a stereotype just for this purpose named «represents». I believe this adds clarity whenever you have model elements representing the same thing at different levels of abstraction. The important thing is that from the logical interface, I can identify how it is physically realized, and from the physical realization, I can locate its logical specification.

Create interface visualization

In this step, we create the diagrams, tables, and matrices that expose the interfaces and their properties to the stakeholders.

Example

In this example, we will focus on the logical interfaces shown in *Figure 4.4*. We will define the physical interfaces and physical ports.

Reference the logical interfaces

In the last recipe, we added the **Interfaces Pkg** from the **SE Model** for reference. This means that we can view the content of that package, but cannot change it in the **Shared Model**, only in the owning **SE Model**. The model reference allows use to create navigable relations from our physical schema to its logical counterpart.

Select technology for physical interfaces

The interfaces in *Figure 4.4* have no flow properties so we must only concern ourselves with the event receptions of the interfaces. Specifically, the **Configuration App** and **Training App** communicate with the system via a **Bluetooth Low Energy (BLE)** wireless protocol. The system must also interact with its own sensors via either BLE or ANT+ protocols for heart rate and pedal data, but we are considering those interfaces to be internal to the subsystems themselves and so they will not be detailed here.

The BLE interfaces in our current scope of concern are limited to the **Comm** subsystem and the **Configuration App** and **Training App**.

For internal subsystem interfaces, different technologies are possible, including RS-232, RS-485, the **Control Area Network (CAN)** bus, and Ethernet. Once candidate solutions are identified, a trade study (not shown here but described in the previous chapter's *Architectural Trade Studies* recipe) would be done and the best solution would be selected.

In this example, we select the CAN bus protocol. This technology will be set to connect all subsystems that are software services, such as the interfaces among the **Comm**, **Main Computing Platform**, **Rider Interaction**, and **Power Train** subsystems. The interactions for the **Electric Power Delivery** subsystem will be electronic digital signals. *Table 4.1* summarizes the technology choices:

Interface	Participants	Purpose	Technology
App Interface	Comm subsystem Training app Configuration app	Send Rider data to the Training app for display and analysis. Support system configuration by the Configuration app.	BLE
Pedal Interface	Rider Power Train subsystem	Reliably connect to Rider's shoes via cleats in order to determine the power from the Rider in which position and cadence can be computed.	Look™ compatible clipless pedal fit Standard pedal spindle Pedal-based power meter
Internal Bus	Comms subsystem Rider Interaction subsystem Power Train subsystem Main Computing Platform subsystem Mechanical Frame subsystem	Send commands and data between subsystems for internal interaction and collaboration, including sending information to the Comms subsystem to send to the apps, and sending received information from the apps received by the Comms subsystem to other subsystems.	CAN bus
Power Enable	Power subsystem Rider Interaction subsystem Main Computing Platform subsystem	Signal the power system to turn power on and off.	Digital voltage 0-5V DC

Electrical Power	Electric Power Source Electrical Power Delivery subsystem	Electrical power provided from the external power source to the system	100-240V, 5A, 50-60Hz AC
Electrical Power distribution	Electric Power Delivery subsystem and: <ul style="list-style-type: none"> • High Power: Power Train subsystem • Medium Power: Mechanical Frame subsystem • Digital Power: Rider Interaction subsystem • Main Computing Platform subsystem • Comms subsystem • Power Train subsystem 	Electrical power is transformed and delivered to subsystems in one of three ways: <ul style="list-style-type: none"> • High power for the motor to generate resistance to pedaling for the Rider • Medium power for the Mechanical Frame subsystem to raise or lower the incline • Digital Power provided to the components using digital power 	To be defined by the Electric Power Delivery subsystem team working with the other subsystem teams

Table 4.1: Interface technologies

Of these technologies, BLE and the CAN bus will require the most attention.

Select technology for UIs

There are several points of human interaction in this system that are not in the scope of this particular example handoff, including the mechanical points for fit adjustments, the seat, and the handlebars. In this particular example, we focus on the rider interface via the pedals, the gear shifters, the power button, and the gearing display. The rider metrics and ride performance are all displayed by the **Training App** while the **Configuration App** provides the UI for setting up gearing and other configuration values.

The **Pegasus** features a simple on-bike display for the currently selected gearing and current incline, mocked up in *Figure 4.17*. It is intended to be at the front of the bike, under the handlebars but in easy view of the **Rider**. It sports the power button, which is backlit when pressed, a gearing display, and the currently selected incline.

The gearing display will display the index of the gear selected, not the number of teeth in the gear:



Figure 4.17: On-bike display

To support Shimano and DI shifting emulation, each handlebar includes a brake lever that can be pressed inward to shift up or down for the bike (left side for the chain ring, right side for the rear cassette). In addition, the inside of the shifter has one button for shifting (left side for up, right side for down) to emulate **Digital Indexed (DI)** shifting.

Define the physical realization of the logical interfaces

Let's focus on the messaging protocols as they will implement the bulk of the services. Bluetooth and its variant BLE are well-defined standards. It is anticipated that we will purchase a BLE protocol stack so we only need to be concerned about the structuring of the messages at the 1, that is, structuring the data within the **Payload** field of the **Data Channel PDU** (Figure 4.18):

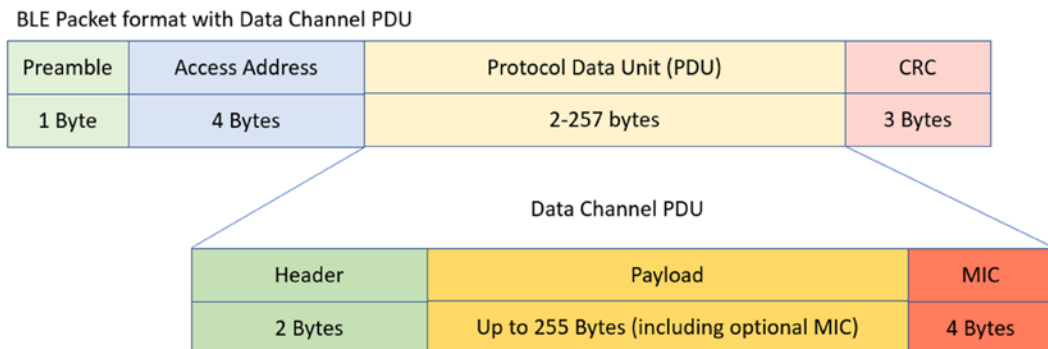


Figure 4.18: BLE packet format

Similarly, for internal bus communication, the CAN bus protocol is well developed with commercial chipsets and protocol stacks available. The CAN bus is optimized for simple data messages and comes in two forms, one providing an 11-bit header and another a 29-bit header. The header defines the message identifier. Because the CAN bus is a bit-dominance protocol, the header also defines the message priority in the case of message transmission collisions. We will use the 29-bit header format so we needn't worry about running out of message identifiers. In either case, the data field may contain up to 8 bytes of data, so that if an application message exceeds that limit, it must be decomposed into multiple CAN bus messages and reassembled at the receiver end. The basic structure of CAN bus messages is shown in *Figure 4.19*:

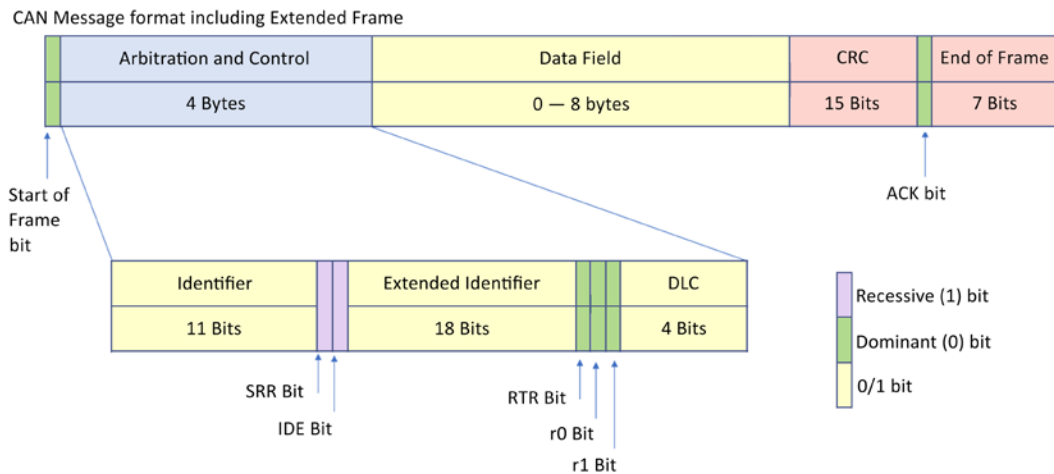


Figure 4.19: CAN bus message format

For CAN bus messages, we'll use an enumeration for the message identifier and define the structure of the data field for interpretation by the system elements.

The basic approach taken is to create base classes for the BLE and CAN bus messages in the **Shared Model**. Each will have an ID field that is an enumeration of the message identifiers; it will be the first field in the BLE PDU. That is, the BLE messages that have no data payload will only contain the message ID field in the BLE message data PDU. The CAN bus messages will be slightly different; the ID field in the message type will be extracted and put into the 29-bit identifier field. Data-less messages will have no data in the CAN bus data fields but will at least have the message identifier.

We won't show all the views of all the data elements, but *Figure 4.20* shows the types for the BLE packets and physical data types:

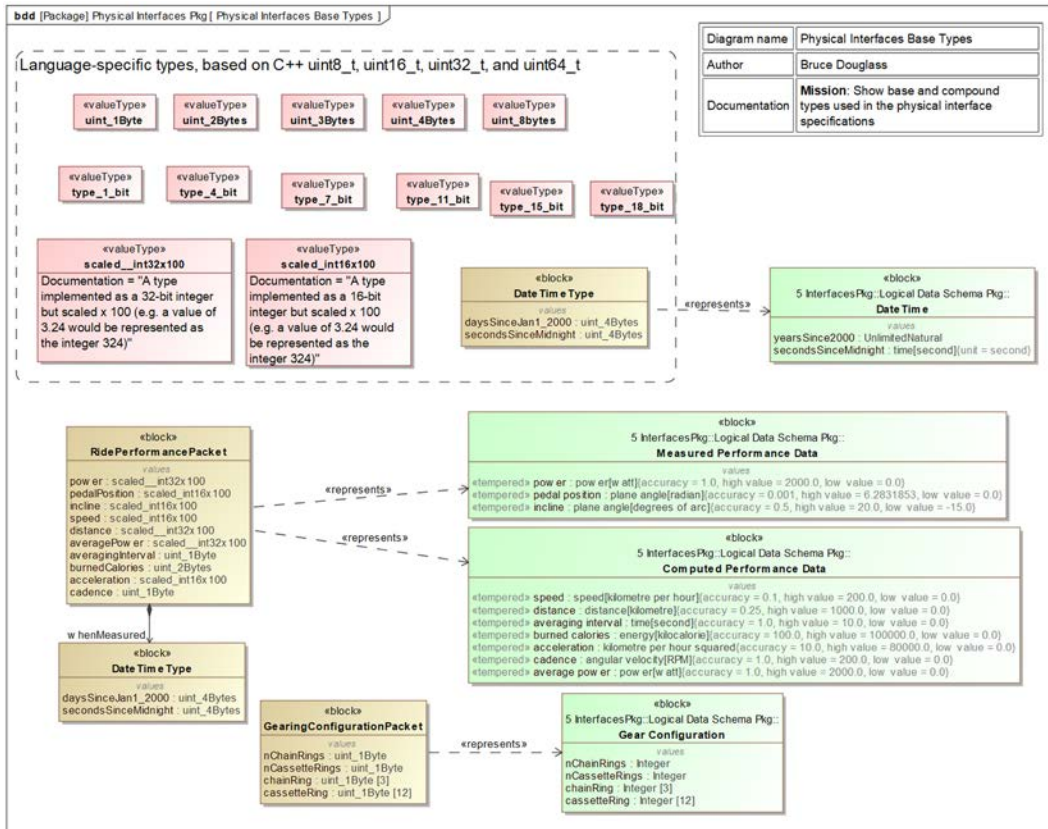


Figure 4.20: Physical data schema for BLE messages

This figure merits a bit of discussion. First, this data schema denotes the physical schema for bits-over-the-air; this is not necessarily how the subsystem stores the data internally for its own use. Secondly, note that the figure denotes the use of some C++ language-specific types, such as `uint_t` as the basis for the corresponding transmission type `uint_1Byte`. Next, note the types `scaled_int32x100` and `scaled_int16x100`. These take a floating-point value and represent them as scaled integers – in the former case, multiplying the value times 100 and storing it as a 32-bit signed integer, and in the latter case, multiplying the value times 100 and storing it in a 16-bit signed integer. The practice of using scaled integers is very common in embedded systems and cuts down on bandwidth when lots of data must be passed around. Metadata specified for the logical interfaces must be copied or added and refined in the physical interfaces.

For example, if the range of power applied by the **Rider** in the logical interface is specified to be in the subrange (0...2000) and units of watts, then this would need to be replicated in the physical data schema as well. Lastly, note how the physical data schema blocks have «represents» relations back to elements in the logical data schema (color-coded to show them more distinctly).

Figure 4.21 shows how these data elements are used to construct BLE packets. The logical data interface blocks for the **Training App** and **Configuration App** are shown in the upper part of the figure (again, with special coloring), along with their services. Each service must be represented by one of the defined packet structures. The structure of the **BLE Packet** class has a **pdu** part, of type **PDU_Base**. **PDU_Base** contains a 2-byte **header**, which indicates the message size as well as a **msg_type** attribute of type **APP_MESSAGE_ID_TYPE**, which is an enumeration of all possible messages. Messages without data, such as **evClose_Session**, can be sent with a **BLE Packet** with a **pdu** part of type **PDU_Base**.

For messages that carry data, **PDU_Base** is subclassed to add appropriate data fields to support the different messages, so that a **BLE Packet** can be constructed with an appropriate subtype of the **PDU_Base** to carry the message data. For example, to send the logical message **evReqSetGearConfiguration**, the **BLE Packet** would use the **PDU_Gearing** subclass of **PDU_Base**:

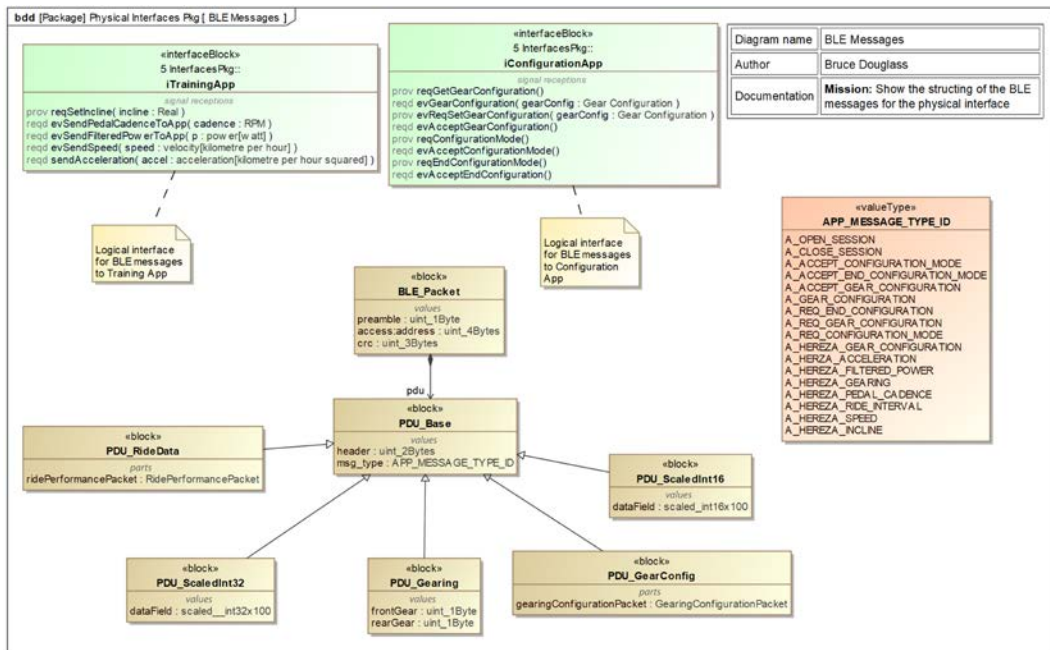


Figure 4.21: BLE Packets

While messages between the system and the apps use Bluetooth, messages between subsystems use an internal CAN bus. The CAN bus physical schema is different in that it must be detailed to the bit level, since there are packet fields of 1, 4, 7, 11, and 18 bits in addition to elements of one or more byte lengths. We'll let the software engineers define the bit-level of the manipulation of the fields and just note, by type, those fields that have special bit lengths where necessary. *Figure 4.22* shows the structure of the CAN bus messages that correspond to the services:

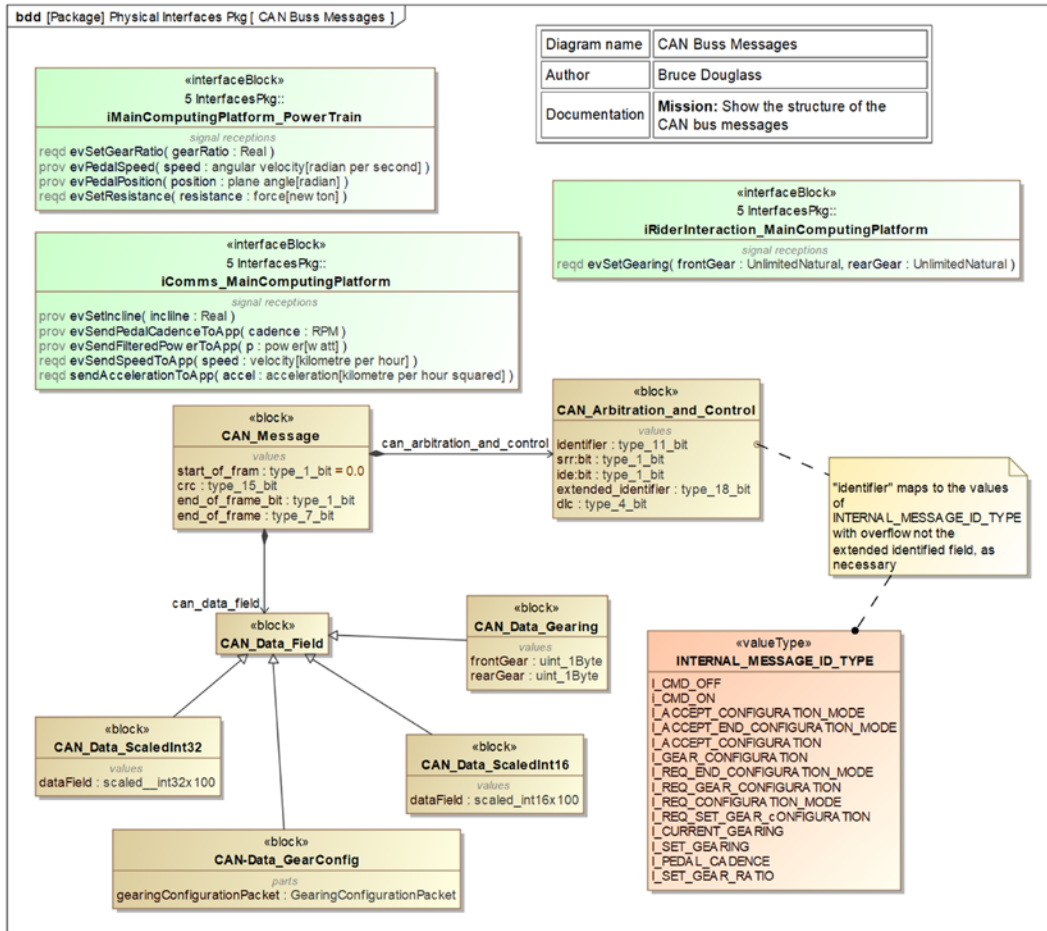


Figure 4.22: CAN bus messages

Add «represents» relations

We must add the «represents» relations so it is clear which Bluetooth or CAN bus messages are used to represent which logical services in the **SE Model** interface blocks.

These relations are best visualized in a matrix. We will define a matrix layout that uses the following settings:

- *From* Element Types: Block
- *To* Element Types: Receptions
- Cell Element Types: Represents the stereotype of dependency

Then we can use that matrix layout to create a matrix view (created in Cameo as a “dependency matrix”) with the **Physical Interfaces Pkg** set to the *From Scope* and **Interfaces Pkg** referenced from the **SE Model**.

When you're all done, you should have a matrix of the relations from the messaging classes to the event receptions in the **SE Model** interface blocks, as shown in *Figure 4.23*:

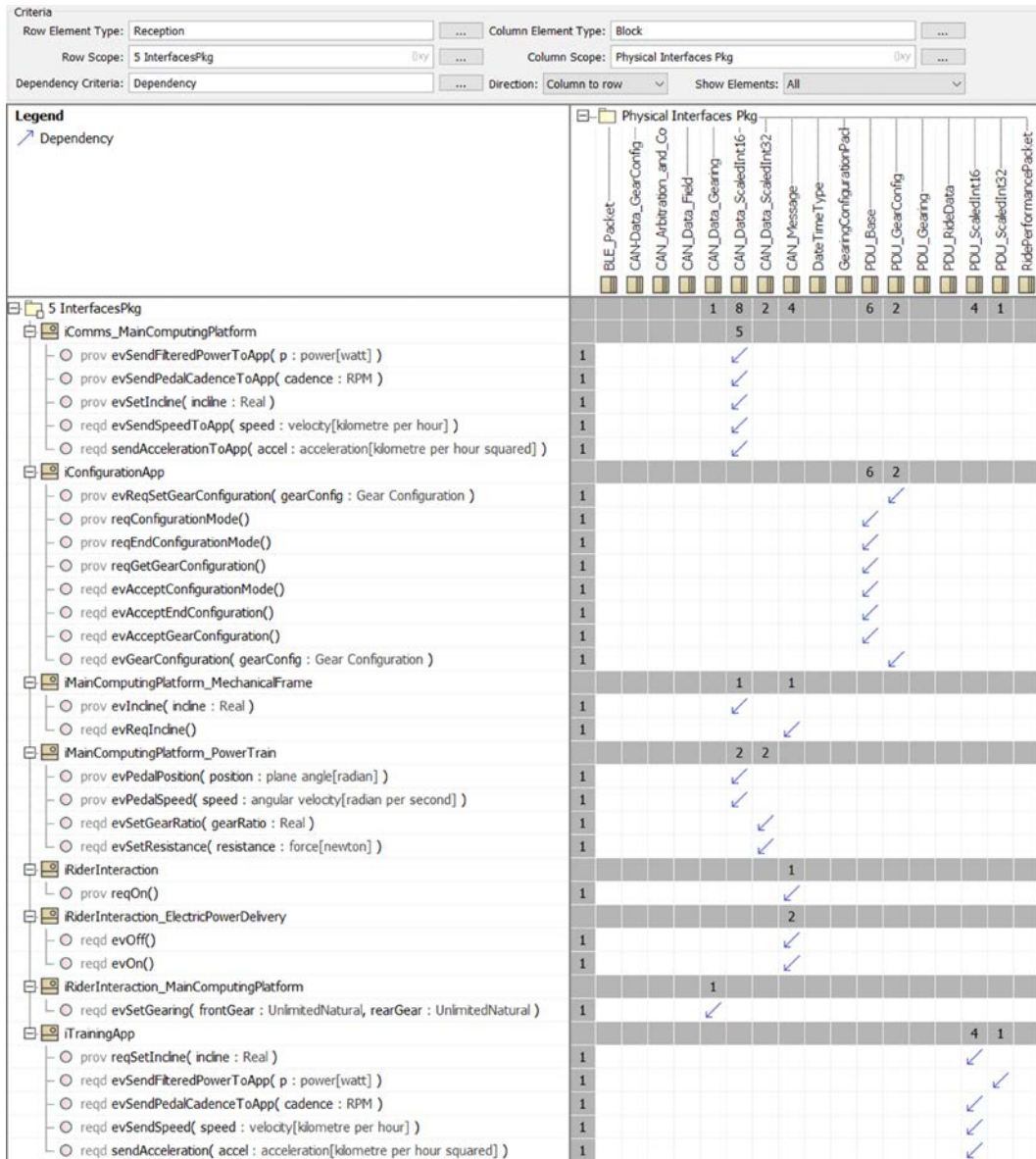


Figure 4.23: Message mapping to logical services

Create interfaces visualization

In this step, we want to create a model-based version of an **Interface Control Document (ICD)**.

In this example, I will show a subsystem interface diagram (Figure 4.24) and an interface specification table (Figure 4.25). In a real project, I would create a subsystem interface diagram for every subsystem in the project increment and the table would be fully elaborated:

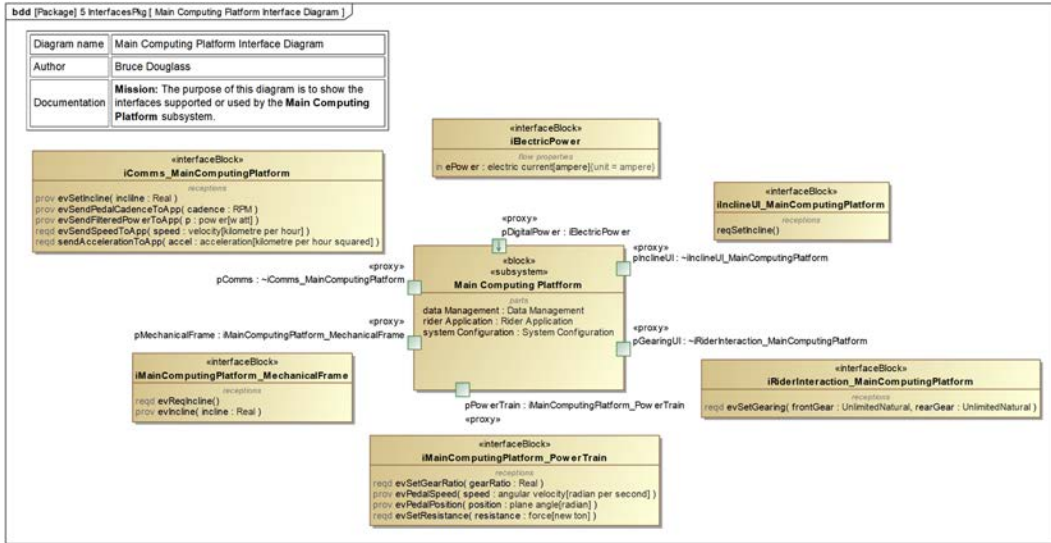


Figure 4.24: Subsystem interface diagram

#	Name	Owned Reception	Attribute
1	iComms_MainComputingPlatform	<ul style="list-style-type: none"> prov evSetIncline(incline : Real) prov evSendPedalCadenceToApp(cadence : RPM) prov evSendFilteredPowerToApp(p : power[watt]) prov evSendSpeedToApp(speed : velocity[kilometre per hour]) req sendAccelerationToApp(accel : acceleration[kilometre per hour squared]) 	
2	iConfigurationApp	<ul style="list-style-type: none"> prov reqGetGearConfiguration() reqd evGearConfiguration(gearConfig : Gear Configuration) prov evReqSetGearConfiguration(gearConfig : Gear Configuration) reqd evAcceptGearConfiguration() prov reqConfigurationMode() reqd evAcceptConfigurationMode() prov reqEndConfigurationMode() reqd evAcceptEndConfiguration() 	
3	iElectricPower		in ePower : electric current[ampere]
4	iInclineUI_MainComputingPlatform	<ul style="list-style-type: none"> req reqSetIncline() 	
5	iMainComputingPlatform_MechanicalFrame	<ul style="list-style-type: none"> reqd evReqIncline() prov evIncline(incline : Real) 	
6	iMainComputingPlatform_PowerTrain	<ul style="list-style-type: none"> reqd evSetGearRatio(gearRatio : Real) prov evPedalSpeed(speed : angular velocity[radian per second]) prov evPedalPosition(position : plane angle[radian]) reqd evSetResistance(resistance : force[newton]) 	
7	iMechanicalFrame_Comms		
8	iPowerTrain		<ul style="list-style-type: none"> in appliedPower : power[watt] in pedalSpeed : angular velocity[radian per second] out resistance : force[newton]
9	iPowerTrain_MechanicalFrame		
10	iRider		<ul style="list-style-type: none"> inout pPowerTrain : iPowerTrain pin interaction : ~iRiderInteraction
11	iRiderInteraction	<ul style="list-style-type: none"> prov reqOn() reqd evOn() reqd evOff() 	
12	iRiderInteraction_ElectricPowerDelivery		
13	iRiderInteraction_MainComputingPlatform	<ul style="list-style-type: none"> reqd evSetGearing(frontGear : UnlimitedNatural, rearGear : UnlimitedNatural) 	
14	iTrainingApp	<ul style="list-style-type: none"> prov reqSetIncline(incline : Real) reqd evSendPedalCadenceToApp(cadence : RPM) reqd evSendFilteredPowerToApp(p : power[watt]) reqd evSendSpeed(speed : velocity[kilometre per hour]) reqd sendAcceleration(accel : acceleration[kilometre per hour squared]) 	

Figure 4.25: Interface specification table

Deployment Architecture I: Allocation to Engineering Facets

The *Federating Models for Handoff* recipe created a set of models: a **Shared Model** and a separate model per subsystem. The current recipe creates what is called the **deployment architecture** and allocates subsystem features and requirements to different engineering disciplines. Once that is done, the real job of software, electronic, and mechanical design can begin, post-handoff. In this chapter, we will separate two key aspects of the deployment architecture into different recipes. The first will deal with the allocation of subsystem features to engineering facets. The next recipe will focus on the definition of interdisciplinary interfaces enabling those facets to collaborate.

Deployment architecture

Chapter 3, Developing Systems Architecture, began with a discussion of the “Six Critical Views of Architecture.” One of these – the *Deployment Architecture* – is the focus of this and the next recipes. The deployment architecture is based on the notion of *facets*. A facet is a contribution to a design that comes from a single engineering discipline (*Figure 4.26*). We will not design the internal structure or behavior of these facets, but will refer to them collectively; thus, we will not identify here software components, or electrical parts, but rather simply refer to all the contributions from these as the “software facet” or “electronics facet.” A typical subsystem integrates a number of different facets, the output from engineering in disciplines such as:

- Electronics:
 - Power electronics
 - Motor electronics
 - Analog electronics
 - Digital electronics
- Mechanics:
 - Thermodynamics
 - Materials
 - Structural mechanics
 - Pneumatics
 - Hydraulics
 - Aerodynamics
 - Hydrodynamics
- Optics

- Acoustics
- Chemical engineering
- Software:
 - Control software
 - Web/cloud software
 - Communications
 - Machine learning
 - Data management

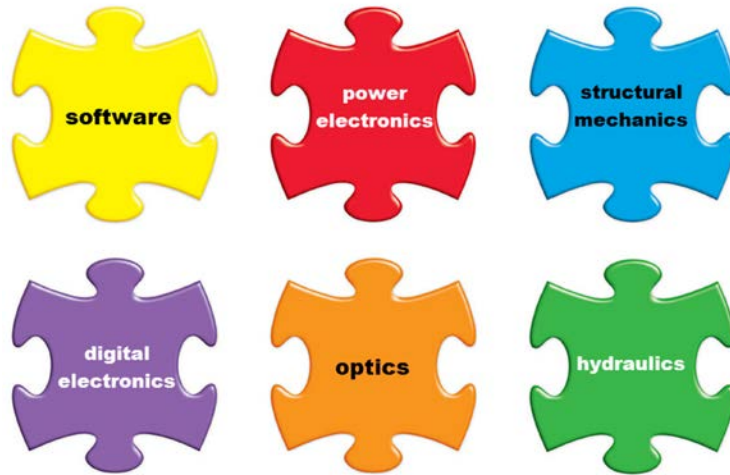


Figure 4.26: Some engineering facets

These disciplines may result in facets at a high-level of abstraction – such as an electronics facet – or may result in more detailed facets – such as power, motor, and digital electronics, depending on the needs and complexity of the subsystem.

A subsystem team is generally comprised of multiple engineers of each discipline working both independently on their aspect of the design and collaboratively to produce an integrated, functioning subsystem.

Creating a deployment architecture identifies and characterizes the facets and their responsibilities in the scope of the subsystem design. This means that the subsystem features – subsystem functions, information, services, and requirements – must be allocated to the facets.

This makes it clear to the subsystem discipline-specific engineers what they need to design and how it contributes to the overall subsystem functionality.

The deployment architecture definition is best led by a system engineer but must include input and contributions from discipline-specific engineers to carry it forward. This is often called an **Interdisciplinary Product Team (IPT)**. Historically, a common cause of project failure is for a single engineering discipline to be responsible for the creation of the deployment architecture and allocation of responsibilities. Practice clearly demonstrates that an IPT is a better way.

The system engineering role in the deployment architecture is crucial because system engineers seek to optimize the system as a whole against a set of product-level constraints. Nevertheless, it is a mistake for systems engineers to create and allocate responsibilities without consulting the downstream engineers responsible for detailed design and implementation. It is also a (common) mistake to let one discipline make the deployment decisions without adequate input from the others. I've seen this on many projects, where the electronics designers dictate the deployment architecture and end up with a horrid software design because they didn't adequately consider the needs of the software team. It is best for the engineers to collaborate on the deployment decisions, and in my experience, this results in a superior overall design.

Purpose

The purpose of this recipe is to create the deployment architecture and allocate subsystem features and requirements to enable the creation of the electronic, mechanical, and software design.

Inputs and preconditions

The preconditions are that the subsystem architecture has been defined, subsystems have been identified, and system features and requirements have been allocated to the subsystems.

Inputs include the subsystem requirements and system features – notably system data, system services, and system functions – that have been allocated to the subsystem.

Outputs and postconditions

The output of this recipe is the defined deployment architecture, identified subsystem facets, and allocation of system features to those facets.

The output is the updated subsystem model with those elements identified and allocated.

How to do it

Figure 4.27 shows the recipe workflow. This recipe is similar to the *Architectural Allocation* recipe in Chapter 3, *Developing Systems Architecture*, except that it focuses on the facets within a subsystem rather than on the subsystems themselves. This recipe is best led by a system engineer but performed by an interdisciplinary product team to optimize the deployment architecture:

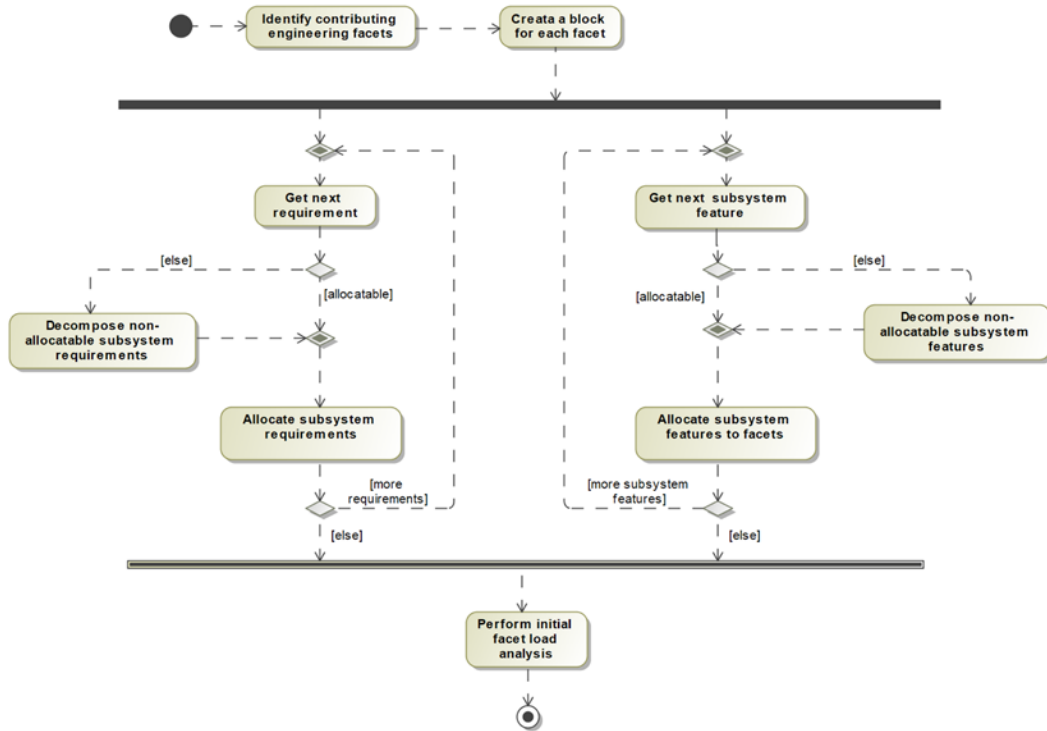


Figure 4.27: Create the deployment architecture

Identify contributing engineering facets

The first step of the recipe is to identify the engineering disciplines involved. Some subsystems might be mechanical only, electrical-mechanical, software-electronic, or virtually any other combination. Any involved engineering discipline must be represented so that their facet contribution can be adequately characterized. For an embedded system, a software-centric subsystem typically also includes electronic computing infrastructure (a digital electronic facet) to provide hardware services to support the functionality. The capacities of the electronics facet, including memory size, CPU throughput, and so on, are left to negotiation among the coordinating engineering disciplines.

This topic is discussed in a bit more detail in the next recipe, *Deployment Architecture II: Interdisciplinary Interfaces*.

Create a block for each facet

In the deployment model, we will create a block for each facet. We will not, in general, decompose the facet to identify its internal structure. That is a job for discipline-specific engineers who have highly developed skills in their respective subject matters. It is enough, generally, to create a separate block for each facet. Each facet block will serve as a container or collector of specification information about that facet for the downstream engineering work.

Decompose non-allocatable requirements

Requirements can sometimes be directly allocated to a specific facet. In other cases, it is necessary to create *derived requirements* that take into account the original subsystem requirements and the specifics of the subsystem deployment architecture. In this latter case, create derived requirements that can be directly allocated to subsystems. Be sure to add «deriveReq» relations from the derived requirements back to their source subsystem requirements. These derived requirements can either be stored in the subsystem model, the system model, or in a requirements management tool, such as IBM DOORS™.

Allocate requirements to facets

Allocate system requirements to the identified facets. In general, each requirement is allocated to a single facet, so in the end, the requirements allocated to the subsystem are clearly and unambiguously allocated to facets. The set of allocated subsystem requirements after this step is normally known as **software, electronic, or mechanical requirements**. It is crucial that the engineers of the involved disciplines are a part of the allocation process.

Decompose non-allocatable subsystem features

Some subsystem features – which refer to operations, signal receptions, flows, and data – can be directly allocated to a single facet. In practice, most cannot. When this is the case, the feature must be decomposed into engineering-specific-level features that trace back to their subsystem-level source feature but can be directly allocated to a single facet.

Allocate subsystem features to facets

Each subsystem function now becomes a service allocated to a single facet OR it is decomposed to a set of services, each of which is so allocated. Subsystem flows and data must also be allocated to facets as well.

Perform initial facet load analysis

This step seeks to broadly characterize the size, capacity, and other summary quantitative properties of the facets. The term *load* means different things in different disciplines. For mechanical facets, it might refer to weight or shear force. For digital electronics, it might refer to CPU throughput and memory size. For power electronics, it might mean the maximum available current. For software, it might mean volume (roughly, “lines of code”), nonvolatile storage needs, volatile memory needs, or message throughput. The important properties will also differ depending on the nature of the system. Helicopters, for example, are notoriously weight-sensitive, while automobiles are notoriously component price-sensitive (meaning they want to use the smallest CPUs and the least memory possible). These system characteristics will drive the need for the quantification of different system properties.

These properties will be estimates of the final product qualities but will be used to drive engineering decisions. Digital electronics engineers will need to design or select CPU and memory hardware, and they need a rough guess of the needs or requirements of the software to do so. It is far too common that a lack of understanding leads to the design of underpowered computing hardware resulting in decreased software (and therefore, system) performance. Is a 16-bit processor adequate or does the system need a pair of them or a 32-bit CPU? Is 100Kb of memory adequate or does the system need 10Mb? Given these rough estimates, downstream design refines and implements these properties.

Estimating the required capacity of the computing environment is difficult to do well and a detailed discussion is beyond the scope of this book. However, it makes sense to introduce the topic and illustrate how it might be applied in this example. Let’s consider memory sizing first.

Embedded systems memory comes in several different kinds, each of which must be accessible by the software (i.e. part of the software “memory map”). Without considering the underlying technology, the kinds of memory required by embedded software are:

- Non-reprogrammable non-volatile memory:

This kind of memory provides storage for code that can never be changed after manufacturing. It provides boot-loading code and at least low-level operating system code.

- Programmable non-volatile memory:

This kind of memory provides storage for software object code and data that is to be retained across power resets. This can be updated and rewritten by program execution or via **Over-the-Air (OTA)** updates.

- Programmable read-write memory for software object code execution:
Because of the relatively slow access times for non-volatile memory, it is not uncommon to copy software object code into normal RAM for execution. Normally, the contents of this memory are lost during power resets.
- Programmable read-write memory for software data (heap, stack, and global storage):
This memory provides volatile storage for variables and software data during execution. Generally, the contents of this memory are lost during power resets.
- Electronic registers (“pseudo-memory”):
This isn’t really memory, per se, as it refers to hardware read-only, write-only, and read-write registers used to interface between the software and the digital electronics.
- Interrupt vector table:
This isn’t a different kind of memory as it may be implemented in any of the above means but must be part of the memory map.

A typical approach is to estimate the need for each of the above kinds of memory and then construct a memory map, assigning blocks of addresses to the kinds of memory. It is common to then add a percentage beyond the estimated need to provide room to grow in the future.

CPU capacity can be measured in many ways. CPU capacity estimating can be performed by comparing the computation expectations of a new system based on throughput measurements of existing systems. An alternative is to base it on experimentation: write a “representative” portion of the software, run it on the proposed hardware platform, measure the execution time, bandwidth or throughput, and then scale it to your estimated software size. You can even do “cycle counting” by determining the CPU cycles needed to perform critical functionality on a given CPU, and then compute the CPU speed as a cycle rate fast enough to deliver the necessary functionality within the timeliness requirements. Of course, there are many other considerations that go into a CPU decision including cost, availability, longevity, and development support infrastructure.

Example

We will limit our example to a single subsystem identified in the last chapter: the **Power Train** subsystem.

Identify contributing engineering facets

The **Power Train** subsystem is envisioned to have mechanical, electronic, and software aspects.

Create a block for each facet

Figure 4.28 shows the blocks representing these facets while *Figure 4.29* shows how they connect. Note the «software», «electronics», and «mechanical» stereotypes added to the facet blocks. I defined these stereotypes in the **Shared Model::Shared Common Pkg** package so that they are available to all subsystems. This is an example of the use of stereotypes to indicate a “special kind of” modeling element.

Also note the connection points for those facets. There are two pairs of ports between the **Software** and **Electronics** blocks, one for communications (it is anticipated that the electronics will provide an interface to the CAN bus) and one for interaction with the electronics of the power train. This separation is not a constraint on either the software or electronics design but represents the connections between the facets being really independent of each other.

There is both a port pair and an association between the **Electronic** and **Mechanical** blocks on the BDD and a corresponding connector between their instances on the IBD. This is a personal choice of how I like to separate *dynamic* and *static* connections between these facets by modeling them with ports and direct associations, respectively. By *dynamic*, I am referring to connections that convey information or flows during system operation.

By *static*, I am referring to connections that do not, such as the physical attachments of electronic components to the mechanical power train with bolts or screws:

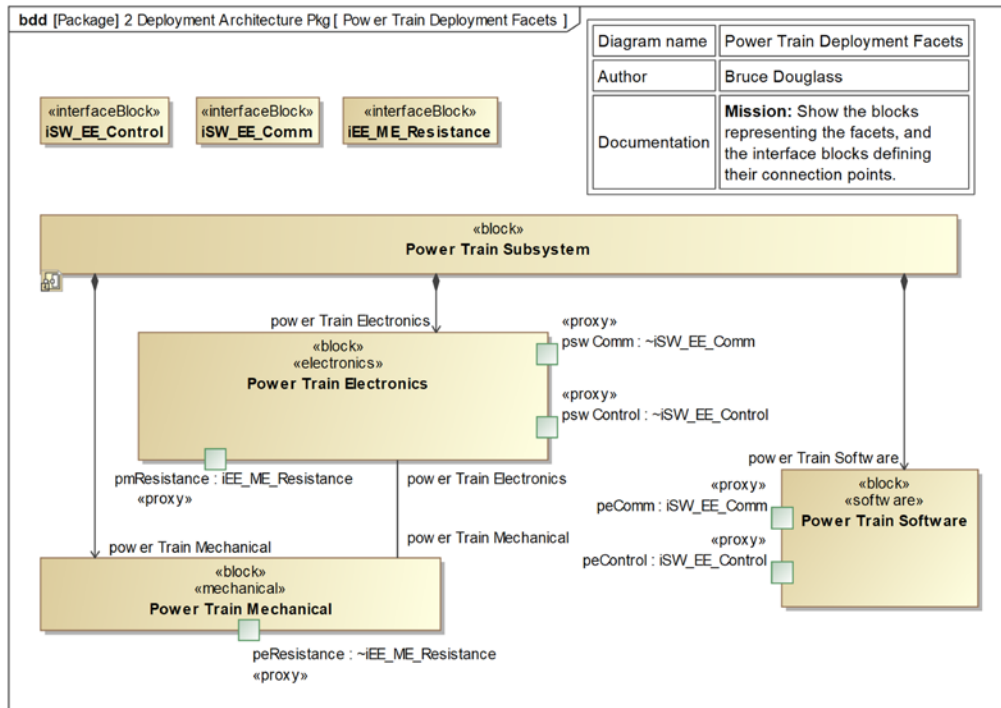


Figure 4.28: Deployment Architecture BDD

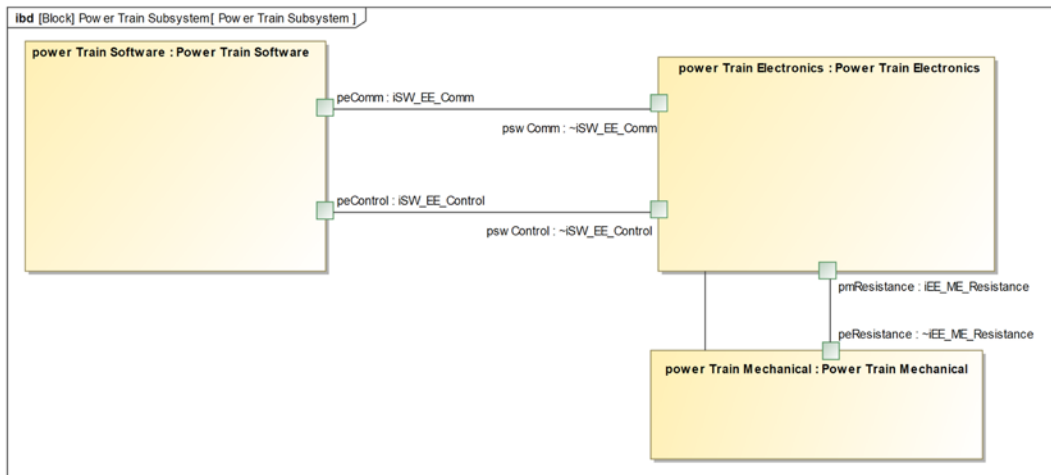


Figure 4.29: Connected deployment architecture IBD

Decompose non-allocatable requirements

These facets provide responsible elements to which requirements may be allocated. As mentioned, many requirements must be decomposed into derived requirements prior to allocation.

One issue is where to place such requirements. If you are using a third-party requirements management tool such as DOORS, then clearly, this tool should hold those requirements. If you are instead managing the requirements directly in your model, then there are a couple of options: the **Requirements Pkg** package in the **SE Model** or in the subsystem model. I personally prefer the latter, but valid arguments can be made for the former. In this case, I will create a **Capabilities::Requirements Pkg** package inside the **Power Train Subsystem Model** to hold these requirements.

Before we can decompose requirements for this subsystem, we need to highlight the relevant requirements, i.e. the ones to which the **Power Train** subsystem has an «allocate» or «satisfy» relation. The **Power Train Pkg**, imported from the **SE Model**, has these relations, and the **Requirements Pkg**, also loaded from the **SE Model**, has the requirements. The appropriate requirement may be easily identified by creating a relation map in Cameo for the **Power Train** block (right-click the block and select **Related elements > Create Relation Map**) or by building a matrix. Both are shown in *Figure 4.30*:

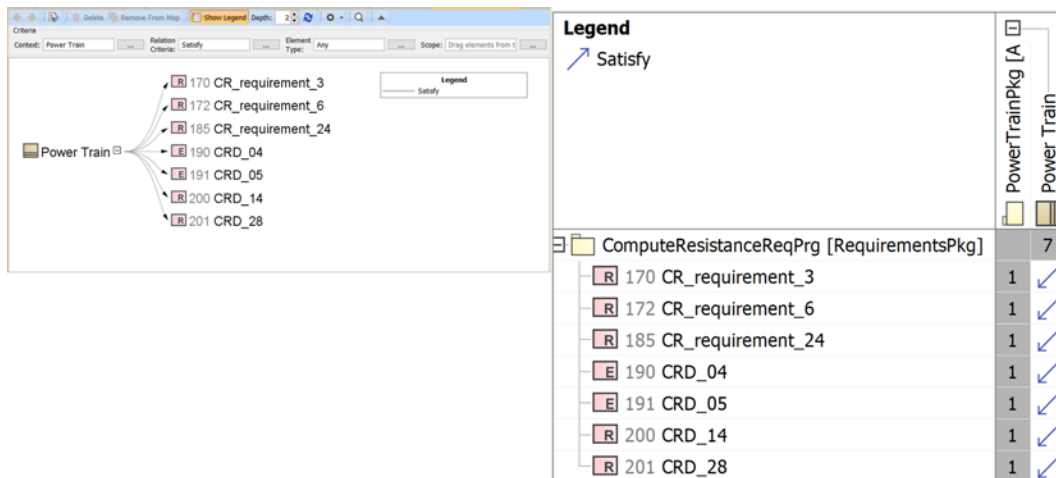


Figure 4.30: Finding requirements allocated to the subsystem

Having identified the relevant requirements, it is a simple matter to build a requirements diagram exposing just those requirements in the **Subsystem Requirements Pkg** package (Figure 4.31):

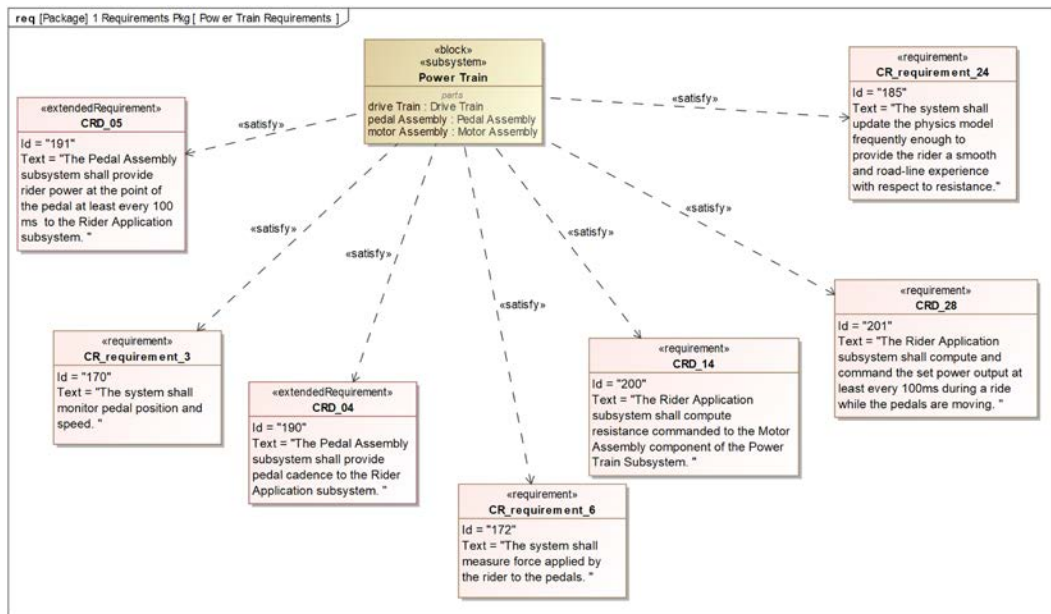


Figure 4.31: Requirements allocated to the power train subsystem

When working in diagrams, adding derived requirements is straightforward. For example, look at Figure 4.32. In this figure, which works with a subset of the subsystem requirements, we see the facet requirements with «deriveReq» relations. I've also added the facet stereotypes to clarify the kind of requirement being stated:

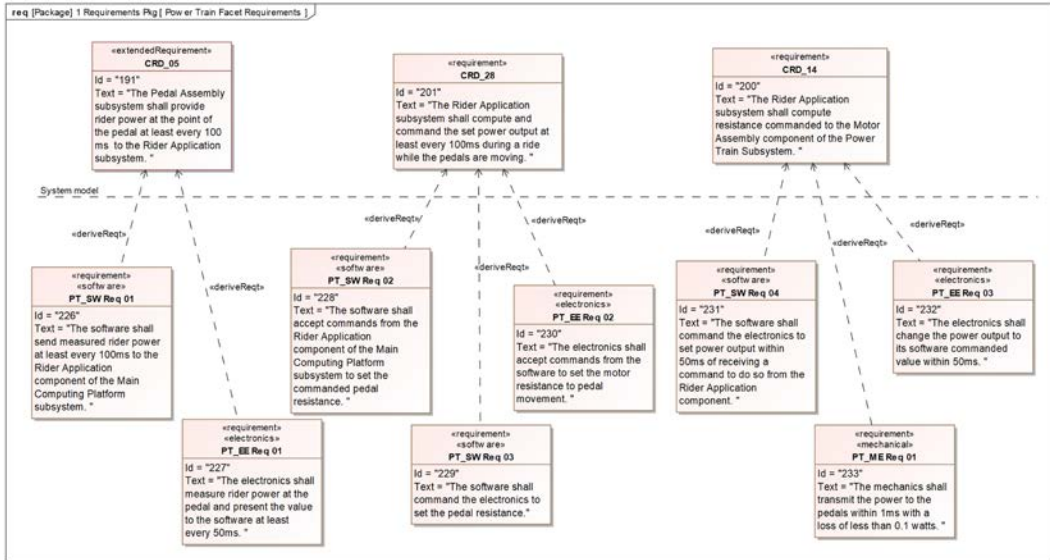


Figure 4.32: Diagrammatically adding facet requirements

The other subsystem requirements are decomposed in other requirements diagrams. The derived requirements are summarized in *Table 4.2*:

Requirement Name	Specification	Derived From
EEReq_01	The electronics will measure rider power at the pedal and present the value to the software at least every 50 ms.	CRD_05, CR_requirement_6
EEReq_02	The electronics shall accept commands from the software to set the resistance to the motor.	CRD_14
EEReq_03	The electronics shall change the power output to its software-commanded value within 50 ms.	CRD_28, CRD_27
EEReq_04	The electronics shall measure pedal position within 1 degree of accuracy.	CR_requirement_3
EEReq_05	The motor shall produce resistance to be applied to the pedal.	CRD_26
EEReq_06	The electronics shall measure the pedal position at least every 10 ms.	CR_requirement_3
EEReq_07	The electronics shall provide the measured pedal position to the software at least every 10 ms.	CR_requirement_3
EEReq_08	The electronics shall accept commands from the software to set the resistance to the rider, resulting in power to the pedals in the range of 0 to 2,000W.	CRD_24

EEReq_09	The electronics shall produce resistance resulting in the command power with an accuracy of +/- 1 watt.	CRD_24
EEReq_10	The electronics shall update the resistance at the pedal to a commanded value within 10ms.	CRD_24
MEReq_01	The mechanical drive train shall transmit the motor resistance to the pedal without rider-perceivable loss.	CRD_26
MEReq_02	The mechanical drive train shall deliver resistance to the pedal with a power loss of < 0.2W.	CRD_24
SWReq_01	The software will send measured rider power at least every 100 ms to the Rider Application Subsystem.	CRD_05, CR_requirement_6
SWReq_02	The software shall accept commands from the Rider Application to set the commanded pedal resistance.	CRD_14
SWReq_03	The software shall command the electronics to set the pedal resistance.	CRD_14
SWReq_04	The software shall command the electronics to set power output within 50ms of receiving a command to do so from the Rider Application subsystem.	CRD_28, CRD_27
SWReq_05	The software shall read a measured pedal position from the electronics at least every 10ms.	CR_requirement_3
SWReq_06	The software shall compute pedal cadence and pedal speed from pedal position changes at least every 20ms.	CR_requirement_3
SWReq_07	The software shall convey pedal position and pedal cadence at least every 50ms to the Rider Interaction subsystem.	CRD_04

SWReq_08	The software shall receive resistance commands from the Rider Application subsystem and convey them to the electronics within 10ms.	CRD_24
SWReq_09	The software shall reject commanded resistance, resulting in power outputs that fall outside the range of 0–2,000W, and return an error message to the Rider Application subsystem.	CRD_24
SWReq_10	The software shall convey commanded resistance to the electronics within 50ms of receipt from the Rider Application subsystem.	CRD_24

Table 4.2: Derived facet requirements table

Allocate requirements to facets

The next step in the recipe is to allocate the requirements to the facets. *Figure 4.33* shows the use of the «satisfy» relations from the facets to the requirements:

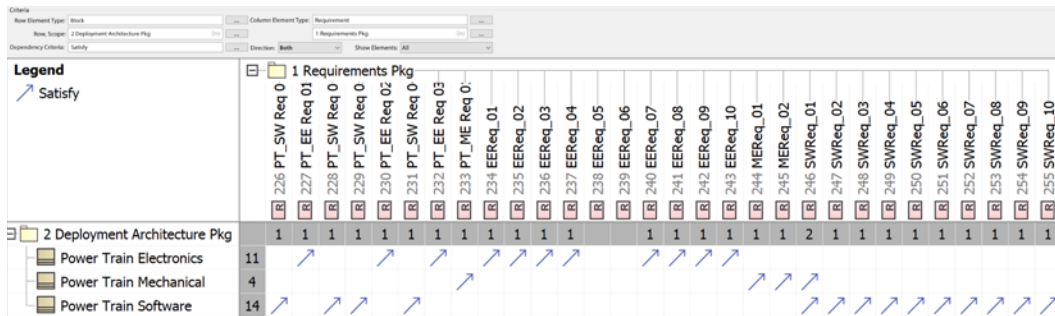


Figure 4.33: Matrix of facet and subsystem requirements

Decompose non-allocatable subsystem features

Figure 4.34 shows the subsystem block features to allocate. On the right side of the figure is the logical **Power Train** subsystem block from the **SE Model**, while on the left is the physical version.

There are a few differences:

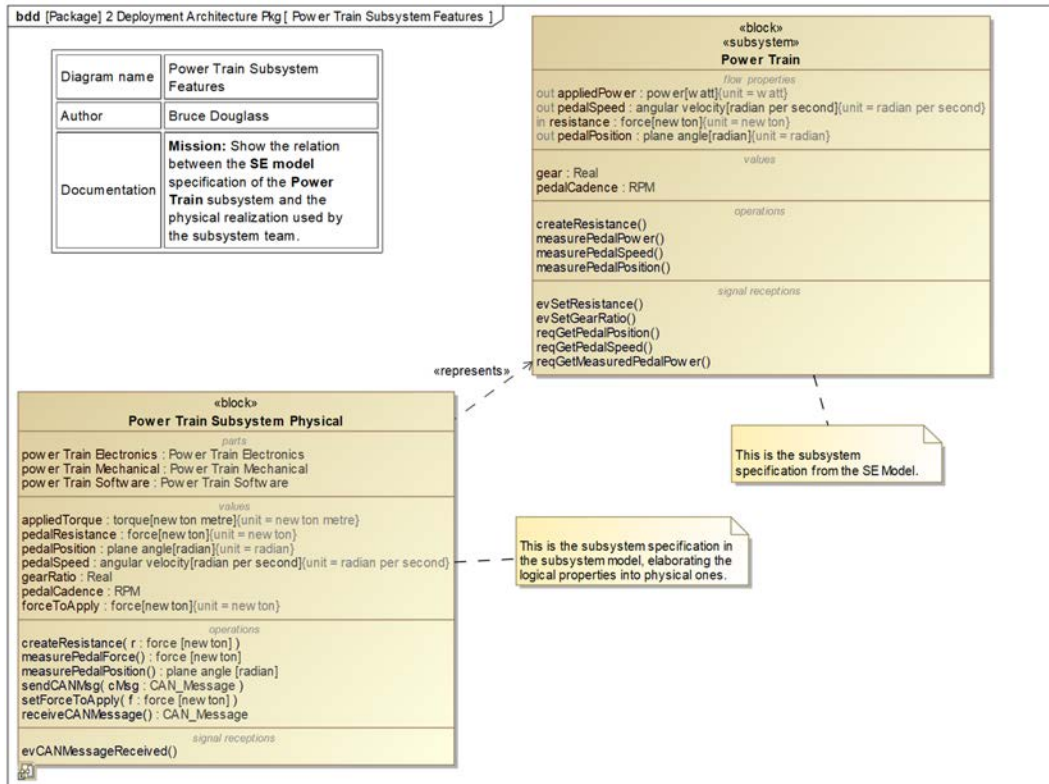


Figure 4.34: Subsystem block features to allocate to facets

First, note that the physical version contains no flow properties because all the flow properties for this block represent things that the subsystem’s internal sensors will measure. Secondly, the gearing and computation of gear ratio are really managed by the **Main Computing Platform** subsystem and all this subsystem cares about is the **gear ratio** itself. Third, all of the event receptions, specifying the logical services available across the interface, are summarized by a single event – **evCanMessageReceived()** – and two operations – **sendCANMessage()** and **receiveCANMessage()**.

All of the value properties in the **Power Train Subsystem Physical** block must be decomposed into elements in the electronics and software facets. For example, **applied Torque** is (perhaps) measured by the hardware in a range of 0 to 10,000 in a 16-bit hardware register. The software must convert that value to a scaled integer value (**scaled_int32_x100**) that represents applied force in watts (ranging from 0 to 2,000) for sending in a CAN message.

We want the software in the **Power Train Subsystem Physical** to encapsulate and hide the motor implementation from other subsystems to ensure robust, maintainable design in the future. This means that the information is represented in both the software and the electronic facets. The expectation is that the software will be responsible for scaling, manipulating, and communicating these values with the **Main Computing Platform**, while the electronics will be responsible for setting or monitoring device raw data and presenting it to the software.

Figure 4.35 and Figure 4.36 show the derived value properties and operations respectively. I added a «deriveFeature» stereotype for the dependency between the derived features and its base feature in the **Power Train Subsystem Physical** block. These figures show a matrix of such dependencies from the facet properties to the subsystem block properties. The column elements are properties of the facet blocks – **Power Train Electronics**, **Power Train Mechanicals**, and **Power Train Software** – while the rows are the value properties allocated to the subsystem itself. This can be shown diagrammatically but the matrix form is easier to read.

Figure 4.35 shows how some elements (such as **gearRatio**) are allocated only to a single facet while others (such as **appliedTorque**) are decomposed and allocated to multiple facets:

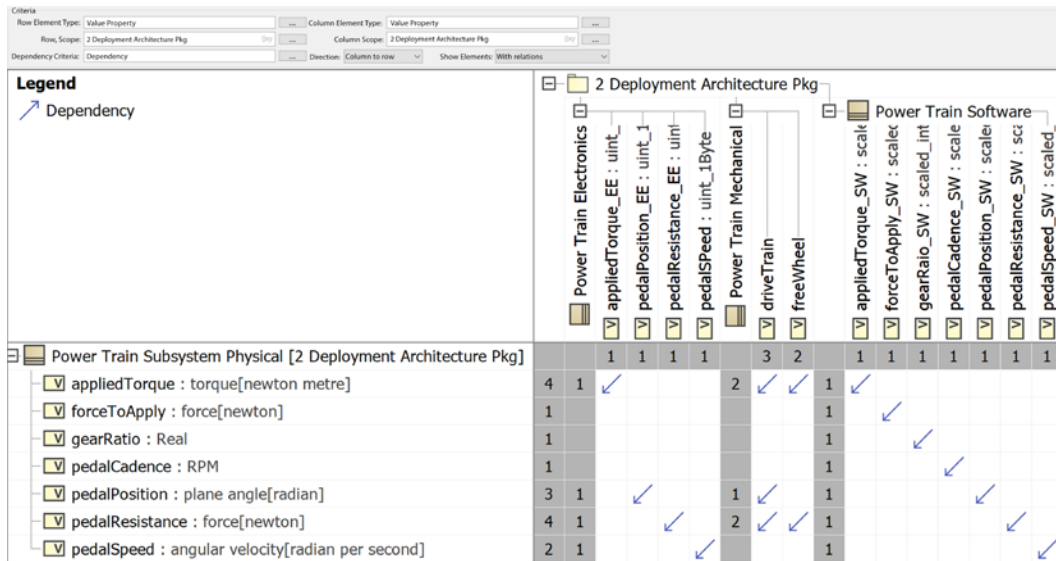


Figure 4.35: Derived facet value properties matrix

The subsystem **pedal Cadence** value property is only represented in the software because the software will compute it from the value property **pedal Speed**, which is provided by the electronics.

Similarly, the **gear ratio** is used by the software to determine how much resistance should be applied to the pedal given the **pedal cadence** and desired **pedal resistance**. *Figure 4.36* shows a similar mapping but of operations from the software and electronics facets to the operations allocated to the **Power Train Subsystem Physical** block:

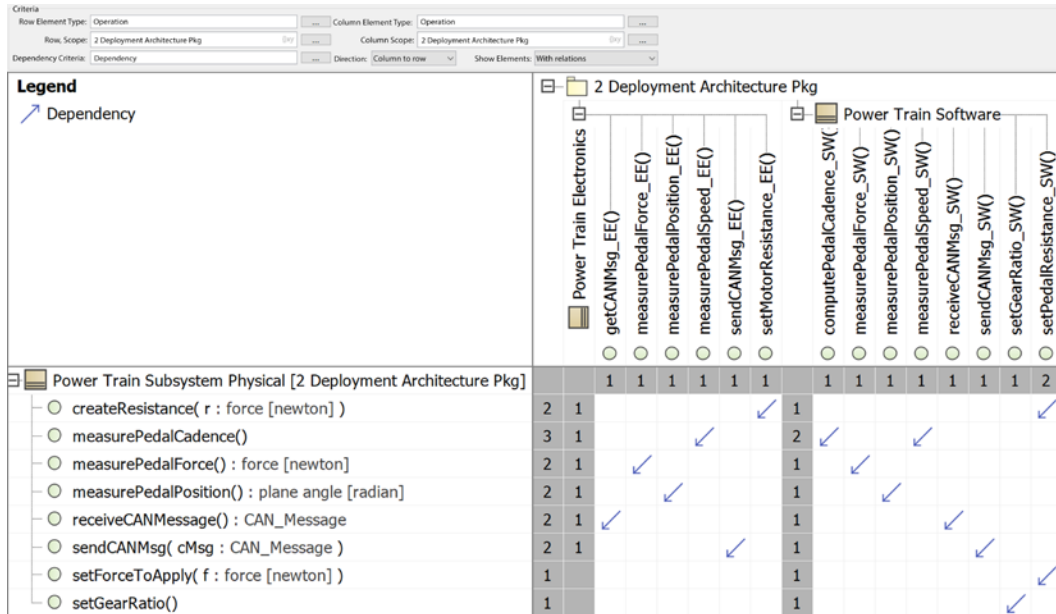


Figure 4.36: Derived facet functions matrix

The allocation of subsystem data and function is not meant to overly constrain the design of the facets; they are internal features that the design of these features must support, rather than the actual design of those features. Facets should feel free to design those features in whatever means makes the most sense to them. The primary reason for the allocation of the system features to facets is to be clear about the data and functionality that the facet design is expected to deliver.

Allocate subsystem features to facets

In this example, the identification of the derived facet feature was performed concomitantly with their allocation, as shown in the previous two figures.

Perform initial facet load analysis

The last step in the recipe is to determine facet capacity. In this example, the primary concerns are the power of the motor and the computational capacity and memory size of the digital electronics.

The motor power is specified in the requirements; the system is expected to deliver up to 2,000W of power via resistance to the rider pedal motion.

In this case, we determine that a 32-bit STM32F2 ARM processor running at 120MHz with 1MB ROM and 1MB RAM is the best fit for our needs and expected future expansion.

Deployment Architecture II: Interdisciplinary Interfaces

One of the most common points of failure in the development of embedded systems is inadequately nailing down the interdisciplinary interfaces, especially the electronics-software interfaces. These interfaces inform related disciplines about common expectations as to the structure and behavior of those interfaces. Left to their own devices (so to speak), software engineers will develop interfaces that are easy for the software implementation while electronics engineers will develop interfaces that simplify the electronic design. Interdisciplinary interfaces are best developed cooperatively with all the contributing engineering disciplines present.

In my experience it is best to develop these interfaces early to set expectations, and freeze these interfaces under configuration management. This is important even if some of the details may change later. When it becomes obvious during the development of a facet that an interface needs to be modified, then thaw the interface from configuration management, discuss with all the stakeholders of that interface, agree upon an appropriate revision, and refreeze it in the CM tool. Failure to define the interfaces early will inevitably lead to significant downstream rework and often to suboptimal designs and implementations. It's an annoyingly common yet easily avoidable problem.

Purpose

The purpose of this recipe is to detail the interactions between the engineering facets in a system or subsystem design.

Inputs and preconditions

The context of the deployment architecture (typically a subsystem) must be specified in terms of its external interfaces and its responsibilities (requirements). Additionally, the contributions of the engineering facets to the design of the context (i.e. the requirements allocated to the facets) must be understood.

Outputs and postconditions

When the recipe is completed, the interfaces among the disciplines are adequately specified to enable the design of the involved facets.

How to do it

Figure 4.37 shows the workflow for the recipe. The flow is pretty simple but some of the actions identified in the workflow may require significant thought and effort:

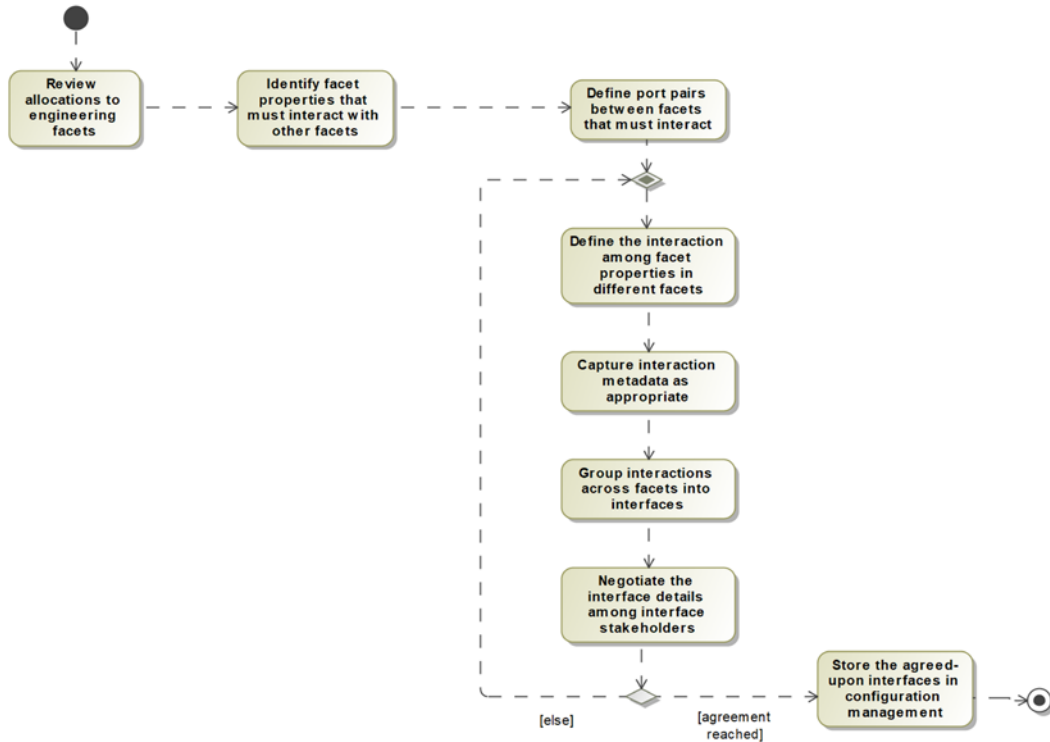


Figure 4.37: Define interdisciplinary interfaces

Review allocations to engineering facets

The previous recipe allocated requirements, values, and functions to the different engineering disciplines. Having a clear understanding of the contributions that must be provided by the engineering facets is a crucial step to creating good interdisciplinary interfaces.

Identify facet properties that must interact with other facets

Interfaces are all about defining the interactions between collaborating elements. In this case, we want to focus on the coordination of elements in different engineering facets. The previous recipe decomposed subsystem features into features within different engineering disciplines. The task here is to relate the properties in collaborating facets.

For example, an electronic sensor may provide a value to the software for manipulation and use, so the device driver (in the software facet) must interact with the actual sensor (in the electronics facet). This interaction will result in a service or flow connection of some kind.

Define port pairs between facets that must interact

Once we have identified that properties in two facets must interact, we define a port pair between the facets to convey the values or service invocations that will be added to the interface.

Define the interaction among facet properties in different facets

The exact nature of the interaction between the facets must be captured in the interface. For example, the software sensor device driver may write to a hardware register to signal the sensor to read a value, wait at least 1 ms, read the sensed value from another hardware register, and then scale the value to what the software needs to manipulate. This interaction and use must be clear in the interface specification.

Capture interaction metadata as appropriate

Metadata is “data about data.” In this case, we use metadata to capture the aspects of an interface that may not be obvious or easy to model in other ways. For example, the previous paragraph provided a simple example of the interaction of a software device driver with an electronic sensor. Relevant metadata would be the memory map address of the hardware registers and the bit-mapping of the values in the registers. For example, in the *sensor control register*, writing a 1 to bit 0 might activate the sensor, while writing a 0 bit to the same register has no effect; the value from bits 1–5 might provide a read-only error code should the electronics fail. The fact that the software must wait for 1 ms after activating the sensor before reading the value must be captured in the interface as metadata if the software and electronics are to work together properly.

Group interactions across facets into interfaces

All the interactions between the facets must be captured in interfaces. These interfaces must contain relevant information about the interfaces, including how they are accessed by the respective facets. This is most obvious with respect to electronic-software interfaces. The two most common technical means for interfacing between software and electronics are memory-mapped registers and hardware-generated software interrupts.

Negotiate the interface details among interface stakeholders

Keep in mind in this step is that the engineering disciplines must be free to develop their facets using their hard-won engineering skills without being overly encumbered by expectations from other engineering disciplines.

In other words, the systems or software engineers shouldn't dictate the electronics design and the systems or electronics engineers shouldn't dictate the design of the software. Nevertheless, the interfaces do specify the overlap between the disciplines. All the stakeholders should agree on the specifications of the interfaces. This includes the systems engineers – who have a stake in overall system optimization – and the involved engineering disciplines.

Store the agreed-upon interfaces in configuration management

There should always be a “known target” for the interfaces between facets even if that interface changes in the future. Yes, changing interfaces downstream will entail some amount of rework, but in practice, it is much less rework than if agreements aren't there in the first place. The worst outcomes occur when the disciplines continue without a shared understanding of those interfaces. Storing the interfaces under configuration management establishes a baseline and a source of truth from which the engineering disciplines can draw. Should future work uncover some inadequacy of the interfaces, the interfaces can be renegotiated and the configuration-managed baseline updated.

Example

This example will continue from the last recipe with the definition of the **Power Train** subsystem interfaces. The example in the previous section detailed the allocated requirements and subsystem features to the disciplines of mechanical, electronic, and software.

Review allocations to engineering facets

Read the example from the previous section to review the requirements, value, and function allocation.

Identify facet properties that must interact with other facets

In an effort to avoid over-specification of the facet designs, let's limit our concerns to the subsystem features that decomposed into elements mapped into different facets. In the previous example, we added dependencies from the decomposed facet-specific feature to the subsystem block features. *Figure 4.38* shows the mapping of facet features to subsystem block features:

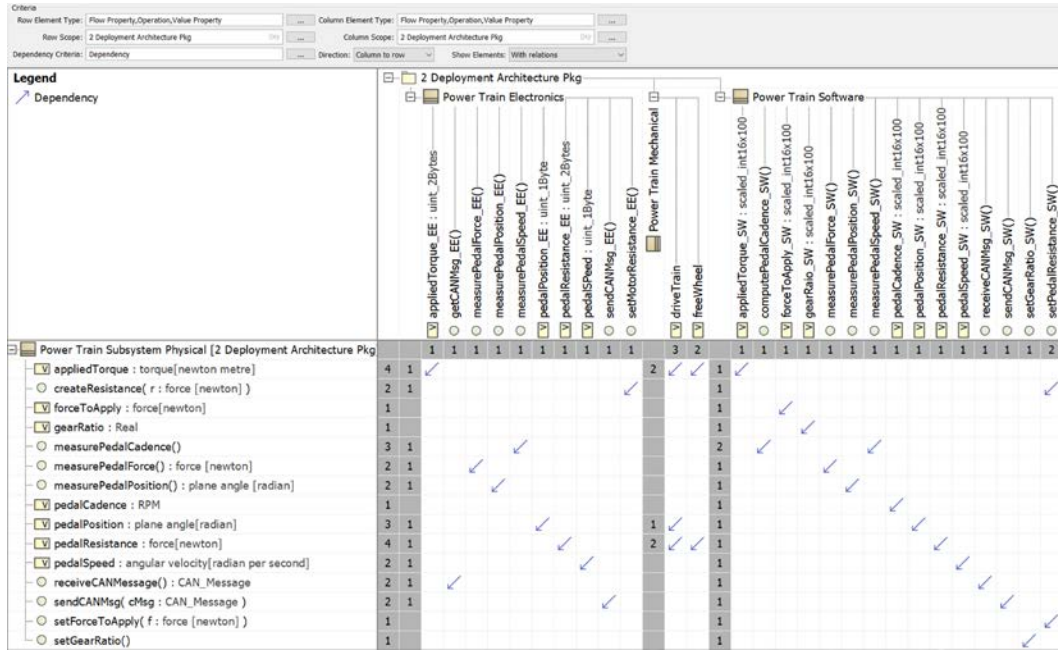


Figure 4.38: Subsystem features decomposed to facets

Define port pairs between facets that must interact

We previously defined the port and added empty interfaces to define them. See *Figure 4.28*.

Define the interaction among facet properties in different facets

This is the interesting part of the recipe: defining exactly how the facets will interact. Because we are using SysML proxy ports, we will capture these as interface blocks as shown in *Figure 4.39*:

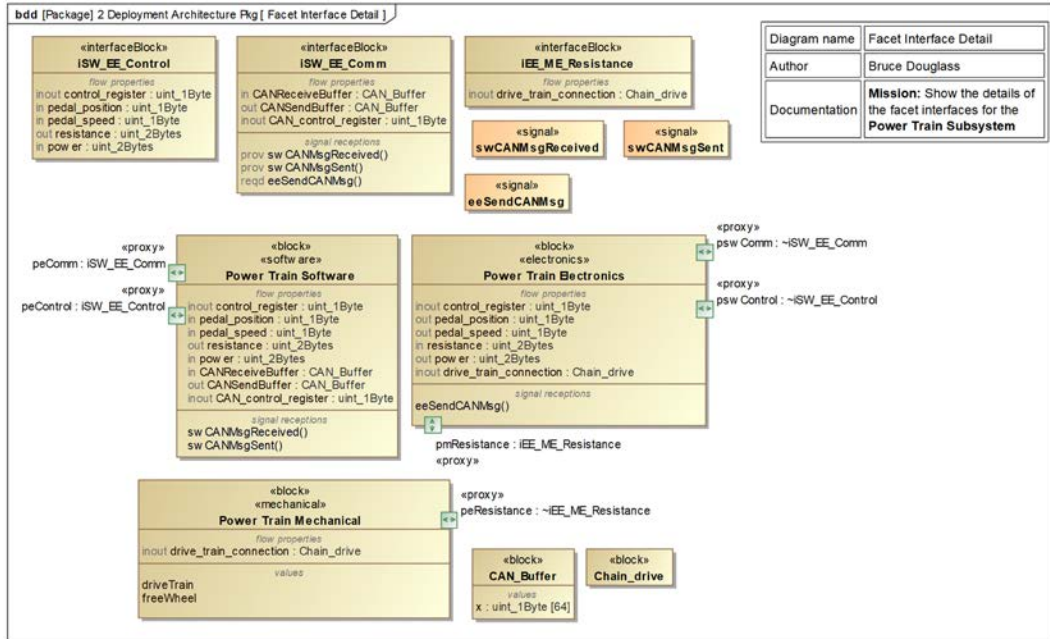


Figure 4.39: Deployment interfaces

The software-electronic interfaces are mostly composed of hardware registers, represented as flow properties. Both the **iSW_EE_Control** and **iSW_EE_Comm** interface blocks have control and data registers. The software triggers hardware actions or reads status information with these control registers. The data registers provide raw data from the software to the hardware or from the hardware to the software. The **iSW_EE_Comm** interface block also has two 64-byte blocks, one for sending a CAN bus message and the other for receiving one. The event receptions indicate interrupts generated by the hardware under different conditions. Also, the electronics-mechanical interface block **iEE_ME_Resistance** contains a drive train connection value property defined by the special value type **chain_drive**.

Let's now consider how the software-electronic interfaces will work, starting with the **iSW_EE_Control** interface block. In actual practice, this would be an outcome of a discussion between software, electronic, and systems engineers.

The control register in this case will have control or error indication responsibility, each of which requires a single bit:

- Bit 0 will be a software write-only bit that informs the hardware to start monitoring the pedal and providing resistance. When a 1 is written to this bit, the hardware will start applying the force as specified in the resistance hardware register and also start measuring pedal position, pedal speed, and power. The electronics will write those values to the corresponding hardware registers. When a 0 is written to this bit, the electronics stop providing resistance to the pedals and stop measuring pedal and power data. The hardware must assure the software that the updates to any register are all completed within a single CPU cycle to prevent race conditions.
- Bit 1 of the control register is read-only by the software and is used to indicate an error condition in the delivery of power via the motor: a 0 value indicates no error while a 1 indicates an error has occurred.
- Bit 2 of the control register is read-only by the software and indicates an error in the pedal position or pedal speed measurement (0 is no error, 1 indicates an error).
- Bit 3 is an indicator of an error reading power delivered by the rider (0 is no error, 1 indicates an error).
- Bits 4–7 are unused.

The other registers indicate either measured values from the electronics (read-only by the software) or commanded resistance from the software (write-only by the software). The electronics will have their own scale of these values, using either 8 bits (such as for the pedal speed and position) or 16 bits (for resistance and power). The direction of the flow properties is from the software perspective (following our previous convention that the first field in the interface block name indicates the unconjugated side of the interface). However, the range of values represented is not scaled by how the user would interpret the values, but rather by the hardware capability. The software is expected to scale the values to units meaningful elsewhere.

The electronics measures pedal position as an 8-bit value from 0 (right crank vertical) to 255 (almost vertical). Pedal speed is an 8-bit value ranging from 0 (no movement) to 255 (maximum measured speed of the hardware). Resistance is written to the register by the software as a 16-bit value from 0 (no resistance) up to 65,535 (maximum resistance). Likewise, power is a hardware-measured value from 0 (no power) to 65,535 (maximal measured power). The exact scaling factors will be specified by the hardware at a later date.

The **iSW_EE_Comm** interface is a bit more interesting. The core flow properties are the **CAN Receive Buffer** and **CAN Send Buffer**. Each of these is a 64-byte wide, byte-addressable memory register that holds CAN bus messages. The expected interaction flow looks like this:

To send a message:

1. The software checks bit 0 of the CAN control register. If it's a 0, then data may be written to the CAN Send Buffer. If not, the software must wait.
2. The software then writes the CAN bus message it wants to send into the **CAN Send Buffer**.
3. The software then writes a 1 to bit 0 of the **CAN control register**.
4. The hardware sets the read value of this bit to 1.
5. The hardware bangs the bits out on the CAN bus.
6. When the hardware has sent the message, the hardware sets the software read value of bit 0 of the **control register** to 0.
7. The hardware generates an interrupt 2 to the software to indicate that the **CAN Send Buffer** is now available.
8. The software may install an interrupt service routine to interrupt 2 (**swCANMsgSend**) to send the next message if desired.

To receive a message:

1. The hardware begins receiving the CAN bus message, filling in the 64-byte wide **CAN Receive Buffer**.
2. When the message has been received and stored, the hardware generates interrupt 3 (**swCANMsgReceived**) and sets the read value of bit 1 of the control register to 1 to indicate that a message is available.
3. The software is expected to either poll bit 1 of the control register or install an interrupt service routine to read the message.
4. Once the software has read the message, it is expected to write a 0 value to bit 1. The hardware ignores this value, but it serves as a flag to the software.

Capture interaction metadata as appropriate

The proper way to interpret the bits of the flow properties and the ordering and timing of actions to properly interact over the interface constitute the metadata of interest. To this end, I created a set of stereotypes for this purpose and put them into a **shared common profile** (*Table 4.3*).

Most of these stereotypes have tags to represent metadata of interest about the element, but some stereotypes are simply used to identify a “special kind of thing.” There is some redundancy in the stereotype tags to allow a degree of flexibility in modeling the necessary information:

Stereotype	Applicable to	Tag	Description
bitmapped	Argument	bit_0	Interpretation and use, including read, write, or read/write
	Attribute/Value Property	bit_1	Interpretation and use, including read, write, or read/write
	Call Operation	bit_2	Interpretation and use, including read, write, or read/write
	Class/Block	bit_3	Interpretation and use, including read, write, or read/write
	Signal	bit_4	Interpretation and use, including read, write, or read/write
	Flow/Flow Property/Item Flow	bit_5	Interpretation and use, including read, write, or read/write
	Part	bit_6	Interpretation and use, including read, write, or read/write
	Instance	bit_7	Interpretation and use, including read, write, or read/write
	Operation	bit_8	Interpretation and use, including read, write, or read/write
	Reception	bit_9	Interpretation and use, including read, write, or read/write
		bit_10	Interpretation and use, including read, write, or read/write
		bit_11	Interpretation and use, including read, write, or read/write
		bit_12	Interpretation and use, including read, write, or read/write
		bit_13	Interpretation and use, including read, write, or read/write
		bit_14	Interpretation and use, including read, write, or read/write
		bit_15	Interpretation and use, including read, write, or read/write
		Number_Of_Bits	Which bits are valid
	Start_Address	Memory map starting address	
	T i m i n g _ Constraints	Timing and delays in use	
	Usage	Description of the use of the register overall	

bytemapped	Argument	Endianism	Big or little endian
	Attribute/Value Property	Format	How to interpret the collection bytes
		Number_Of_Bytes	How many bytes are included in the value
	Call Operation	Start_Address	Memory map starting address
		Starting_Byte_Number	Index into a larger array, if necessary
	Class/Block	T i m i n g _ Constraints	Timing and delays in use
	Signal	Units	If appropriate, units represented by the value
	Flow/Flow Property/Item Flow	Usage	Description of the use of the register overall
	Part		
	Instance		
Operation			
Reception			
interrupt-mapped	Call Operation	Byte_Width	Byte width of arguments (if any)
	Signal	Data_Address	Location of arguments (if any)
		Data_Field_Type	Format/interpretation of arguments (if any)
	Operation	I n t e r r u p t _ number	Interrupt vector #
	Reception	Usage	What the interrupt indicates
memory-mapped	Argument	Bitmap	Internal format of the memory-mapped item
	Attribute/Value Property	Numer_Of_Bytes	Size (in bytes) of the memory-mapped item
		Range_High	High end of range (if continuous)
		Range_low	Low end of range (if continuous)
	Call Operation	Start_Address	Location of the first element in the memory map
	Class/Block	T i m i n g _ Constraints	Timeliness constraints on use
	Signal	Usage	Description of the use of the element
	Flow/Flow Property/Item Flow		
	Part		
	Instance		
	Operation		
Reception			
Triggered Operation			

<none>digitalVoltage	Attribute/Value Property Class/Block Reception Operation Signal Part Instance	Usage	How to interpret different voltage levels
ee_hy_interface	Attribute/Value property Argument Call Operation Class/Block Part Instance Operation Reception Triggered Operation	<none>	Indicates this interface is between electronics and hydraulics
ee_me_interface	Attribute/Value property Argument Call Operation Class/Block Part Instance Operation Reception Triggered Operation	<none>	Indicates this interface is between electronics and mechanical parts

electrohydraulic	Attribute/Value Property Class/Block Part Instance Operation Reception		Indicates this element is composed of integrated electronics and hydraulics
electronics	Attribute/Value Property Class/Block Part Instance Operation Reception	<none>	Indicates this element is implemented in electronics
hydraulic	Attribute/Value Property Class/Block Part Instance Operation Reception	<none>	Indicates this element is implemented in hydraulics
mechanics	Attribute/Value Property Class/Block Part Instance Operation Reception	<none>	Indicates this element is implemented in mechanics
physicalRealization	Dependency	<none>	Relates a logical element (target) to its physical realization (source)

software	Attribute/Value Property Class/Block Part Instance Operation Reception	<none>	Indicates this element is implemented in software
staticMechanical	Association Link/Connector	<none>	Indicates this relation indicates a mechanical linkage that does not convey a flow (e.g. a static connection, such as bolting something together)
sw_ee_interface	Attribute/Value property Argument Call Operation Class/Block Part Instance Operation Reception Triggered Operation	<none>	
voltagemapped	Attribute/Value Property Flow/Flow Property/Item Flow	<none>	Indicates this element represents values via voltage levels

Table 4.3: Some useful handoff stereotypes

We will use these stereotypes to model relevant metadata for the interfaces in our model. The flow properties and event receptions in the interface blocks in *Figure 4.39* are elaborated and filled out in *Table 4.4*:

Interface	Feature	Feature Type	Tag	Value
iEE_ME_Resistance	drive_train_connection	FlowProperty	direction	Bidirectional

iSW_EE_Comm	CAN_control_register	FlowProperty	bit_0	SW read: 0 = data may be written to send buffer; 1 = buffer is in use SW write: 0 = no effect, 1 = signal for electronics to send message
iSW_EE_Comm	CAN_control_register	FlowProperty	bit_1	SW read 0 = no new message, 1 = new message in CAN Receive Buffer SW Write: 0 = value saved to display on next read (otherwise ignored by electronics)
iSW_EE_Comm	CAN_control_register	FlowProperty	direction	Bidirectional
iSW_EE_Comm	CAN_control_register	FlowProperty	Start_Address	A000: A0020
iSW_EE_Comm	CAN_control_register	FlowProperty	Usage	Controls/indicates the status of the CAN message flow
iSW_EE_Comm	CANReceiveBuffer	FlowProperty	direction	In
iSW_EE_Comm	CANReceiveBuffer	FlowProperty	Format	Defined by the CAN Bus Standard
iSW_EE_Comm	CANReceiveBuffer	FlowProperty	Start_Address	A000: 0100
iSW_EE_Comm	CANReceiveBuffer	FlowProperty	Starting_Byte_Number	0
iSW_EE_Comm	CANReceiveBuffer	FlowProperty	Usage	Electronics write the values to the buffer and notify the SW when the complete message is stored.
iSW_EE_Comm	CANSendBuffer	FlowProperty	direction	Out
iSW_EE_Comm	CANSendBuffer	FlowProperty	Format	Defined by the CAN Bus standard
iSW_EE_Comm	CANSendBuffer	FlowProperty	Start_Address	A000: 1200

iSW_EE_Comm	CANSendBuffer	FlowProperty	Usage	SW writes the CAN message to be sent and then notifies the electronics to send the message via the CAN Control Register. Electronics notify the software when the message is sent.
iSW_EE_Comm	swCANMsgReceived	Reception	Interrupt_number	3
iSW_EE_Comm	swCANMsgReceived	Reception	Usage	Electronics generate this interrupt to indicate that a message has been received and is available in the CAN Receive Buffer.
iSW_EE_Comm	swCANMsgSent	Reception	Interrupt_number	2
iSW_EE_Comm	swCANMsgSent	Reception	Usage	Electronics generate interrupt #2 to indicate that the commanded CAN message has been sent.
iSW_EE_Control	control_register	FlowProperty	bit_0	SW WO (Write Only). 0 = do not provide resistance, 1 = provide resistance
iSW_EE_Control	control_register	FlowProperty	bit_1	SW RO (Read Only). 0 = no error in resistance, 1 = error
iSW_EE_Control	control_register	FlowProperty	bit_2	SW RO (Read Only). 0 = no pedal position error, 1 = pedal position error
iSW_EE_Control	control_register	FlowProperty	bit_3	SW RO (Read Only). 0 = no rider power measurement error, 1 = rider power measurement error
iSW_EE_Control	control_register	FlowProperty	direction	Bidirectional

iSW_EE_Control	control_register	FlowProperty	Number_Of_Bits	4
iSW_EE_Control	control_register	FlowProperty	Start_Address	0 x A000 0001
iSW_EE_Control	control_register	FlowProperty	T i m i n g _ Constraints	none
iSW_EE_Control	control_register	FlowProperty	Usage	Bit 0 enables/disables power measurement and delivery. Other bits provide error indicators.
iSW_EE_Control	pedal_position	FlowProperty	direction	In
iSW_EE_Control	pedal_position	FlowProperty	Numer_Of_Bytes	1
iSW_EE_Control	pedal_position	FlowProperty	Start_Address	A000:0002
iSW_EE_Control	pedal_position	FlowProperty	Starting_Byte_Number	0
iSW_EE_Control	pedal_position	FlowProperty	Units	Each value corresponds to 1/256 of a circle.
iSW_EE_Control	pedal_position	FlowProperty	Usage	SW RO (Read Only). Read the value to get the pedal position.
iSW_EE_Control	pedal_speed	FlowProperty	direction	In
iSW_EE_Control	pedal_speed	FlowProperty	Numer_Of_Bytes	1
iSW_EE_Control	pedal_speed	FlowProperty	Start_Address	A000 0004
iSW_EE_Control	pedal_speed	FlowProperty	Units	Each value corresponds to 1/1,608 radians/min (maximum of 256 revs/min).
iSW_EE_Control	pedal_speed	FlowProperty	Usage	SW RO (Read Only). Read this value to get the current velocity of the pedals.
iSW_EE_Control	power	FlowProperty	direction	In
iSW_EE_Control	power	FlowProperty	Units	0 = no resistance, 0 x FFFF = max power (0.046W per step)

iSW_EE_Control	power	FlowProperty	Usage	SW RO (Read Only). Read the value of the power applied to the pedal by the rider.
iSW_EE_Control	resistance	FlowProperty	direction	Out
iSW_EE_Control	resistance	FlowProperty	Format	16-bit value
iSW_EE_Control	resistance	FlowProperty	Numer_Of_Bytes	2
iSW_EE_Control	resistance	FlowProperty	Start_Address	A000:0006
iSW_EE_Control	resistance	FlowProperty	Starting_Byte_Number	0
iSW_EE_Control	resistance	FlowProperty	Units	0 = no resistance, 0 x FFFF = max resistance (18.3N per step)
iSW_EE_Control	resistance	FlowProperty	Usage	SW Write Only (WO). Specify the resistance to be applied to the rider at the pedal.

Table 4.4: Power train deployment interface metadata

Group interactions across facets into interfaces

We incrementally added the interface block features directly into the interface blocks already, so this step is already done!

Negotiate the interface details among interface stakeholders

Now that we have defined the interface blocks and their features (flow properties and event receptions) and characterized them with metadata, the stakeholders can meet to review and negotiate the interface details. The stakeholders, in this case, include representatives from systems, software, electronics, and mechanical engineering. The interface will define a kind of contract that all parties agree to honor.

Store the agreed-upon interfaces in configuration management

The defined interfaces can now be baselined in configuration management. All parties agree to uphold those interfaces. Should downstream work demonstrate inadequacies or identify problems with the interface, the relevant stakeholders can meet again, renegotiate the interface definition, and get back to engineering.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>



5

Demonstration of Meeting Needs: Verification and Validation

Recipes in this chapter

- Model simulation
- Model-based testing
- Computable constraint modeling
- Traceability
- Effective reviews and walkthroughs
- Managing model work items
- Test-driven modeling

George Box famously said that “all models are wrong, but some are useful,” (Box, G. E. P. (1976), “Science and statistics,” *Journal of the American Statistical Association*, 71 (356): 791–799). Although he was speaking of statistical models, the same can be said for MBSE models. In the systems world, models are both approximations of the real systems and subsets of their properties.

First, all models are abstractions; they focus on details of relevance of the system but completely ignore all other aspects. Therefore, the models are *wrong* because they don’t include all aspects of the system, just the ones we find of interest.

Second, all models are abstractions; they represent information at different levels of detail. When we model fluid flow in a water treatment system, we don't model the interactions at the molecular level. Rather, we model fluid dynamics but not the interaction of electric fields of atoms bound together into molecules, let alone get into quantum physics. Therefore, the models are *wrong*, because they don't model the electrodynamics of the particles and the interaction of the quantum fields of the elementary particles.

Third, all models are abstractions; they represent information at a level of precision that we think is adequate for the need. If we model the movement of an aircraft rudder control surface, we may very well decide that position, represented with 3 significant digits and an accuracy of ± 0.5 degrees, meets our needs. Therefore, our models are wrong because we don't model infinite levels of precision.

You get the idea. Models are not exact replicas of reality. They are simplified characterizations that represent aspects in a (hopefully) useful conceptual framework with (also hopefully) enough precision to meet the need.

So, questions arise when considering a model:

1. Is the model *right*?
2. What does *right* mean?
3. How do we *know*?

We will try to provide some practical answers to these questions in this chapter. The answers will vary a bit depending on the kind of model being examined.

Let's address the second question first: what does *right* mean? If we agree with the premise that all good models have a well-defined scope and purpose, then *right* surely means that it addresses that scope with the necessary precision and correctness to meet its purpose. If we look at a stakeholder requirements model, then the requirements it contains would clearly, unambiguously, and completely state the needs of the stakeholders with respect to the system being specified. If we develop an architecture, then the architectural structures it defines meet the system requirements and the needs of the stakeholder (or will, once it is developed) in an "optimal" way. If we develop a set of interfaces, then we describe all the important and relevant interactions with all the applicable details of that system with its actors.

If we use that answer for question #2, then the answer to the first question pertains to the veracity of the statement that the model at hand meets its scope and purpose. Subsequently, the answer to the third question is that we generate evidence that the answer to question #1 is, in fact, the case.

Verification and validation

Most engineers will agree with the general statement that verification means *demonstration that a system meets its requirements* while validation means *demonstration that the system meets the need*. In my consulting work, I take this slightly further (Figure 5.1):

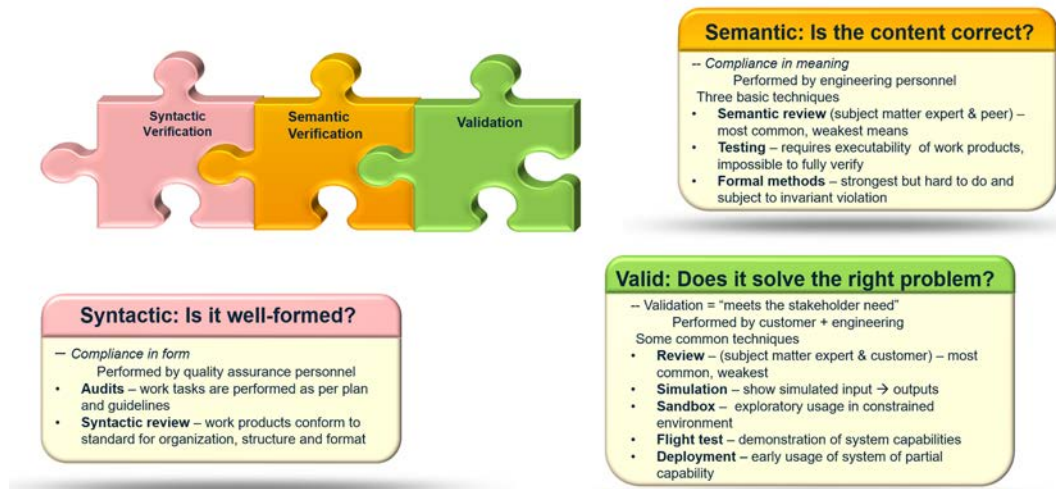


Figure 5.1: Verification and validation core concepts

In my mind, there are two kinds of verification: syntactic and semantic. **Syntactic verification** is also known as *compliance in form*, because it seeks to demonstrate that the model is well-formed, not necessarily that it makes sense or represents statements of truth. This means that the model complies with the modeling language syntactic rules and the project modeling standards guidelines. These guidelines typically define how the model should be organized, what information it should contain, the naming conventions that are used, and action language used for primitive actions, and so on. Every project should have a modeling guidelines standard in place and it is against such a standard that the syntactic well-formedness of a model may be verified. See <https://www.bruce-douglass.com/papers> for downloadable examples of MBSE and MDD guidelines. In addition, check out SAIC’s downloadable style guide and model validation tool at

Semantic verification is about demonstrating that the content makes sense; for this reason, it is also known as *compliance in meaning*. It is possible to have well-formed sentences that are either incorrect at best or nonsense at worst. There is even a Law of Douglass about this:



Any language rich enough to say something useful is also expressive enough to state utter nonsense that sounds, at first glance, reasonable.

Law of Douglass, #60

For the current state of the *Laws of Douglass*, see <https://www.bruce-douglass.com/geekosphere>.

The reason why this is important becomes clear when we look at the techniques of semantic verification (see *Figure 5.1*): semantic review, testing, and formal (mathematical) analysis.

If we have vague or imprecise models, we only have the first technique for ascertaining correctness: we can look at it. To be clear, a review is valuable and can provide insights for subtle issues or problems with a model; however, it is the weakest form of verification. The problem is largely one of vigilance. In my experience, humans are pretty good at finding issues for only a few hours at most before their ability to find problems degrades. You've probably had the experience of starting a review meeting with lots of energy but by the time the afternoon rolls around, you're thinking, "Kill me now." Reviews definitely add value in the verification process but they are insufficient by themselves. There's a reason why we don't say, "We can trust the million lines of code for the nuclear reactor because Joe looked at it really hard."

The second technique for performing verification is to test it. Testing basically means that we identify a set of test cases – a set of inputs with specific values, sequence, and timing and well-defined expected outputs or outcomes – and then apply them to the system of interest to see if it performs as expected. A *test objective* is what you want to get from running the test case, such as the demonstration of compliance to a specific set of requirements. In SysML, a test case is a stereotype of behavior; in Cameo, a test case can be an activity, interaction, or state machine. This approach is more expensive than just looking at the system but yields far superior results, in terms of demonstration of correctness and identifying differences between what you want and what you have. Testing fundamentally requires *executability* and this is crucial in the MBSE domain. If we want to apply testing to our model – *and we should* – then it means that our models must be specified using a language that is inherently precise, use a method that supports executability, and use a tool environment that supports execution. An issue with testing is that it isn't possible to fully test a system; there is always an essentially infinite set of combinations of inputs, values, sequences, and timings. You can get arbitrarily close to complete coverage at increasing levels of cost and effort, but you can never close the gap completely.

The third technique for verification is applying formal methods. This requires rendering the system in a mathematically precise language, such as Z or predicate logic, and then applying the rules of formal mathematics to demonstrate universals about the system. This is arguably the strongest form of verification but it suffers from two problems: 1. It is incredibly hard to do, in general. PhDs and lots of effort are required; and 2. The approach is sensitive to invariant violations. That means that the analysis must make some assumptions, such as “the power is always available” or “the user doesn’t do this stupid thing,” and if you violate those assumptions, you can’t trust the results. You can always incorporate any specific assumption in your analysis, but that just means there are other assumptions being made. Gödel was right (Kurt Gödel, 1931, “Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme,” I Monatshefte für Mathematik und Physik, v. 38 n. 1, pp. 173–198).

In my experience, the best approach for semantic verification is a combination of all three approaches – review, test, and analyze.

Verification is not the same as *validation*. Being *valid* means that the model reflects or meets the true needs of the stakeholders, even if those needs differ from what is stated in the system requirements. If meeting the system requirement is demonstrated through verification, meeting the stakeholder need is demonstrated through validation. For the most part, this is historically done with the creation of stakeholder requirements, and then reviewing the work products that describe the system. There are a couple of problems with that.

First, there is an “air gap” between complying with the stakeholder requirements and meeting the stakeholder need. Many a system has failed in its operational environment because, while it met the requirements, it didn’t meet the true needs. This is primarily why agile methods stress continuous customer involvement in the development process. If it can be discovered early that the requirements are not a true representation of the customer’s needs, then the development can be redirected and the requirements can be amended.

In traditional methods, customer involvement is limited to reviews of work products. In government programs, a **System Requirements Review (SRR)**, **Preliminary Design Review (PDR)**, and **Critical Design Review (CDR)** are common milestones to ensure the program is “on track.” These are almost always appraisals of many, many pages of description, with all of the problems inherent in the review previously mentioned. However, if you build executable models, then that execution – even if it is a simulated environment – can demonstrate how the system will operate in the stakeholders’ operational context and provide better information about system validity.

In this chapter, I will use the terms *computable* and *executable* when referring to well-formed models. When I say *computable model*, I mean a model that performs a computation, such as $F=ma$. In that equation, given two values, the system can compute the third, regardless of which two values are given. An *executable* model is a computable model in which the computation has a specific direction. Executable models are not necessarily computable. It is difficult to unscramble that egg or unexplode that bomb; these are inherently irreversible processes.

This chapter contains recipes that provide approaches to demonstrate the correctness of your models. We'll talk about simulation when it comes to creating models, developing computational models for analysis, performing reviews, and creating system test cases for model verification.

Model simulation

When we test an aircraft design, one way is to build the aircraft and see if it falls out of the sky. In this section, I'm not referring to testing the final resulting system. Rather, I mean verifying the model of the system before detailed design, implementation, and manufacturing take place. SysML has some expressive views for representing and capturing structure and behavior. Both the IBM Rhapsody and No Magic Cameo SysML tools have some powerful features to execute and debug models as well as visualize and control that execution.

The Rhapsody modeling tool performs simulation by generating software source code from the model in well-defined ways, and automatically compiling and executing that code. Rhapsody can instrument this code to interact with the Rhapsody tool so that the tool can visualize the model execution graphically and provide control of the execution. This means that SysML models can be simulated, executed, and verified in a relatively straightforward fashion, even by non-programmers. Cameo, on the other hand, is a true simulation tool. It does not generate code for the execution of the simulation. While in many senses, the execution capabilities of Cameo are somewhat weaker than those in Rhapsody, the fact that Cameo provides a true simulation improves other capabilities. For example, because it performs simulation, Cameo gives you complete control over the flow of time, so time-based analyses are simplified in Cameo over Rhapsody.

Simulation can be applied to any model that has behavioral aspects. I commonly construct executable requirements models, something discussed in *Chapter 2, System Specification*. I do this one use case at a time, so that each use case has its own executable model stored in its own package. Architectural models can be executed as well to demonstrate their compliance with the use case models or to verify architectural specifications. Any SysML model or model subset can be executed if it can be specified using well-define behavioral semantics and is rendered in a tool that supports execution.

Purpose

The purpose of a model simulation is to explore, understand or verify the behavior of a set of elements defined within the model. This simulation can be used to evaluate the correctness of a model, to explore what-if cases, or to understand how elements collaborate within a context.

Inputs and preconditions

The model is defined with a purpose and scope, and a need for simulation has been identified.

Outputs and postconditions

The results of the simulation are the computation results of running the simulation. These can be captured in values stored in the model, as captured diagrams (such as animated sequence diagrams), or as outputs and outcomes produced by the simulation.

How to do it

Figure 5.2 shows the workflow for performing general-purpose model-based simulation. Given the powerful modeling environment, it is generally straightforward to do, provided that you adhere to the tooling constraints:

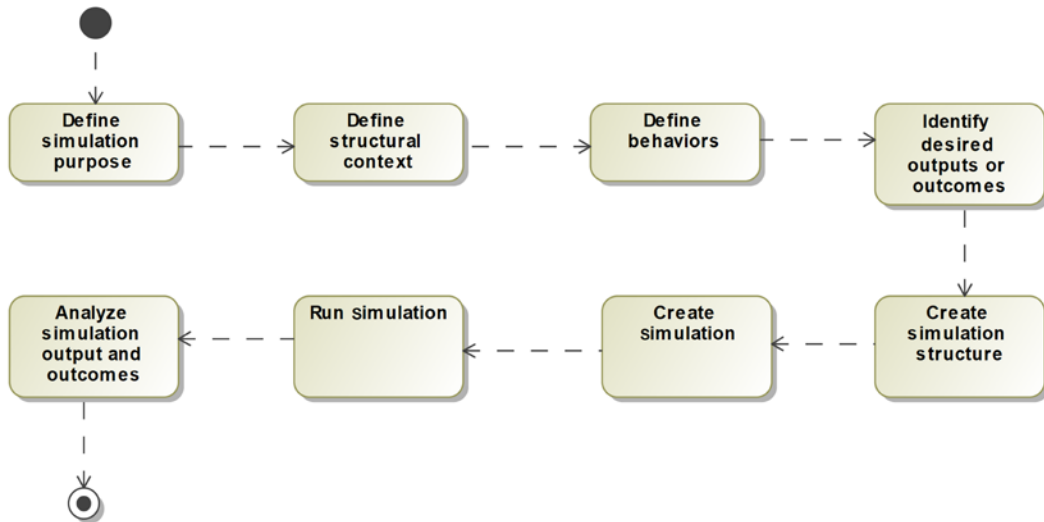


Figure 5.2: Model simulation workflow

It is common to have many different simulations within a systems model, to achieve different purposes and results. In Cameo, a simulation context is normally a block – it defines the set of elements (as parts) that collaborate in the simulation.

You may also create a **simulation configuration** with properties such as **listeners**, including sequence diagrams and **user interfaces**, as defined within the Cameo tool. It is common to have many such structures within a single MBSE model.

Define simulation purpose

It is imperative that you understand *why* you're performing model simulation in order to get a useful outcome. I like to phrase the problems as “what does success smell like” in order to ensure that you're going after the kind of results that will address your concerns. It matters because the models you create will vary depending on what you want to achieve.

Common purposes of model simulation include:

- Ensure the completeness and correctness of a set of requirements traced to by a use case.
- Demonstrate the adequacy of a set of interfaces.
- Explore behavioral options for some parts of a design model.
- Prove compliance of an architecture with a set of requirements.
- Reveal emergent behaviors and properties of a complex design.
- Verify a system specification.
- Validate a system architecture.
- Explore the consequences of various input values, timing, and sequence variation (“what-if” analyses).

Define structural context

The structural context for the model identifies the structural elements that will exhibit behavior – or used by structural elements that do – during the simulation. This includes blocks, use cases, and actors, along with their structural properties, such as value and flow properties, ports, and the relations among the structural elements.

Define behaviors

At the high-level, behavioral specification such as activity and state models provide the behavior of the system context as a whole or of the individual structural elements. At the detailed level, operations and activities specify primitive behavioral elements, often expressed in an action language, such as C, C++, Java, or Ada. In Cameo, I use the Groovy language, even though the default action language is, inexplicably, English. I can find and download a language specification for Groovy but not, again inexplicably, for Cameo's English usage.

It should be noted that the SysML views that contribute directly to the simulation and code generation are the activity and state diagrams, and any code snippets put into the action and operation definitions. Sequence diagrams, being only partially constructive, do not generally completely specify behavior; rather, they depict specific examples of it, although in Cameo, they can be used to drive a simulation.

Identify desired outputs or outcomes

In line with the identification of the simulation purpose, it is important to specify what kind of output or outcome is desired. This might be a set of output interactions demonstrating correctness, captured as a set of automatically generated sequences (so-called “animated sequence diagrams”). Or it might be a demonstration that an output computation is correct under different conditions. It might even be “see what happens when I do *this*”. Clarity in expectation yields satisfaction in outcomes.

Define simulation structure

The simulation structure refers to the elements used to create the simulation. In Cameo, this will be the set of elements that participate in the simulation, a simulation context (typically a block), and, optionally, a simulation configuration and user interface.

Create simulation

Creating the simulation is easily done by right-clicking the simulation context and selecting **Simulation > Run**, or selecting the simulation configuration in the simulation configuration selection list. In Cameo, it’s common to create a diagram that contains other diagrams and use this to visualize the behaviors of the simulation. While this can be almost any kind of diagram, I generally use a BDD or **free-form** diagram. Cameo has properties that can be specified for simulation initialization in the **Options > Environment > Simulation** menu. See a description of these options at <https://docs.nomagic.com/display/CST185/Automatic+initialization+of+context+and+runtime+objects>.

Run the simulation

Once created, the executable can be run as many times and under as many different circumstances as you desire. In Cameo, ill-formed models are detected either by running validation checks (with **Analyze > Validation > Validate**) or through execution. When a problem occurs during the execution, the element manifesting the problem will drop out of the simulation and an obscure error message is displayed.

Analyze simulation outputs and outcomes

The point of running a simulation is to create some output or outcome. Once this is done, it can be examined to see what conclusions can be drawn. Are the requirements complete? Were the interfaces adequate? Did the architecture perform as expected? What behavior emerged under the examined conditions? I especially like creating animated sequence diagrams from the execution for this purpose.

Example

The Pegasus model offers many opportunities to perform valuable simulations. In this case, we'll simulate one of the use cases from *Chapter 2, System Specification, Emulate Basic Gearing*. The use case was used as an example to illustrate the *Functional analysis with scenarios* recipe.

The mechanisms for defining simulations are tool-specific. The example here is simulated in Cameo and uses the Groovy action language. If you're using a different SysML tool, then the exact means for defining and executing your model will be different.

Define simulation purpose

The purpose of this simulation is to identify mistakes or gaps in the requirements represented by the use case.

Define structural context

The execution context for the simulation is shown in *Chapter 2, System Specification*, in *Figure 2.9* and *Figure 2.10*. The latter is an internal block diagram showing the connected instances of the actor blocks `prtTrainingApp:aEBG_TrainingApp` and `prtRider:aEBG_Rider` and the instance of the use case block `itsUc_EmulatedBasicGearing:Uc:EmulateBasicGearing`.

This model uses the canonical model organization structure identified in the *Organizing your models* recipe from *Chapter 1, Basics of Agile Systems Modeling* (see *Figure 1.35*). In line with this recipe, the detailed organization of the model is shown in *Figure 5.3*:

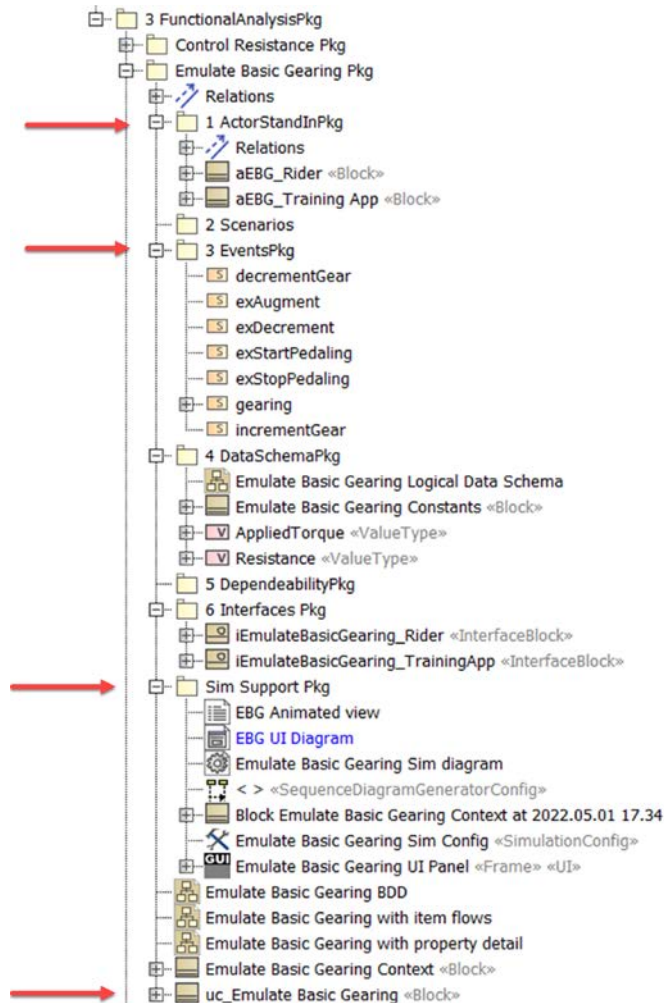


Figure 5.3: Organization of simulation elements

The **Functional Analysis Pkg** package contains a package for every use case being analyzed. This figure uses arrows to highlight elements and packages of importance for the simulation, including the actor blocks, the **Events Pkg** package, the **Sim Support Pkg** package, and the use case block. The **Sim Support Pkg** package contains the simulation configuration and UI developed for the simulation.

Define behaviors

The use case block state machine is shown in *Figure 2.11*. This state machine specifies the behavior of the system while executing that use case. The state machine represents a concise and executable restatement of the requirements and is not a specification of the design of the system. *Figures 2.12* and *Figure 2.13* show the state machines for the actor blocks. Elaborating the behavior of the actors for the purpose of supporting simulation is known as **instrumenting the actors**. Actor instrumentation is non-normative (since the actors are external to our system and we're not designing them) and is only done to facilitate simulation.

Identify desired outputs or outcomes

The desired outcome is to either identify missing requirements by applying different test cases to the execution, or to demonstrate that the requirements, as represented by the model, are adequate. To do this, we will generate animated sequence diagrams of different situations, examine the captured sequences, and validate with the customer that it meets the need. We will also want to monitor the output values to ensure that the **Rider** inputs are processed properly and that the correct information is sent to the **Training App**. Note that this is a low-fidelity simulation and doesn't emulate all the physics necessary to convert input **Rider** torque and cadence into output resistance and take into account **Rider** weight, wind resistance, current speed, and incline. While that would be an interesting simulation, that is not our purpose here.

We want to ensure that the **Rider** can downshift and upshift within the gearing limits, and this results in changing the internal gearing and the **Training App** being notified. Further, as the **Rider** applies input power, the output torque should change; a bigger gear with the same input power should result in larger output torque. These relations are built into the model; **applied torque** and **resistance** are modeled as **flow properties** while **upshift** and **downshift** are simulated with **increment gear** and **decrement gear** signal events. Further, the gearing (represented in gear inches, which is the distance the bike travels with one revolution of the pedal) should be limited between a minimum of 30 and a maximum of 140, while a gear shift changes the gearing by a constant 5 gear inches, up or down.

For this simulation, the conditions we would evaluate include:

- The gearing is properly increased as we upshift.
- The gearing no longer increases if we try to upshift past the maximum gearing.
- The gearing is properly decreased as we downshift.
- The gearing no longer decreases if we try to downshift past the minimum gearing.

- For a given level of input force, the output resistance is increased as we upshift (although the correctness of the output resistance is not a concern here).
- For a given level of input force, the output resistance is decreased as we downshift (although the correctness of the output resistance is not a concern here).

Define simulation view

Another nested package, named **Sim Support Pkg**, contains three things of interest with regard to the simulation structure. First, it contains a **simulation configuration**. In this case, the **execution target** is the context of the simulation, the block **Emulate Basic Gearing Context**. It also specifies that there is an **execution listener**, a **Sequence Diagram Generator Config**. Using this listener means that when the simulation is performed, an animated sequence diagram is created.

In addition, a **UI Frame** is added. This user interface allows the simulator to insert events and monitor and control values within the running simulation. The UI Frame is set to be the UI for the simulation configuration. See *Figure 5.4*:

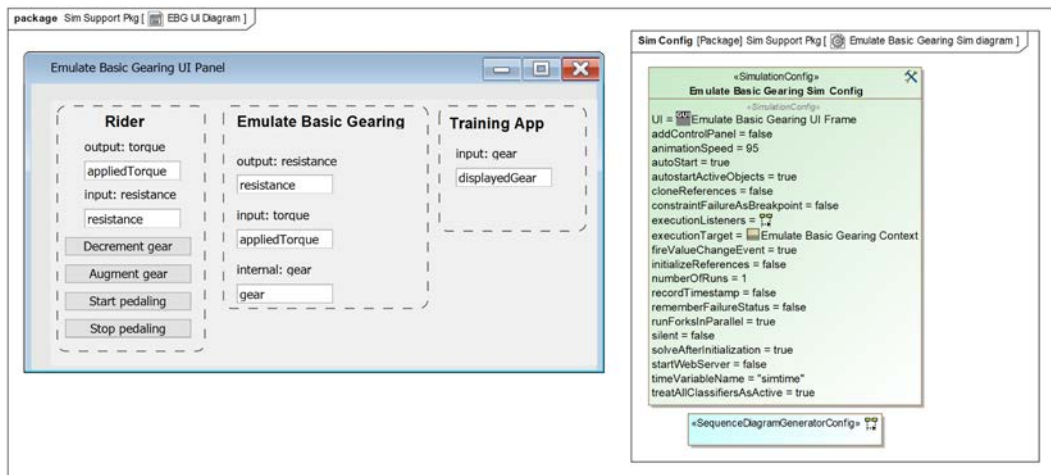


Figure 5.4: Simulation Settings

Although not a part of the simulation structure per se, the figure shows a simple panel used to visualize and input values and to enter events during the execution. Such UI Frames are not the only means Cameo provides to do this, but they are convenient.

Create simulation

Creating the simulation is a simple matter of selecting the simulation configuration to run from the simulation configuration list, and clicking the **Play** button. This list is directly beneath the **Analyze** menu at the top of the Cameo window.

Run the simulation

Simply run the simulation configuration. With a simulation configuration being run, the simulation starts immediately. If you simulate by right-clicking on a block, there is an extra step of clicking a green triangle on the Simulation Toolkit window that pops up. In this case, the start of the simulation looks like *Figure 5.5*:

The screenshot displays the Simulink environment for a gear simulation. The main workspace shows a block diagram with several interconnected blocks, including 'Emulate Basic Gearing', 'Basic Gearing', and 'Training App'. A dialog box titled 'Emulate Basic Gearing (UI Panel)' is open, showing the following parameters:

- output resistance: 0.0000
- input torque: 15.0000
- input resistance: 0.0000
- output torque: 0.0000
- nominal gear: 15.0000
- Decrement gear: [button]
- Increment gear: [button]
- Start pedaling: [button]
- Stop pedaling: [button]

The console window at the bottom shows the following log messages:

```

00:00:00.334: **** Block Emulate Basic Gearing.Contact execution is terminated. ****
00:00:00.000: Initial solving ...
00:00:00.000: Initial solving completed.
00:00:00.000: **** Block Emulate Basic Gearing.Contact is initialized. ****
00:00:00.000: **** Block Emulate Basic Gearing.Contact is started. ****
Training App: Gear = 30.0
    
```

Figure 5.5: Running the simulation

Analyze simulation outputs and outcomes

We capture the inputs and outputs using a number of simulation runs. In the first such run, we initialize the running simulation to output a resistance of 100 and the rider to provide a power value of 200. Then we augment the gearing and see that it increases. Further, the training app is updated with the gearing as it changes. We see that in the animated sequence diagram in *Figure 5.6*:

That's what we see in *Figure 5.7*; notice the coloring of the animated state machine shows the **[else]** path was taken and the gearing was not decremented. This is also seen in the **gear** value for the **Emulate Basic Gearing** panel (highlighted with arrows). The sequence diagram shows the **false** result of the guard following the **exDecrement** signal event:

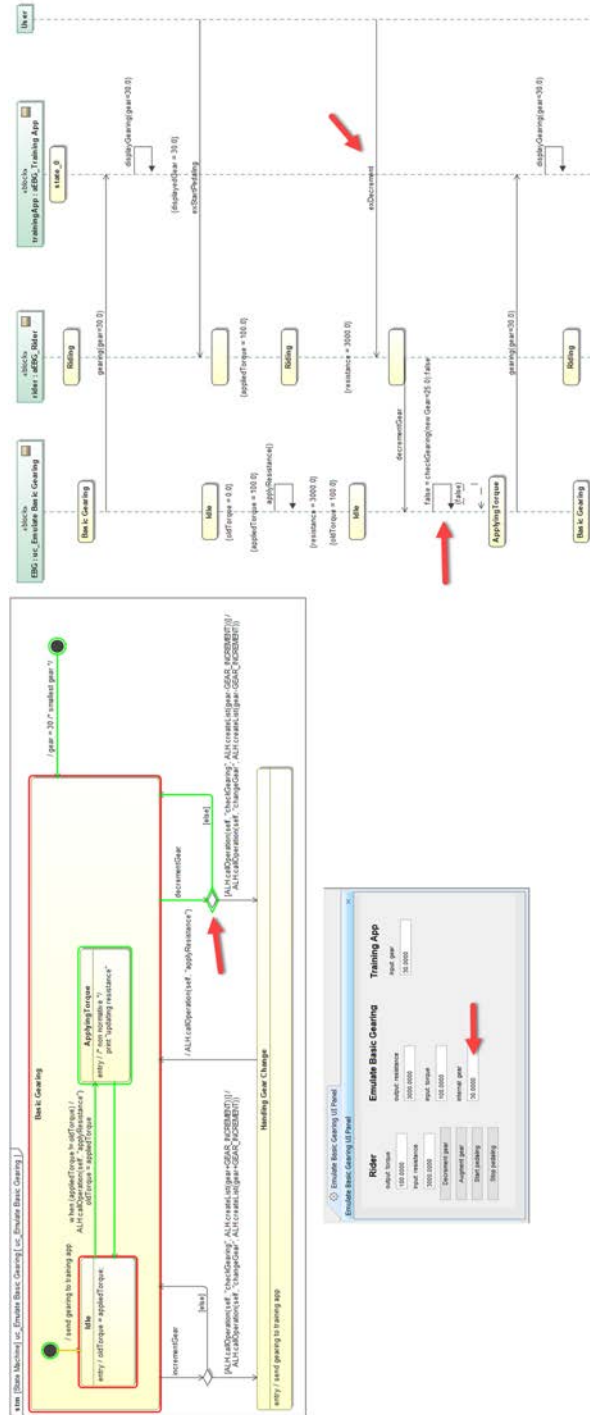


Figure 5.7: Simulation run 2

In the last run shown here, we set the gearing to 135, and then augment it twice to try to exceed the maximum gearing of 140 gear inches. We should see the resistance value go up with the first upshift, and then not change with the second. This is in fact what we see in *Figure 5.8*.

I've marked relevant points in the figure with red arrows. In the state diagram, you can see the **[else]** path was taken because the guard correctly identifies that the gear is at its maximum (140 gear inches). You can see the value in the UI at the bottom left of the figure. The sequence diagram shows that the guard returns **false** and the limiting value, 140, is sent to the **Training App**:

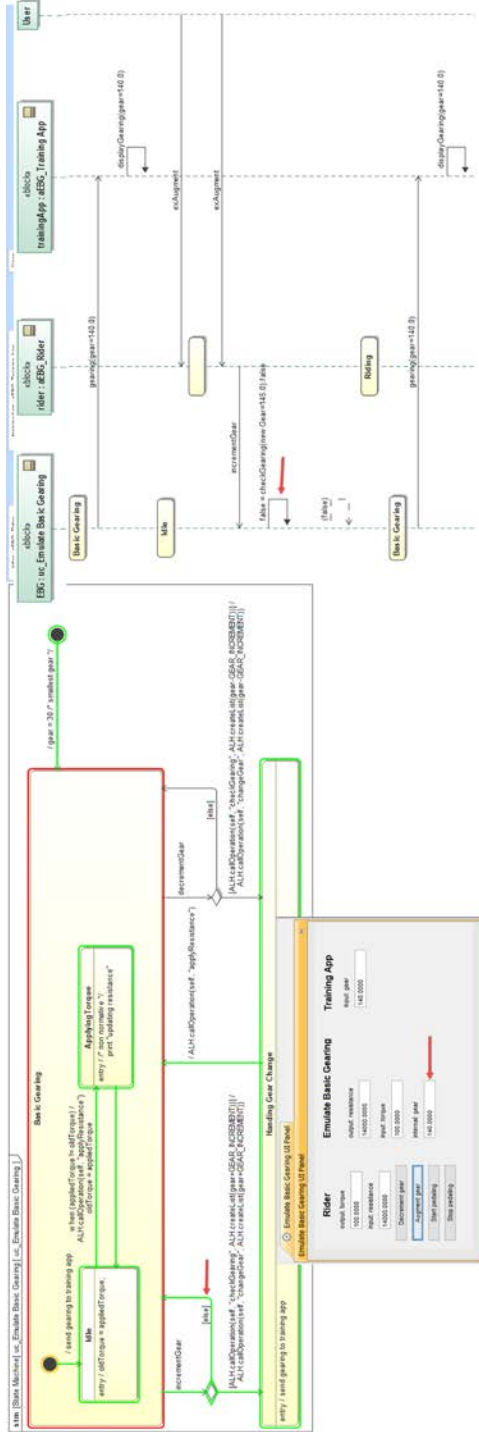


Figure 5.8: Simulation run 3

We have achieved the desired outcome of the simulation by demonstrating that the requirements model correctly shifts up and down, stays within gearing limits, properly updates the output resistance as the gearing is changed, and the **Training App** is properly notified of the current gearing when it changes.

Model-based testing

If you agree that modeling brings value to engineering, then **Model-Based Testing (MBT)** brings similar value to verification. To be clear, MBT doesn't limit itself to the testing of models. Rather, MBT is about using models to capture, manage, and apply test cases to a system, whether or not that system is model-based. In essence, MBT allows you to:

- Define a test architecture, including:
 - Test context
 - Test configuration
 - Test components
 - The **System Under Test (SUT)**
 - Arbiter
 - Scheduler
- Test cases, using:
 - Sequence diagrams (most common)
 - Activity diagrams
 - State machines
 - Code
- Test objectives

Bringing the power of the model to bear on the problems of developing test architectures, defining test cases, and then performing the testing is compelling. This is especially true when applying it in a tool like Cameo that provides such strong simulation and execution facilities.

While MBT can be informally applied, the **UML Testing Profile** specifies a standard approach. The profile defines a standard way to model test concepts in UML (and therefore in SysML). *Figure 5.9* shows the basic meta-architecture of the profile.

While it doesn't exactly represent what's in the profile, I believe it explains it a little better.

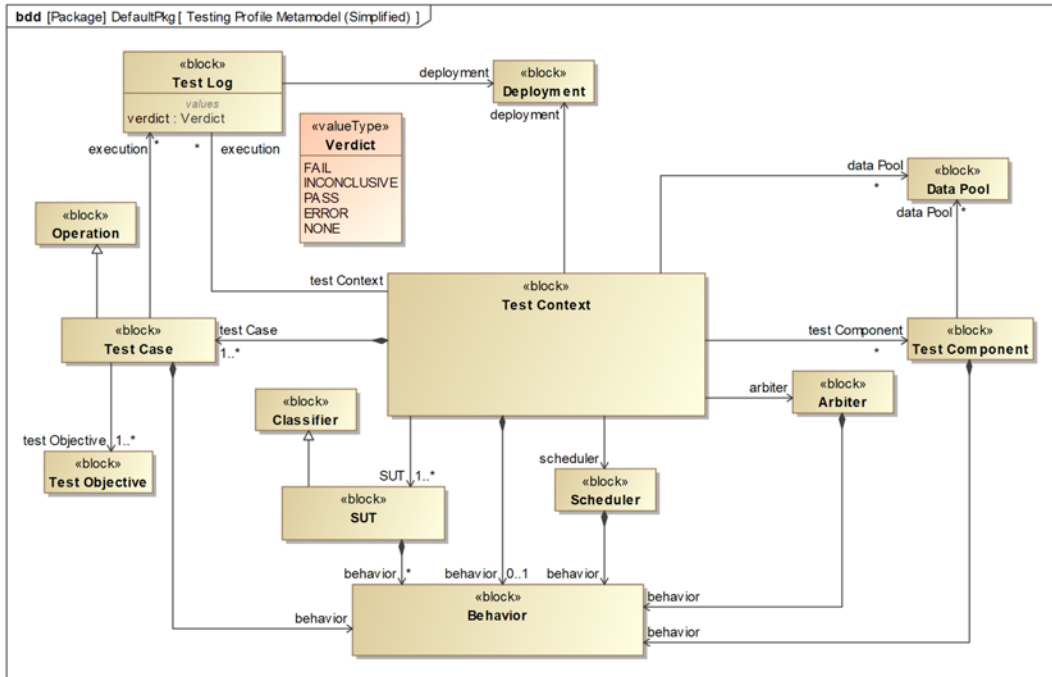


Figure 5.9: UML Testing Profile meta-architecture

I don't want to spend too much time on the details of the profile definition; interested people can check out the standard itself (<https://www.omg.org/spec/UTP2/2.1/PDF>) or books on the topic such as *Model-Driven Testing* (https://www.amazon.com/Model-Driven-Testing-Using-UML-Profile/dp/3540725628/ref=sr_1_1?dchild=1&keywords=uml+testing+profile&qid=1606409327&sr=8-1) by Baker, et al. However, I do want to highlight some aspects.

First, note that a **Test Case** is a stereotype of UML's metaclass **Behavior** such as an interaction, activity, or state machine. That means we can define a **Test Case** using the standard SysML behavioral representations.

The **Test Context** includes a set of **Test Cases**, and also a set (one or more) of **SUTs**. The SUT is a kind of **Classifier**, from the UML Metamodel; **Classifiers** can be **Blocks**, **Use Cases**, and **Actors** (among other things of course), which can be used in a context as singular design elements, composite elements such as subsystems, systems, or even an entire system context. SUTs themselves can, of course, have behavior. The **Test Log** records the executions of **Test Cases**, each of which includes a **Verdict**, of an enumerated type.

You can model these things directly in your model and build up your own **Test Context**, **Test Cases**, and so on. Cameo provides an implementation of the **Testing Profile**, which can be installed in the **Help > Resource/Plugin Manager** dialog, which we will use in the recipe.

Purpose

The purpose of model-based testing is to verify the semantic correctness of a system under test by applying a set of test cases. This purpose includes the definition of the test architecture, the specification of the test cases, the generation of the outcomes, and the analysis of the verdict – all in a model-based fashion.

Inputs and preconditions

The preconditions include both a set of requirements and a system that purports to meet those requirements. The system may be a model of the system – such as a requirement or design model – or it may be the final delivered system.

Outputs and postconditions

The primary output of the recipe is a test log of a set of test executions, complete with pass/fail verdicts of success.

How to do it

A model-based test flow (Figure 5.10) is pretty much the same as a normal test flow but the implementation steps are a bit different:

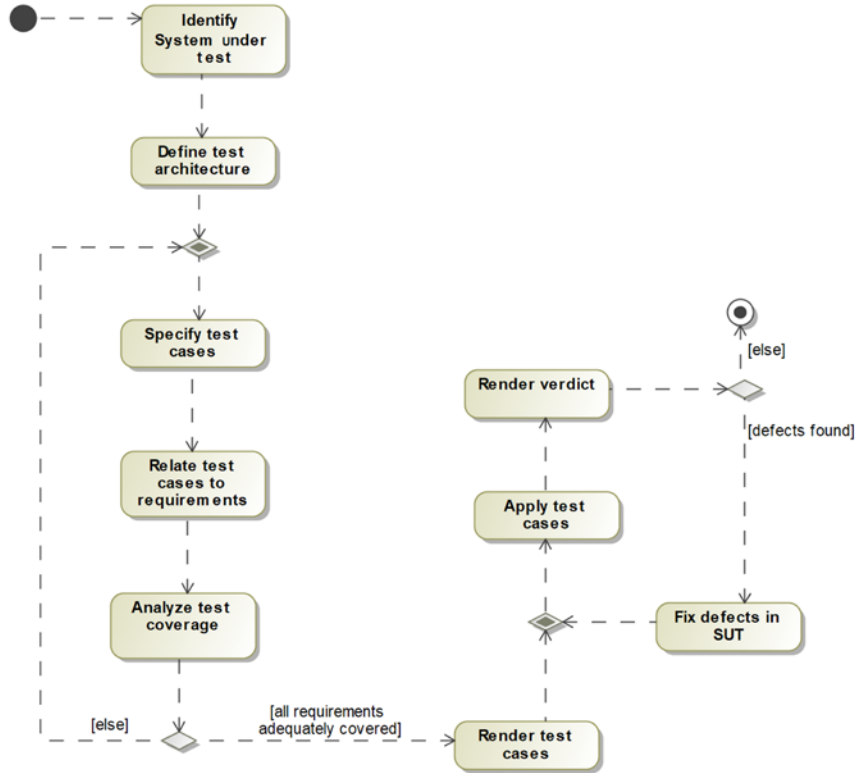


Figure 5.10: Model-based test workflow

Identify system under test

The SUT is the system or system subset that we want to verify. This can be a use case functional analysis model, an architecture, a design, or an actual manufactured system. If it is a design, the SUT can include a single design element or a coherent set of collaborating design elements.

Define test architecture

The test architecture includes, at a minimum, the instance of the SUT and test components substituting for elements in the actual environment (“test stubs”). The test architecture can also include a test context (a test manager), scheduler, and arbiter (to determine verdicts).

Specify test cases

Before we start rendering the test cases themselves, it makes sense to specify the test cases. The test cases include the events the test architecture will introduce and include the values of data they will carry, their sequence and timing, and the expected output or outcome.

Relate test cases to requirements

I am a huge fan of requirements-based testing. I believe that all tests should ultimately be trying to verify the system against one or more requirements. This step in the recipe identifies one or more test cases for each requirement of the SUT. In the end, there should be no requirements that are not verified by at least one test case, and there are no test cases for which there are no requirements.

Analyze test coverage

Test coverage is a deep topic that is well beyond the scope of this book. However, we will at least mention that coverage is important. Test coverage may be thought of in terms of the coverage of the specification (need) and coverage of the design (implementation). In terms of coverage of the specification we can think in terms of coverage of the inputs (sources and events types, values, value fidelity, sequences, and timings) and outputs (targets and event types, values, sequences, accuracy, and timings).

Design coverage is different. Mostly, it is evaluated in terms of path coverage. This is best developed in safety-critical software testing standards, such as DO-178 (https://my.rtca.org/NC_Product?id=a1B36000001IcmwEAC or see <https://en.wikipedia.org/wiki/DO-178C> for a discussion of the standard), that talk about three levels of coverage:

- Statement coverage: every statement should be executed in at least one test case.
- Decision coverage: statement coverage plus each decision point branch should be taken.
- Modified condition/decision coverage (MC/DC):
 - Each entry and exit point is invoked.
 - Each decision takes every possible outcome.
 - Each condition in a decision is evaluated in every possible outcome.
 - Each condition in a decision is independently evaluated as to how it affects the decision outcome.

The point of the step is to ensure the adequacy of the testing given the set of requirements and the structure of the design.

Render test cases

In model-based testing, we have a number of options for test specifications. The most common is to use sequence diagrams; since they are partially constructive, they are naturally suited to specify alternative interaction flows. The second most common approach is to specify multiple test paths using activity diagrams. The third is to specify the test cases with a state machine, a personal favorite. You can always write scripts (code) for the test cases, but as that isn't very model-based, we won't consider it here.

Apply test cases

The stage (test architecture) is set and the dialog (the test cases) has been written. Now comes the performance (test execution). In this step, the test cases are applied against the SUT in the context of the test architecture. The outcome of each test case is recorded in the test log along with a pass or fail verdict.

Render verdict

The overall verdict is a roll-up of the verdicts of the individual test cases. In general, an SUT is considered to pass the test only when it passes *all* of the test cases.

Fix defects in SUT

If some test cases fail, that means that the SUT didn't generate the expected output or outcome. This can be either because the SUT, the test case, or the test architecture is in error. Before moving on, the defect should be fixed and the tests rerun. In some cases, it may be permissible to continue if only non-critical tests fail. When I run agile projects, the basic rule is that critical defects must be addressed before the iteration can be accepted, but non-critical defects are put into the backlog for a future iteration and the iteration can progress.

To classify a defect as "critical" in this context, I generally use the *severity of the consequence* as the deciding factor. If the defect could have a significant negative outcome, then it is critical; such outcomes include system crashing or enable an incorrect decision or output that would have grave impact on system use. Causing a patient to die or leading a physician to a misdiagnosis of a medical issue would be a critical defect; having the color of an icon the wrong shade of red would not be.

Example

In this example, we'll look at the portion of the architectural design of the Pegasus system that accepts inputs from the **Rider** to change gears and make sure that the gears are properly changed within the **Main Computing Platform**.

To simplify the example, we won't look at the impact changing gears has on the delivered resistance.

Identify system under test

The system under design includes a few subsystems and some internal design elements. We are limiting ourselves to only designing the part of the system related to the rider's control of the gearing within those design elements. To show the element under test will take a few diagrams, so bear with me:

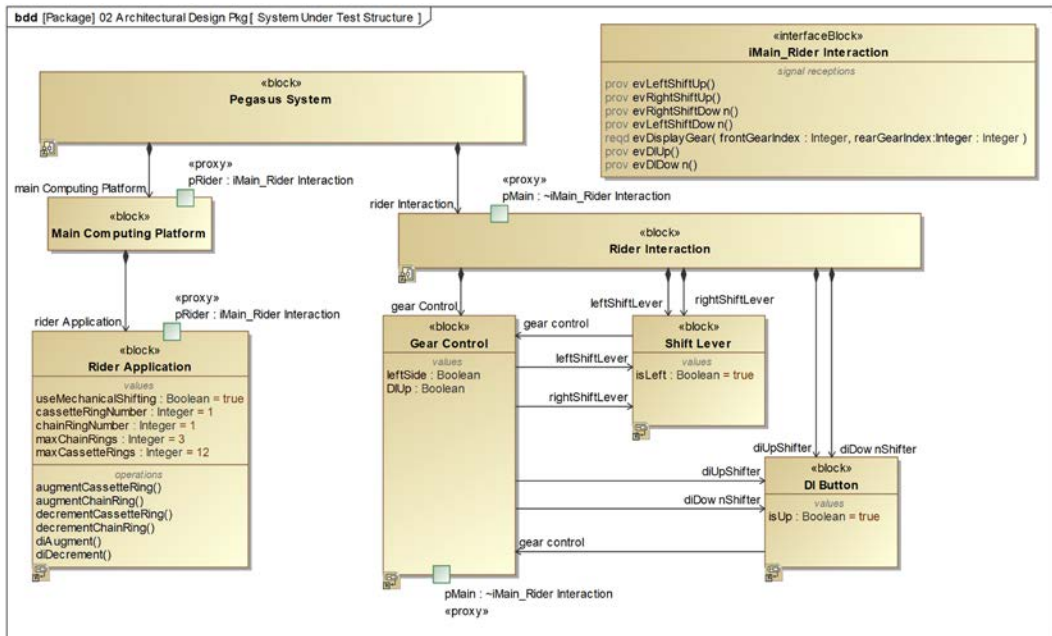


Figure 5.11: SUT structure

Figure 5.11 shows the blocks that together constitute the SUT: the **Rider Interaction** subsystem, which internally contains left and right **Shift Levers** and up and down **Digital Indexed (DI) shift buttons (DI Button)**, is one key part. The other key part is the **Main Computing Platform**, which contains the **Rider Application** that manages the gears. This diagram shows the blocks and their relevant properties.

This diagram doesn't depict well the runtime connected structure, so another diagram, *Figure 5.12*, shows how the instances connect:

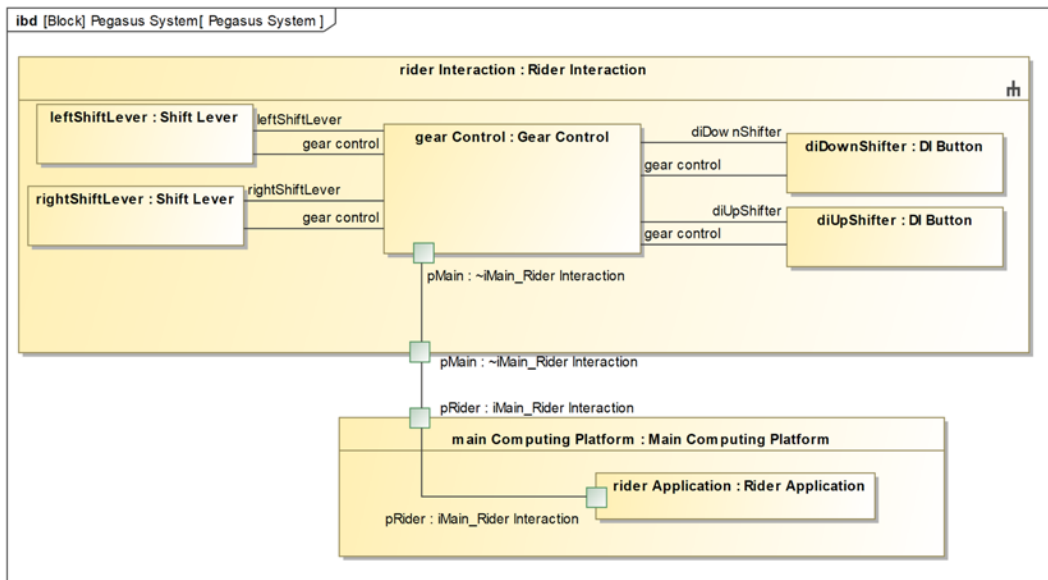


Figure 5.12: Connected parts in the SUT

All of these blocks have state machines that specify their behavior. The **DI Button** has the simplest state machine, as can be seen in *Figure 5.13*. As the **Rider** presses the **DI Button**, it sends an **evDIShift** event to the **Gearing Control** instance. The event carries a **Boolean** parameter that specifies it is the up (TRUE) or down (FALSE) button. The **Gearing Control** block initializes one of the **DI buttons** to be the up button and one to be the down button:

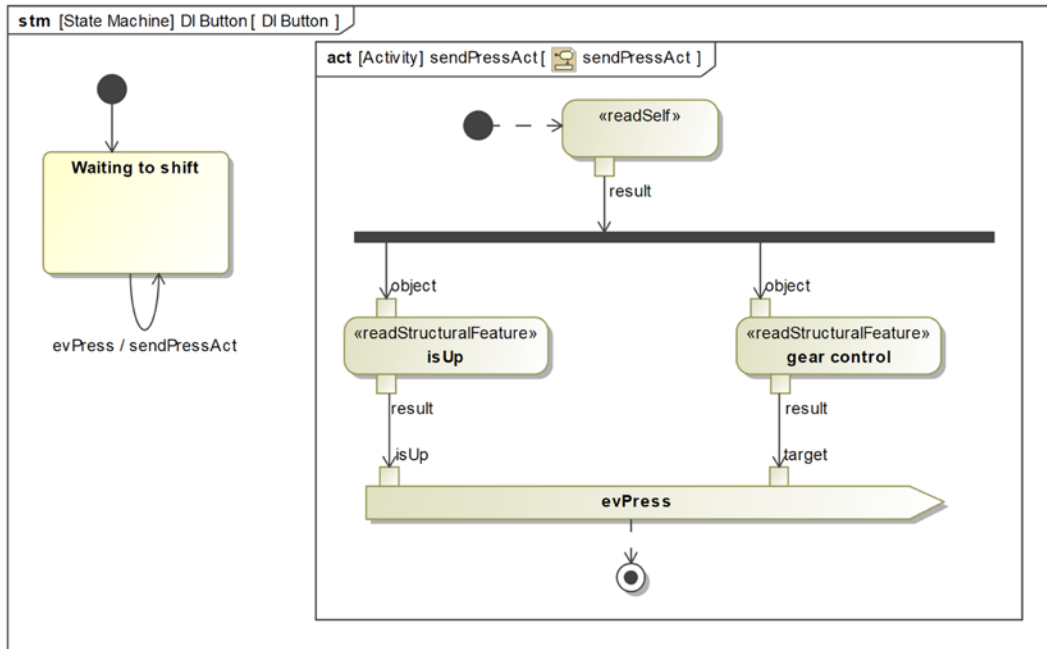


Figure 5.13: DI Button state machine

The **Shift Lever** state machine is only a little more complicated (Figure 5.14). The left **Shift Lever** controls the front chain ring and can either augment or decrement it; the right **Shift Lever** controls the cassette ring gear but works similarly. The state machine shows that either shift lever can send the **evShiftUp** or **evShiftDown** event. As with the **evDISHift** event, these events also carry a parameter that identifies which lever (left (**true**) or right (**false**)) is sending the event:

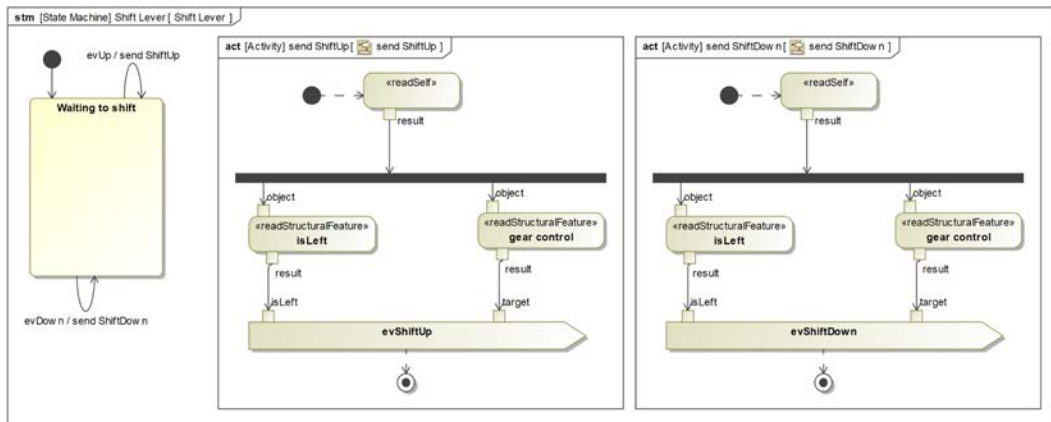


Figure 5.14: Shift Lever state machine

The **Gear Control** state machine is more complex, as you can see in *Figure 5.15*. It receives events from the left and right **Shift Levers** and the up and down **DI Buttons**. It must use the passed parameters to determine which events to send to the **Rider Application**. It does this by assigning a value property to be the passed parameter (either **leftSide** or **DIUp**, depending on which event), and then uses that value in a guard.

The lower AND-state handles the display of the current gearing confirmed by the **Rider Application**; this is sent by the **Rider Application** after it adjusts to the selected gears.

Also notice the initialization actions on the initial transition from the **Initializing** to the **Controlling Gears** state; this specifies the handedness of the **Shift Levers** and the **DI Buttons**:

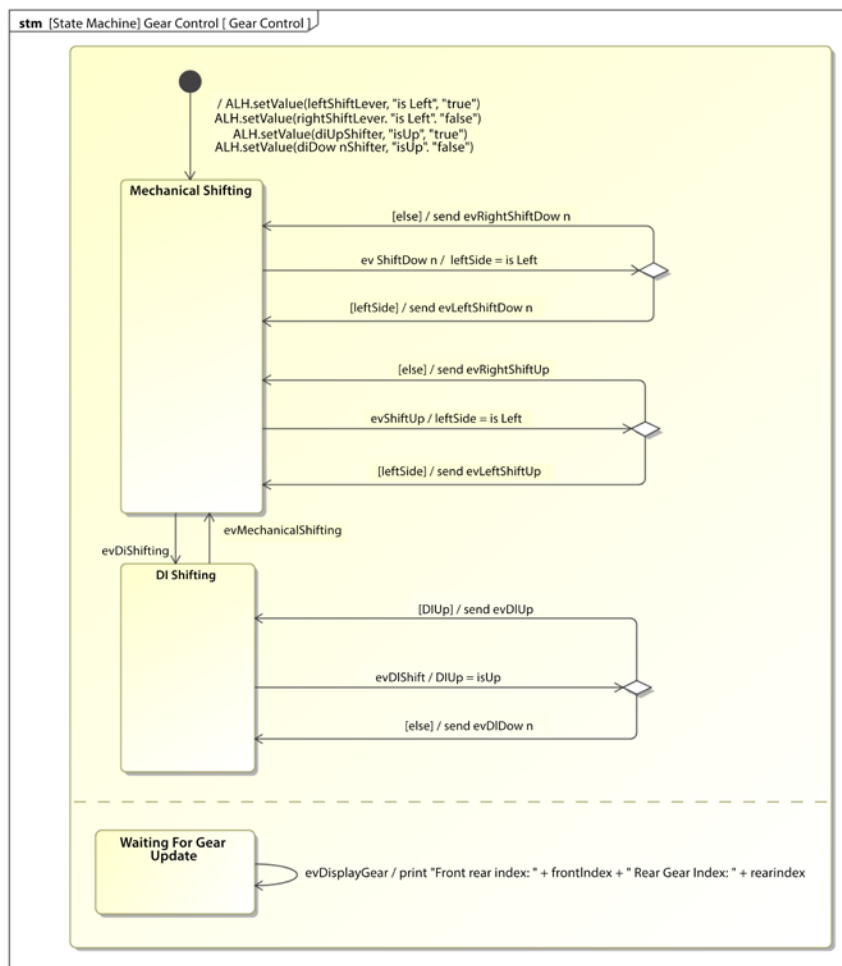


Figure 5.15: Gear Control state machine

The last state machine, shown in *Figure 5.16*, is for the **Rider Application**, which is owned by the **Main Computing Platform** subsystem. This state machine supports both mechanical shifting and DI shifting. It works by receiving the events from the **Gear Control** block and augments/decrements the gearing based on the values:

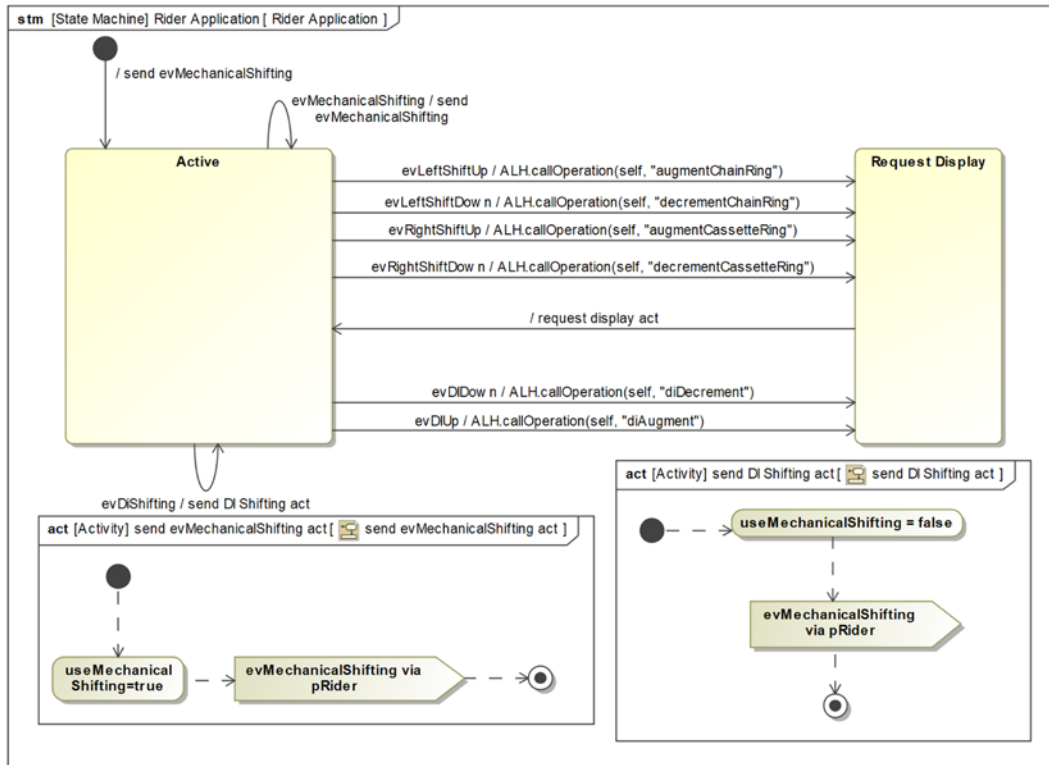


Figure 5.16: Rider Application state machine for gearing

The gear adjustments are made by operations that ensure that the gearing remains within the set limits (1 to **maxChainRings** and 1 to **maxCassetteRings**). *Figure 5.17* shows a table of the opaque behaviors (code snippets) for each of the operations referenced in *Figure 5.16*. Note that the operation itself is in the **Specification** column.

The methods are all written using the Groovy language:

#	Owner	Specification	Name	Body and Language
1	Rider Application	diDecrement()	method diDecrement	<pre>if (cassetteRingNumber > 1) cassetteRingNumber -- else if (chainRingNumber > 1) { chainRingNumber -- cassetteRingNumber = maxCassetteRings }</pre>
2	Rider Application	augmentCassetteRing()	augmentCassetteRing method	<pre>if (cassetteRingNumber < maxCassetteRings) cassetteRingNumber ++</pre>
3	Rider Application	augmentChainRing()	augmentChainRing method	<pre>if (chainRingNumber <= maxChainRings) chainRingNumber ++</pre>
4	Rider Application	decrementCassetteRing()	decrementCassetteRing method	<pre>if (cassetteRingNumber > 1) cassetteRingNumber --</pre>
5	Rider Application	decrementChainRing()	decrementChainRing method	<pre>if (chainRingNumber > 1) chainRingNumber --</pre>
6	Rider Application	diAugment()	diAugment method	<pre>if (cassetteRingNumber < maxCassetteRings) cassetteRingNumber ++ else if (chainRingNumber < maxChainRings) { chainRingNumber ++ cassetteRingNumber = 1 }</pre>

Figure 5.17: Functions for setting gearing

Finally, the **request display act** behavior invoked on the **Rider Application** state machine is shown in Figure 5.18:

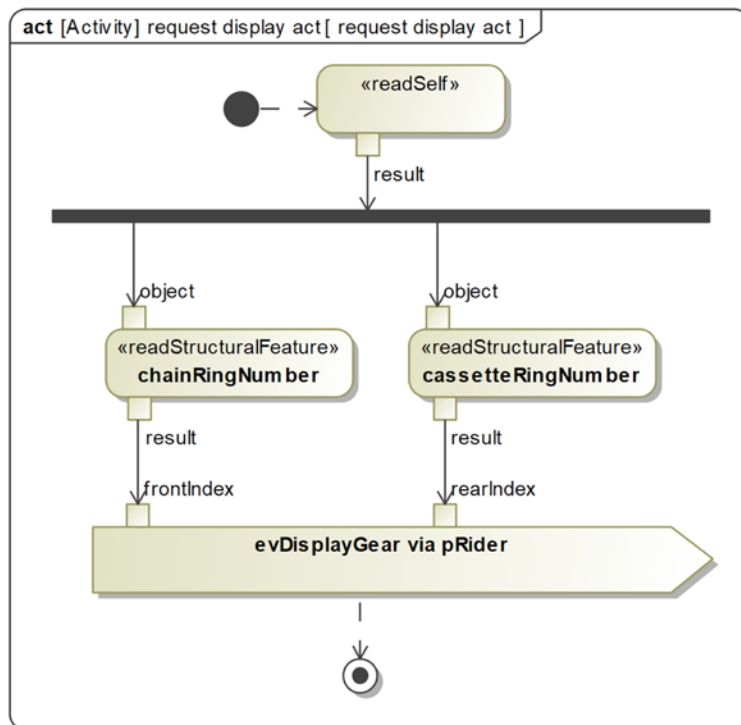


Figure 5.18: Request display act

So, there's our system under test. Is it right? How can you tell? Without testing, it would be easy for simple mistakes to creep in.

Define test architecture

The test architecture includes the development of the test fixtures or stubs we will use to perform the test. The test fixtures must provide the ability to repeatably set up the initial starting conditions for our tests, introduce the events with the proper values, sequences, and timings, and observe the results and outcomes.

We could create some blocks to do this in an automatic fashion. I personally use the stereotypes «testbuddy» and «testarchitecture» for such created elements.

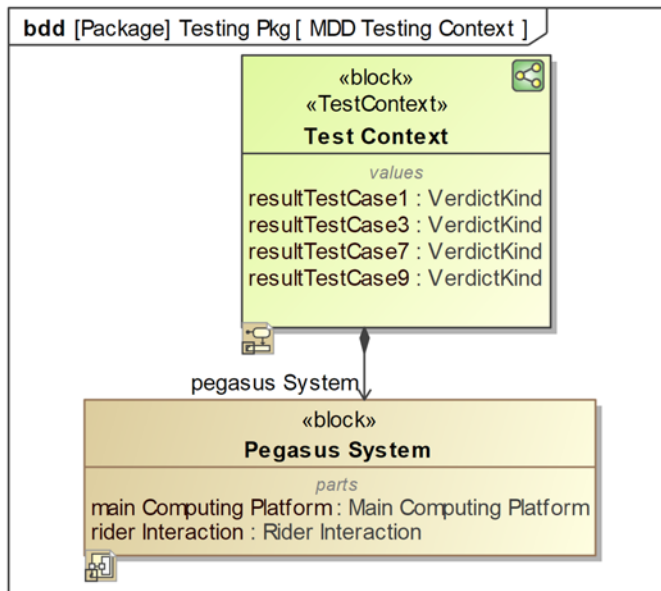


Figure 5.19: Pegasus test architecture

Figure 5.19 shows the **Test Context** block added with the SUT, **Pegasus System**, as a part. This means that the testing doesn't interfere at all with the behavior or structure of the SUT. It is, of course, important to not modify the elements of the SUT for the purpose of the test, as much as possible. As the testing maxim goes, "Test what you do; do what you test." In this case, no modifications of the SUT to facilitate testing are necessary. Sometimes you may want to add «friend» relations from the elements of the SUT to the test components to make it easier to perform the tests. Note that the **Test Context** block has value properties of type **VerdictKind** from the SysML profile or **Verdict** from the testing profile. These will hold the outcomes of the different test cases (in this case, test # 1, 3, 7, and 9). We will capture the outcomes of the test runs as instance specifications of type **Test Context** to record the outcomes.

There are many approaches to doing a model-based test, but in this example, we will use elements from the Cameo **Testing Profile** plugin.

Specify test cases

Let's think about the test cases we want (*Table 5.1*). This table describes the test cases we want to apply:

Test Case	Preconditions	Expected Outcomes	Description
Test case 1	Initial starting conditions. Mechanical shifting, chain ring 1, cassette ring 1	Chain ring selection moves up to desired chain ring	Augment the front chain ring from 1 to 2
Test case 2	Mechanical shifting, chain ring 1	Remain in chain ring 1	Decrement chain ring out of range low
Test case 3	Chain ring 3	Remains in chain ring 3	Augment chain ring out of range high
Test case 4	Initial starting conditions. Mechanical shifting, chain ring 1, cassette ring 1	Cassette ring selection moves up to desired cassette ring	Augment the front cassette ring from 1 to maxCassetteRings (12)
Test case 5	Mechanical shifting, cassette ring 1	Remains in cassette ring 1	Decrement cassette ring out of range low
Test case 6	Mechanical shifting, cassette ring 12	Remains in cassette ring 12	Augment cassette ring out of range high
Test case 7	Mechanical shifting	Switches to DI shifting	Move to DI shifting
Test case 8	DI shifting	Switches to mechanical shifting	Move to mechanical shifting
Test case 9	DI shifting, Chain ring 1, cassette ring 2	Augments to cassette ring 3, then 4, then 5	With DI shifting, press UP three times to see that shifting works
Test case 10	DI shifting in chain ring 2, cassette 1	Decrements to appropriate chain and cassette rings	Decrements to next lowest gear inches, augmenting chain ring, and going to appropriate cassette ring

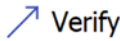
Table 5.1: Test case descriptions

To fully test with even the limited functionality we're looking at, more test cases are needed, but we'll leave those as exercises for you. This should certainly be enough to give the reader a flavor of model-based testing.

Relate test cases to requirements

To look at how the tests address the requirements, I created «verify» relations from each of the test cases to the set of requirements they purported to verify. I then created a matrix displaying those relations. Obviously, more test cases are needed, but *Figure 5.20* shows the starting point for test case elaboration:

Criteria
 Row Element Type: Test Case
 Column Element Type: Requirement
 Row Scope: Testing Pkg
 Column Scope: ReqsPkg
 Dependency Criteria: Verify
 Direction: Both
 Show Elements: With relations

Legend
 Verify

ReqsPkg

- 64 DIReq10
- 65 DIReq11
- 66 DIReq12
- 67 DIReq13
- DI Shifting Req
- 77 DIReq04
- EFaR Reqs Pkg
- 102 efarg19
- 103 efarg20
- 104 efarg21
- 105 efarg22

Test Context [Testing Pkg]	64 DIReq10	65 DIReq11	66 DIReq12	67 DIReq13	DI Shifting Req	77 DIReq04	EFaR Reqs Pkg	102 efarg19	103 efarg20	104 efarg21	105 efarg22
Test Context [Testing Pkg]	2	3	1	2		1		1	2	1	2
Test Case 1 act	5	↗					4	↗	↗	↗	↗
Test Case 3 act	2						2		↗		↗
Test Case 7 act	4	↗	↗	↗							
Test Case 9 act	4	↗	↗	↗	1	↗					

Figure 5.20: Test case verifies requirements table

Analyze test coverage

For our purposes here, we will examine coverage in terms of both the requirements and path coverage in the SysML behavioral models – that is, our test cases should execute every transition in each state machine and every control flow in every activity diagram.

As far as requirements coverage, there is a test case for the requirements in *Figure 5.20*, but many more requirements are not yet covered. We must also consider the degree completion of the coverage of requirements that are covered. The control of the chain ring in mechanical shifting mode is pretty complete: we have test cases to ensure that in the middle of the range, augmenting and decrementing work properly, that you can't augment beyond the limit, and that you can't decrement below the minimum. However, what about DI mode? There are a couple of test cases but we should consider cases where augmentation forces a change in the chain ring as well as the cassette ring, where gear decrement forces a reduction in the chain ring, and cases where they can be handled completely by changing the cassette ring only. In addition, we need cases to ensure that the calculations are right for different gearing configurations, and that we can't augment or decrement beyond available gearing.

To consider design coverage, you can “color” the transitions or control flows that are executed during the executions of tests. As more tests are completed, fewer of the transitions in the design element state behaviors should remain uncolored. One nice thing about the Cameo Simulation Toolkit is that it does color states, actions, transitions, and control flows that are executed, so it is possible to visually inspect the model after a test run.

Render test cases

We will implement these case cases with activity behavior in the **Test Context** block. The testing profile defines **Behavior** and **Operation** as the base metaclasses for the «test case» stereotype; I've applied that stereotype to those specific testing activities. One result of doing this is that each test case has an output activity parameter named **verdict**, which is of enumerated type **Verdict-Kind**. The potential values here are **pass**, **fail**, **inconclusive**, or **unknown**. The basic structure of this test suite is shown in *Figure 5.21*.

The **Test Context** behavior runs the four rendered tests in order; in each case, the initial starting conditions for the test are set, and then the test is run:

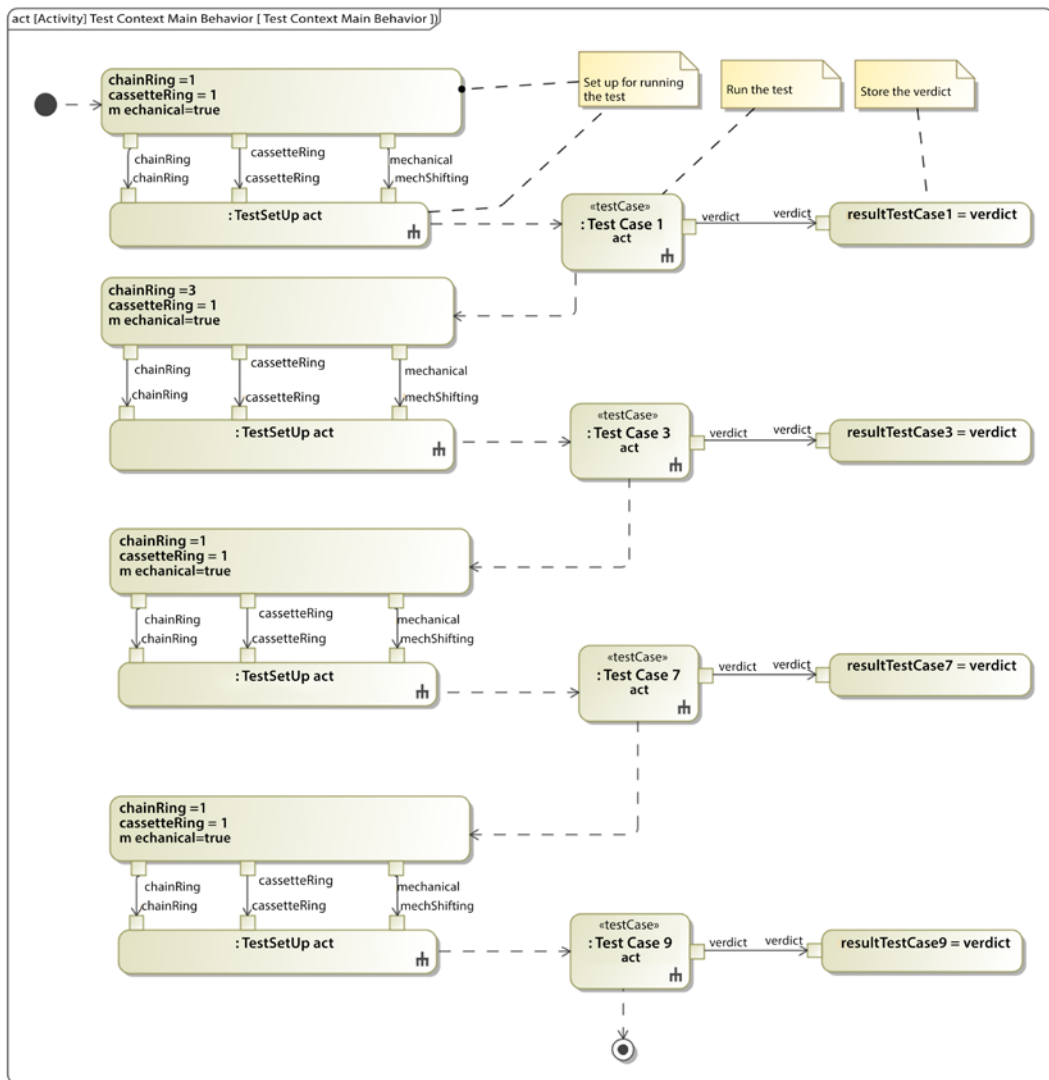


Figure 5.21: Test architecture main behavior

To assist in the execution of the test cases, I also created a simulation configuration. In the configuration, I set the run speed to 100% and instructed the system to put the final resulting instance in the **Testing Results** package. This allows me to create an instance table of test case runs:

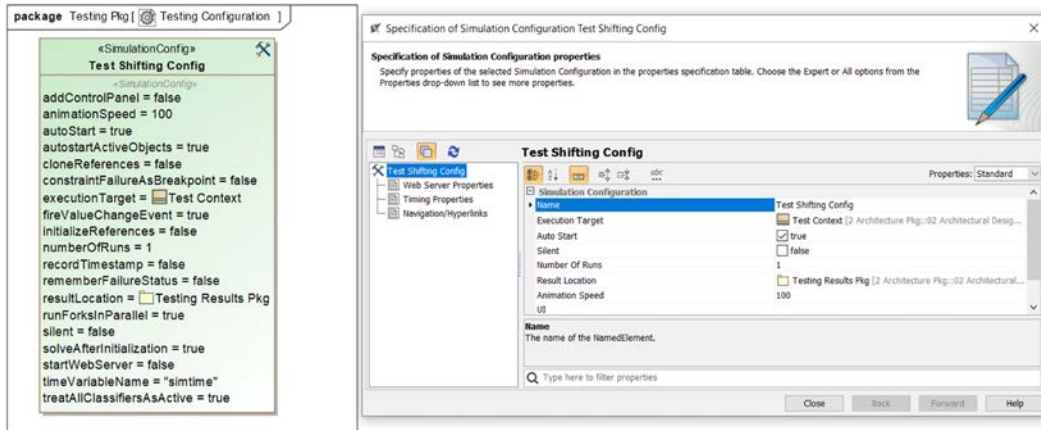


Figure 5.22: Test simulation configuration

Let's render just four of these test cases: 1, 3, 7, and 9.

Render test case 1: Augment the front chain ring from 1 to maxChainRings (3).

Adding the state behavior for this test case is pretty simple. The initialization just needs to:

- Set the shifting mode to mechanical shifting.
- Set the selected chain ring to the first (1).

The test execution behavior is pretty simple. The system just needs to send the left **Shift Lever** the **evUp** event and verify that both the set chain ring and the displayed chain ring are correct.

Figure 5.23 shows the details of the state **TC1_Execution** activity:

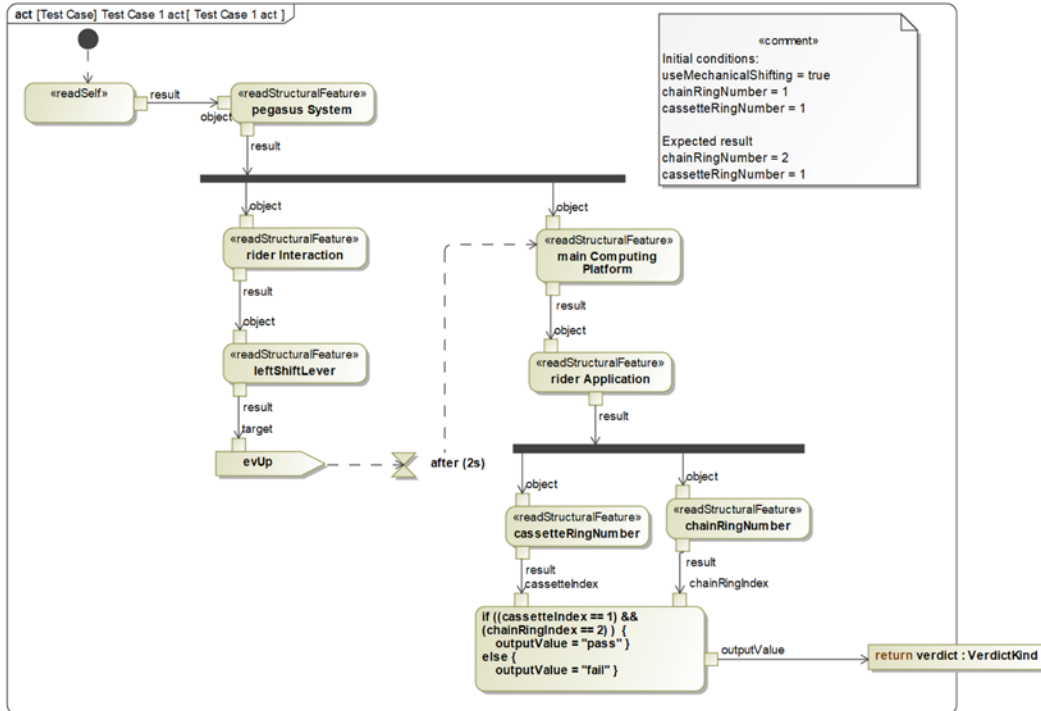


Figure 5.23: Test case 1 specification

In this case, test case 1 sends the `leftShiftLever` part the signal `evUp`. Following a short delay to allow the target to receive and consume the incoming signal, the values of the `cassetteRingNumber` and `chainRingNumber` are checked, and if found to be correct, the test case returns a verdict of `pass`; otherwise, it returns a verdict of `fail`.

Render test case 3: Augment chain ring out of range high

In this test case, the set up for the test for mechanical shifting and sets the selected chain ring to 3. It tries to augment the chain ring (which is already at its maximum); it should remain at the value 3.

The test case activity is shown in *Figure 5.24*:

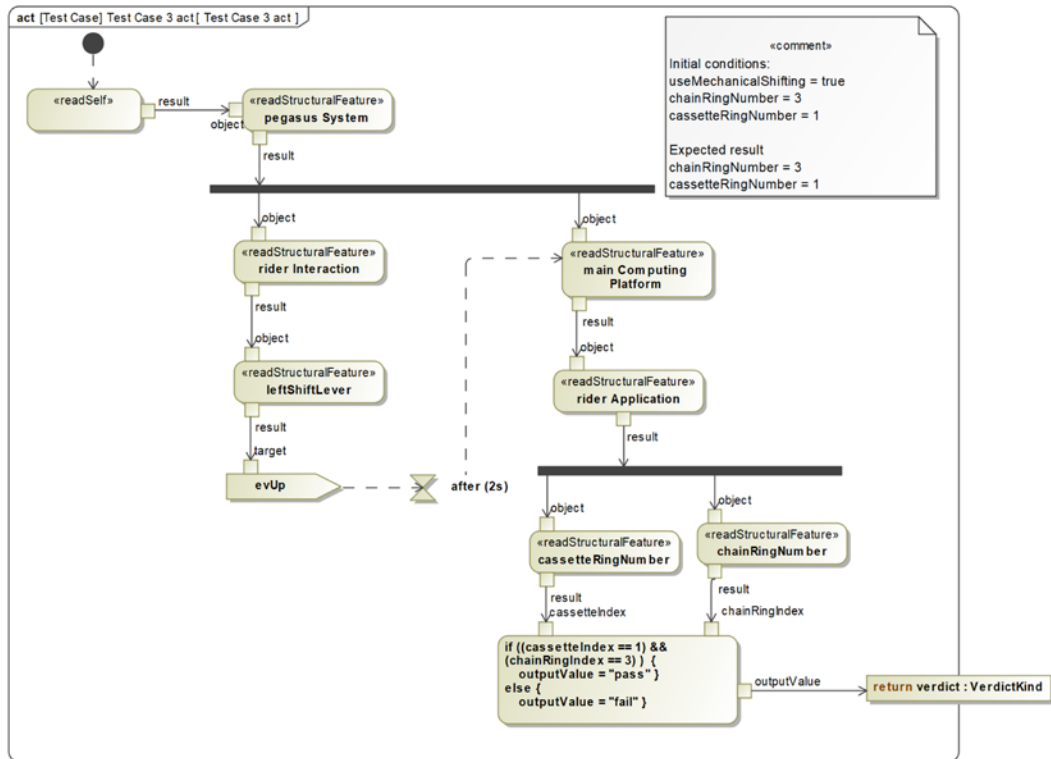


Figure 5.24: Test case 3 specification

Render test case 7: Move to DI shifting

This is an easy test case. All we have to do is to run the system and then check that the system is set to use DI shifting; this is best done by ensuring that the **Gear Control** state is currently **DI Shifting**. Cameo provides an **action language helper** named `ALH.inState` that returns a Boolean if the specified instance is in the specified state and `False` otherwise.

The activity for test case 7 is shown in *Figure 5.25*:

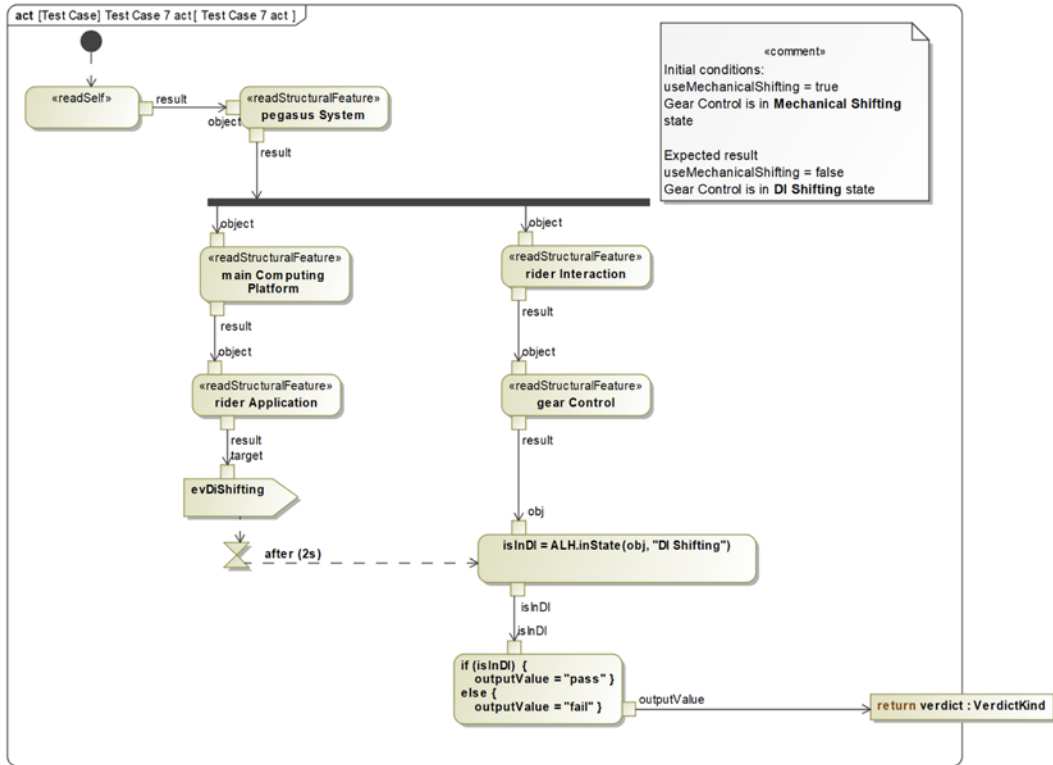


Figure 5.25: Test case 7 specification

Render test case 9: With DI shifting, press UP three times to see that shifting works

In the last test case, we'll enter DI mode and then shift UP twice to ensure that shifting works. The gearing is initialized to chain ring 1 and cassette ring 2 with DI shifting enabled. The test presses the DI Up Button three times, and we expect to see it augment the gearing to 1:2 and on to 1:3:

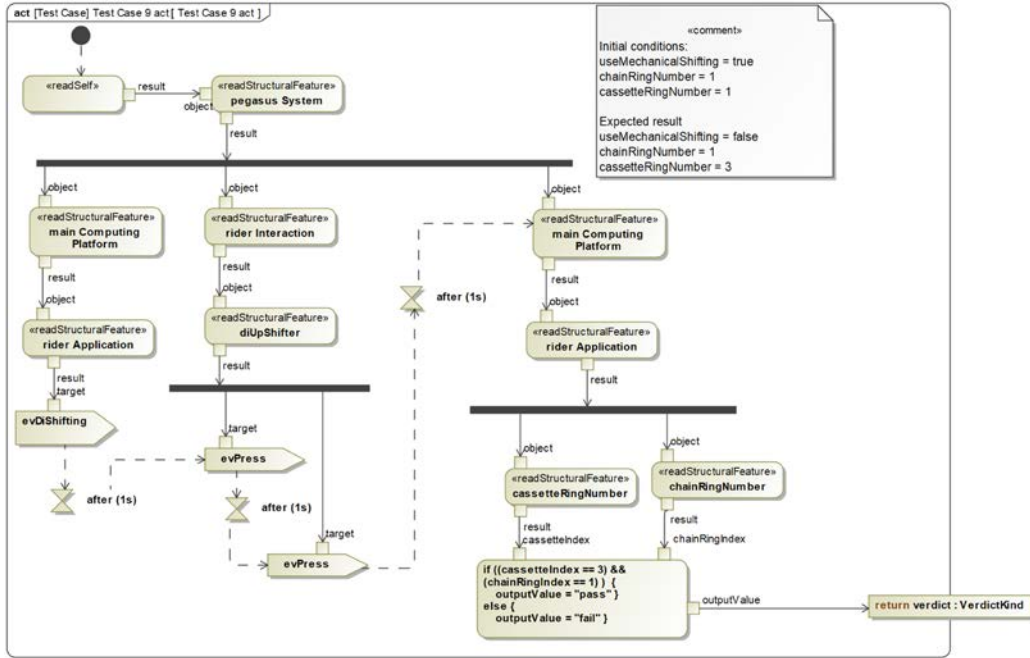


Figure 5.26: Test case 9 specification

Apply test cases

Now, let’s run the test cases and look at the outcomes. Since I think you learn more from test failures than successes, prepare to see some failures. In the **Test Results Pkg** package, I added an instance table showing the results generated by running the test case. In it, we can see that while test 1 passed, test cases 3, 7, and 9 failed:

Criteria				
Classifier: Test Context		Scope (optional): Testing Results Pkg		Filter: <input type="text" value="V*"/>
#	Name	<input type="checkbox"/> resultTestCase1 : VerdictKind	<input type="checkbox"/> resultTestCase3 : VerdictKind	<input type="checkbox"/> resultTestCase7 : VerdictKind
1	test Context	pass	fail	fail

Figure 5.27: Test results instance table

Render verdict

As not all tests succeeded, the test suite, as a whole, fails. This means that the design must be repaired and the tests rerun.

Fix defects in SUT

The defects are all fairly easy to fix. For test 3, the issue is in the implementation of the **Rider Application** operation **augmentChainRing**. The current implementation is:

```
if (chainRingNumber <= maxChainRings) chainRingNumber++;
```

but it should be:

```
if (chainRingNumber < maxChainRings) chainRingNumber++;
```

Tests 7 and 9 failed because the **Rider Application** activity **send DI Shifting act** sent the wrong event. In *Figure 5.16*, you can see that it sends the **evMechanicalShifting** event, but it should send the **evDIShifting** event. This is fixed in the following diagram:

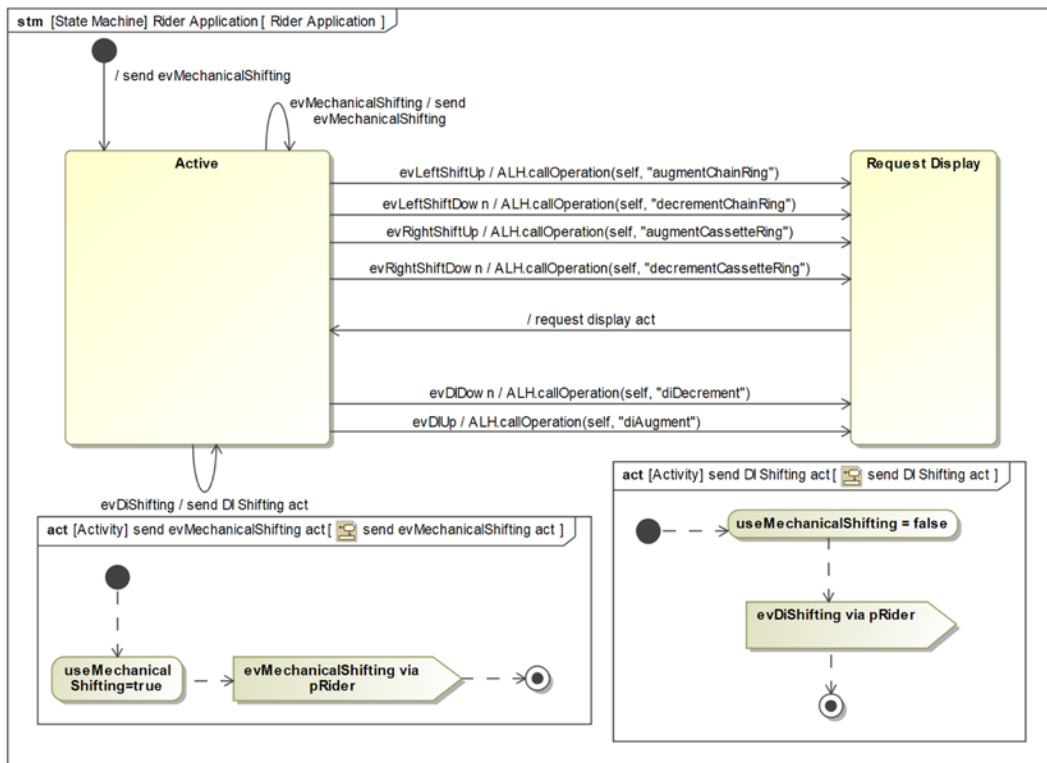


Figure 5.28: Updated Rider Application behavior

After making the fixes, we can rerun the test suite. We can see in *Figure 5.29* that all the tests pass in the second run:

#	Name	resultTestCase1 : VerdictKind	resultTestCase3 : VerdictKind	resultTestCase7 : VerdictKind	resultTestCase9 : VerdictKind
1	test Context	pass	fail	fail	fail
2	test Context1	pass	pass	pass	pass

Figure 5.29: Test outcomes take two

Computable constraint modeling

Mathematics isn't just fun; it's also another means by which you can verify aspects of systems. For example, in the **Architectural trade studies** recipe in *Chapter 3, Developing System Architecture*, we created a mathematical model to evaluate design alternatives as a set of equations, converting raw properties – including measurement accuracy, mass, reliability, parts cost and rider feel – into a computed “goodness” metric for the purpose of comparison. Using trade studies is a way to verify that good design choices were made. We did this using SysML constraints and parametric diagrams to render the problem and “do the math.”

Math can address many problems that come up in engineering, and SysML parametric diagrams provide a good way to render, compute, and resolve such problems and their solutions. An archetypal example is computing the total weight of a system. This can be done by simply summing up the weights of all its constituent parts. However, far more interesting problems can be addressed.

Because math is the language of quantities, it is very general, so it is a challenge to come up with a workflow that encompasses a wide range of addressable problems. But we are not alone. Thinkers such as Polya (Polya, G. *How to Solve It: A new aspect of mathematical methods*, Princeton University Press, 1945) or recent rereleases such as those available on Amazon.com and Wickelgren (Wickelgren, W. *How to Solve Problems: Elements of a theory of problems and problem solving*, W. H. Freeman and Co, 1974). have proposed means by which we can apply mathematics to general kinds of problems. As we did with in the trade studies, we can capture an approach that employs mathematics and SysML to provide a means to mathematically analyze and verify system aspects.

Purpose

The purpose is to solve a range of problems that can be cast as a set of equations relating to values of interest. We will call problems of this type **Mathematically-Addressable Problems (MAPs)**.

Inputs and preconditions

The input to the recipe is a MAP that needs to be solved, such as the emergent properties of a system being developed.

Outputs and postconditions

The output of the recipe is the answer to the problem – typically, a quantified characterization of the emergent property (also known as *the answer*).

How to do it

Figure 5.30 shows the steps involved:

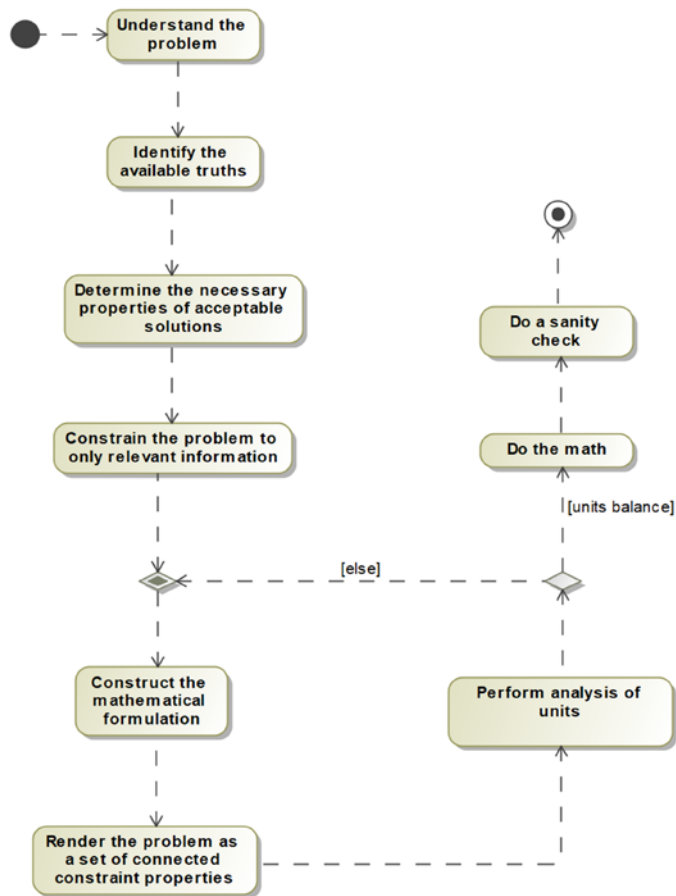


Figure 5.30: Computational constraint modeling workflow

Understand the problem

The first step in this recipe is to understand the problem at hand. In MBSE, that generally means:

- Identifying the structural elements (blocks, value properties, and so on).
- Discovering how they connect.
- Understanding how the elements behave both individually and collectively.
- Reviewing the qualities of services of those behaviors (such as worst-case performance).
- Looking at other quantifiable properties of interest for the system or its elements.

Identify the available truths

This step is about finding potentially relevant values and constraints. This might be universals – such as the value of the gravitational constant on the surface of the earth is 9.81 m/s^2 or that the value of π to nine digits is 3.14159265 . It might be constraints – such as the limit on the range of patient weights (0.5 KG to 300 Kg). It might be quantifiable relations – such as $F = ma$.

Determine the necessary properties of acceptable solutions

This is the step where the problem gets interesting. I personally cast this as “what does the solution smell like?”. I want to understand what kind of thing I expect to see when a solution is revealed. Is it going to be the ratio of engine torque to engine mass? Is it the likelihood of a fault manifesting as an accident in my safety critical system? What is the nature of the result for which I am looking? Many times, this isn’t known or even obvious at the outset of the effort but is critical in identifying how to solve the problem.

Constrain the problem to only relevant information

Once we understand the problem, the set of potentially useful information, and the nature of the solution, we can often limit the input information to a smaller set of truly relevant data. By not having to consider extraneous information, we simplify the problem to be solved.

Construct the mathematical formulation

In this step, we develop the equations that represent the information and their relations in a formal, mathematical way. Remember, *math is fun*.

Render the problem as a set of connected constraint properties

The equations will be modeled as a set of *constraints* owned by SysML *constraint blocks*. The input and output values of those equations are modeled as *constraint parameters*. Then the equations are linked together into computation sets as *constraint properties* on a parametric diagram.

Perform analysis of units

In complex series of computations, it is easy to miss subtle problems that could be easily identified if you just ensure that the units match. Perhaps you end up trying to add inches to meters or add values of force and power. Keeping track of units and ensuring that they balance is a way to find such mistakes.

Do the math

Cameo can evaluate parametrics with its Simulation Toolkit. If you simulate a context that has parametrics, these are evaluated before the actual simulation begins in a pre-execution computational step.

Do a sanity check

A sanity check is a quick verification of the result by examining its reasonableness. One quick method that I often employ is to use “approximate computation.” For example, if you say that $1723/904$ is 1.906, I might do a sanity check by noting that $1800/900$ is 2, so the actual value should be a little less than 2.0. If you tell me that a medical ventilator should deliver 150 L/min, I might perform a sanity check by noting that an average breath is about 500 ml (at rest) at an average rate of around 10 breaths/min, so I would expect a medical ventilator to deliver something close to 5 L/min, far shy of the suggested value.

More elaborate checks can be done. For a critical value, you might compute it using an entirely different set of equations, or you might perform *backward computation* whereby you start with the end result and reverse the computations and see if you end up with the starting values.

Example

The *Architectural trade studies* recipe example from *Chapter 3, Developing System Architectures*, showed one use for computable constraint models. Let’s do another.

In this example, I want to verify a resulting computation performed by the Pegasus system, regarding simulated miles traveled as a function of gearing and cadence. We will build up and evaluate a SysML parameter model for this purpose.

Understand the problem

The potentially relevant properties of the **Bike** and **Rider** include:

- Chain ring gear, in the number of teeth.
- Cassette ring gear, in the number of teeth.

- Wheel circumference, inches.
- Cadence of the rider, in pedal revolutions per minute.
- Power produced by the rider, in watts.
- Weight of the rider and bike, in kilograms.
- Wind resistance of the rider, in Newtons.
- Incline of the road, in % grade.
- How long the rider rides, in hours.

The desired outcome is to know how far the simulated bike travels during the ride.

Identify the available truths

There are some constraints on the values these properties can achieve:

- Chain ring gears are limited to 28 to 60 teeth.
- Cassette ring gears are limited to 10 to 40 teeth.
- The wheel circumference for a normal road racing bike is 82.6 inches (2098 mm).
- Reasonable rider pedal cadence is between 40 and 120 RPM.
- Maximum output power for elite humans is about 2000W, with 150–400W sustainable for an hour or more (depending on the person).
- The UCI limits the weight of road racing bikes to no less than 15 lbs (6.8 Kg) but 17–20 lbs is more typical.
- Rider weight is generally between 100 to 300 lbs (45.3 Kg to 136 Kg).
- Wind resistance is a function of drag coefficient, cross-sectional area, and speed (<https://ridefar.info/bike/cycling-speed/air-resistance-cyclist/>).
- Road inclines can vary from -20 to +20% grade with an approximately normal distribution of around 0%. A notable exception is the 32% climb in the *Savageman Triathlon* “Westernport Wall,” <https://kineticmultisports.com/races/savageman/>.
- Ride lengths of interest are between 0.5 and 8 hours.

Determine the necessary properties of acceptable solutions

What we expect is a distance in miles as determined by a ride of a certain length of time in specific gearing, with other parameters being set as needed.

Constrain the problem to only relevant information

A little thought on the matter reveals that many of the properties outlined are not really relevant to the specific computation. Assuming no wheel slippage on the road, the only properties necessary to determine distance are:

- Chain ring gear
- Cassette ring gear
- Pedal cadence
- Wheel circumference
- Ride time

If the other properties vary, they will manifest as changing one or more of these values. For example, if wind resistance changes due to a headwind, then the rider will either pedal more slowly, change gear, or ride longer in time to cover the same distance.

Construct the mathematical formulation

There are a number of equations we must construct, with several of them solely for converting units:

- Gear ratio = chain ring teeth / cassette ring teeth.
- Gear inches = gear ratio * wheel circumference in inches.
- Gear miles = gear inches / 12 / 5280.
- Revolutions per hour = revolutions per minute * 60.
- Speed in MPH = revolutions per hour * gear miles.
- Distance = speed * duration.

Render the problem as a set of connected constraint properties

These equations are rendered as constraint blocks, as you can see in *Figure 5.31*. The inputs and outputs are shown as the constraint parameters, shown as included boxes along the edge of the constraint blocks.

The front gear and rear gear are modeled as **Integers**, but all others are represented as **Reals**:

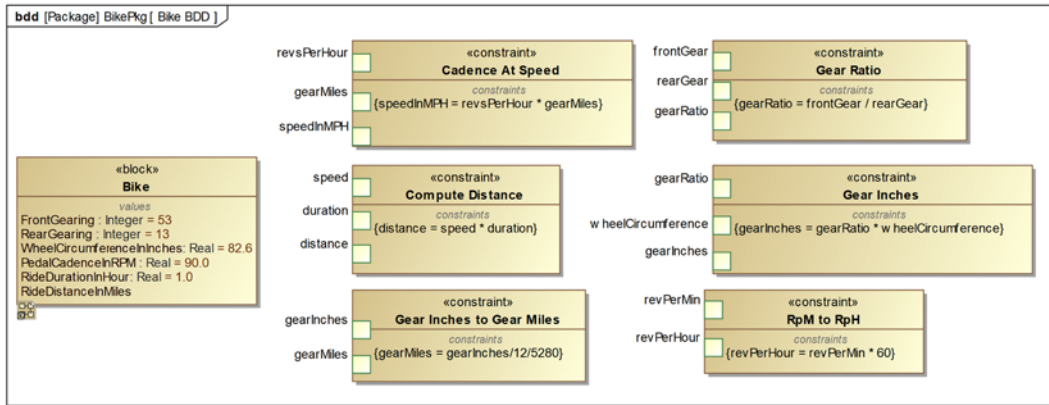


Figure 5.31: Bike gear constraint blocks

A parametric diagram is a specialized form of an internal block diagram. The usage of the constraint blocks, known as constraint properties, in the computational context is shown in *Figure 5.32*:

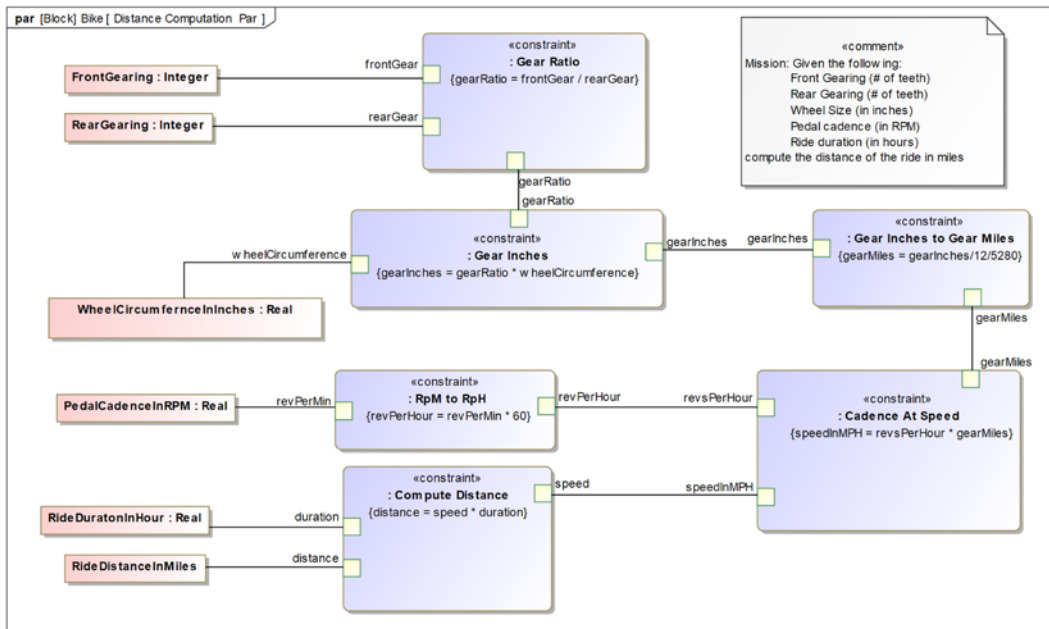


Figure 5.32: Gear calculation parametric diagram

Note the binding connectors relate the constraint parameters together as well as the constraint parameters to the value properties of the **Bike** block. This will become important later in the recipe when we want to evaluate different cases.

Perform analysis of units

This problem is computationally simple but is easy to screw up because of the different units involved. Wheel circumference is measured in inches. Pedal cadence in RPM. However, we ultimately want to end up with the distance in miles.

We can recast the equations as units to ensure that we got the conversions right:

- Gear ratio = chain ring teeth / cassette ring teeth
 - Real = teeth / teeth (note: Real is unitless)
- Gear inches = gear ratio * wheel circumference
 - Inches / revolution = real * inches / revolution
- Gear miles = gear inches / 12 / 5280
 - miles / revolution = inches / revolution / (inches/foot) / (feet/mile)
 - = feet / revolution / (feet / mile)
 - = miles / revolution
- Revolutions per hour = revolutions per minute * 60
 - RPH = revolutions / min * (minute / hour)
 - = revolutions / hour
- Speed in MPH = revolutions per hour * gear miles
 - MPH = (revolutions / hour) * (miles / revolution)
 - = miles / hour
- Distance = speed * duration
 - Miles = miles / hour * hour
 - = miles

The units balance, so it seems that the equations are well-formed.

Do the math

Before we do the math, it will be useful to review SysML *instance specifications* a bit, as they are not covered in much detail in most SysML tutorials. An instance specification is the specification of an instance defined by a block. Instance specifications have slots that hold specific values related to the value properties of the block. That means we can assign specific values to the slots corresponding to value properties. In practice, we can assign a partial set of slot values and use the parametric diagram to compute the rest. We can then save the resulting elaborated instance specification as an evaluation case, much like we might have multiple parts instantiating a block. This concept is illustrated in *Figure 5.33*:

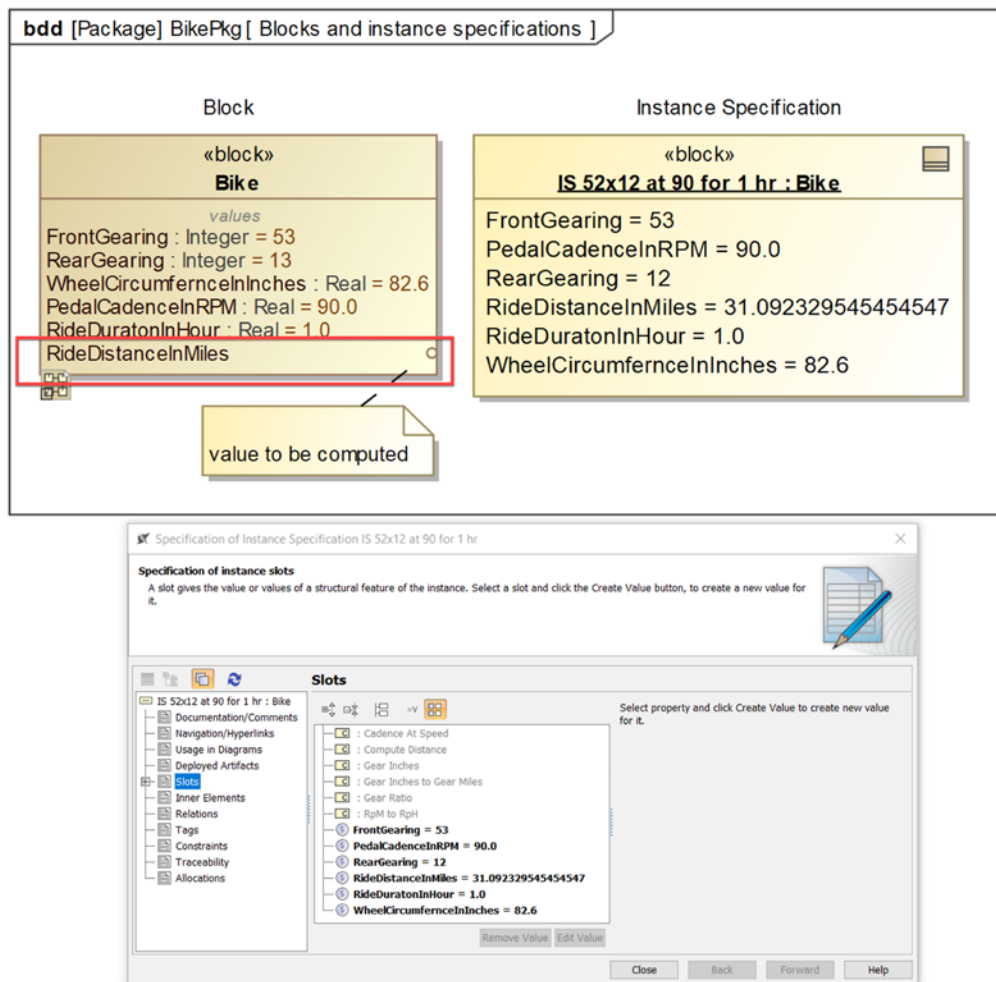


Figure 5.33: Blocks and instance specifications

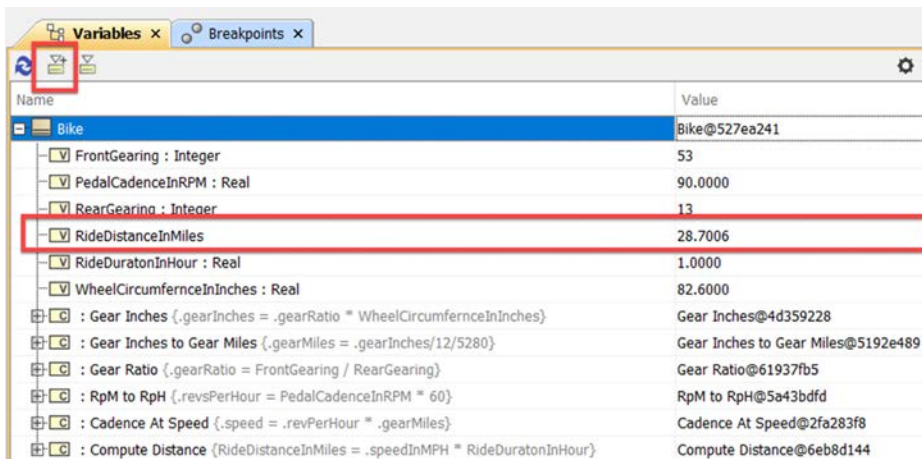
To perform the math, we'll define a set of instance specifications for the cases that we want to compute. In this example, we'll look at six cases:

- 34 x 13 @ 90 RPM for 1 hour
- 34 x 18 @ 90 RPM for 1 hour
- 34 x 13 @ 105 RPM for 0.75 hour
- 53 x 13 @ 90 RPM for 1 hour
- 53 x 18 @ 90 RPM for 1 hour
- 53 x 13 @ 105 RPM for 1.5 hour

We will create a separate instance specification for each of these cases. Each of these cases is an instance specification typed by the block **Bike**. We will name these as "IS" + gearing + pedal cadence, such as **IS 34x13 at 90 for 1 hour**.

To perform the computations, we will use the Cameo Simulation Toolkit. This toolkit performs a pre-execution computation – that is, it evaluates the parameters in the simulation before performing any behaviors.

To actually perform the calculations is easy – if you know how (although it isn't well explained in the documentation). Simply right-click on the **Bike** block and select **Simulation > Run**. Although the block has default values for the value properties, you can set the values that you want to compute for the different cases. Once completed, you can click the **export to new instance** button in the Simulation Toolkit (see *Figure 5.34*) to save the values as an instance specification. Select the package where you want the instance placed. You'll have to rename it after it is created:



Name	Value
Bike	Bike@527ea241
FrontGearing : Integer	53
PedalCadenceInRPM : Real	90.0000
RearGearing : Integer	13
RideDistanceInMiles	28.7006
RideDurationInHour : Real	1.0000
WheelCircumferenceInInches : Real	82.6000
Gear Inches { .gearInches = .gearRatio * WheelCircumferenceInInches }	Gear Inches@4d359228
Gear Inches to Gear Miles { .gearMiles = .gearInches/12/5280 }	Gear Inches to Gear Miles@5192e489
Gear Ratio { .gearRatio = FrontGearing / RearGearing }	Gear Ratio@61937fb5
RpM to RpH { .revsPerHour = PedalCadenceInRPM * 60 }	RpM to RpH@5a43bdfd
Cadence At Speed { .speed = .revPerHour * .gearMiles }	Cadence At Speed@2fa283f8
Compute Distance { RideDistanceInMiles = .speedInMPH * RideDurationInHour }	Compute Distance@6eb8d144

Figure 5.34: Simulation Toolkit variables window

I updated the computed properties and then constructed a table of the values for the different evaluation cases (*Figure 5.35*):

#	Name	FrontGearing : Integer	RearGearing : Integer	WheelCircumferenceInches : Real	PedalCadenceInRPM : Real	RideDurationInHour : Real	RideDistanceInMiles
1	IS 53x13 at 105 for 90 min	53	13	82.6	105	1.5	50.2261
2	IS 53x13 at 90 for 1 hour	53	13	82.6	90	1	28.7006
3	IS 53x18 at 90 for 1 hour	53	18	82.6	90	1	20.7282
4	IS 34x13 at 90 for 1 hour	34	13	82.6	90	1	18.4117
5	IS 34x13 at 105 for 45 min	34	13	82.6	105	0.75	16.1102
6	IS 34x18 at 90 for 1 hour	34	18	82.6	90	1	13.2973

Figure 5.35: Instance table of computed outcomes

Do a sanity check

Are these values reasonable? As an experienced cyclist, I typically ride around 90 RPM, and in a gear 53x18 gear, I would expect to go about 20 miles, so the computed result for that case seems right. As I switch to a small 13-cog cassette, I would expect to go about 50% further, so the 28 miles at 53x13 seems about right. In a smaller front chain ring of 34 with a small cog (13), I'd expect to go about 80% as far as in the 53x18, so that seems reasonable as well.

Traceability

Traceability in a model means that it is possible to navigate among related elements, even if those elements are of different kinds or located in different packages. This is an important part of model verification because it enables you to ensure the consistency of information that may be represented in fundamentally different ways or different parts of the model. For example:

- Are the requirements properly represented in the use case functional analysis?
- Do the design elements properly satisfy the requirements?
- Is the design consistent with the architectural principles?
- Do the physical interfaces properly realize the logical interface definitions?

The value of traceability goes well beyond model verification. The primary reasons for providing traceability are to support:

- Impact analysis: determine the impact of change, such as:
 - If I change this requirement or this design element, what are the elements that are affected and must also be modified?
 - If I change this model element, what are the cost and effort required?

- Completeness assessment: how done am I?
 - Have I completed all the necessary aspects of the model to achieve its objectives, such as with a requirement, safety goal, or design specification?
 - Do I have an adequate set of test cases to verify the requirements?
- Design justification: why is this here?
 - Safety standards require that all design elements exist to meet requirements. This allows the evaluation of compliance with that objective.
- Consistency: are these things the same or do they work together? For example:
 - Are the requirements consistent with the safety concept?
 - Does the use case activity diagram properly refine the requirements?
 - Does the implementation meet the design?
- Compliance:
 - Does the model, or elements therein, comply with internal and external standards?
- Reviews:
 - In a review or inspection of some model aspect, is the set of information correct, complete, consistent, and well-formed?

By way of a very simple example, *Figure 5.36* schematically shows how you use a trace matrix. The rows in the figure are the requirements and the columns are different design elements. Requirement **R2** has no trace relation to a design element, indicating that is an unimplemented requirement. Conversely, design element **D3** doesn't realize any requirements, so it has no justification:

		Design / Implementation Elements				
		D1	D2	D3	D4	D5
Requirements	R1	x				x
	R2					
	R3		x			
	R4				x	

← Unimplemented requirement

↑ Gold plating?

Figure 5.36: Traceability

Some definitions

Traceability means that a navigable relation exists between all related project data without regard to the data location in the model, its representation means, or its format.

Forward traceability means it is possible to navigate from data created earlier in the project to related data produced later. Common examples would be to trace from a requirement to a design, from a design to implementation code, or from a requirement to related test cases.

Backward traceability means it is possible to navigate from data created later in the project to related data produced earlier in the process. Common examples would be to trace from a design element to the requirements it satisfies, from a use case to the requirements that it refines, from code back to its design, or from a test case to the requirements it verifies.

Trace ownership refers to the ownership of the trace relation, as these relations are directional. This relation is *almost always* in the backward direction. For example, a design element owns the satisfy relation of the requirement. Forward traceability is supported by tools via automation and queries by looking at the backward trace links to identify their owners. This is counterintuitive to many people, but it makes sense. A requirement *in principle* shouldn't know how it is implemented, but a design *in principle* needs to know what requirements it is satisfying. Therefore, the «satisfy» relation is owned by the design element and not the requirement.

Trace matrices are tabular relations of the relations between sets of elements. Cameo supports the creation of matrix layouts that define the kinds of elements for the rows and columns and the kind of relation depicted. Matrix views visualize data based on the layout specifications. Multiple kinds of relations can be visualized in the same matrix, but in general, I recommend different tables to visualize different kinds of trace links.

Types of trace links

In SysML, trace relations are all stereotypes of dependency. *Table 5.2* shows the common relations in SysML:

Relation	Source type	Target type	Description
«trace»	Any	Any	A general relation that can be used in all circumstances where a navigable relation is needed. A common use is to relate requirements in different contexts, such as system requirements (source) and stakeholder requirements (target).

«copy»	Requirement	Requirement	Establishes that a requirement used in a context is a replica of one from a different context.
«deriveReq»	Requirement	Requirement	Use when the source requirement is derived from the target requirement; this might be used from subsystem to system requirements, for example.
«refine»	Any	Any	Represents a model transformation; in practice, it is often used to relate a requirement (target) with a behavioral description (source), such as an activity diagram.
«allocate»	Any	Any	Relates elements of different types or in different hierarchies; in practice, it is most commonly used to relate a design element (source) to a requirement (target) and is, in this usage, similar to «satisfy».
«satisfy»	Design element	Requirement	A relation from a model element (source) and the requirement that it fulfills (target).
«verify»	Test case	Requirement	Indicates that the source is used to verify the correctness of the target. The source may be any kind of model element, but it is almost always a test case. See the <i>Model-based testing</i> recipe, earlier in this chapter, for more information.

Table 5.2: SysML trace relations

You may, of course, add your own relations by creating your own stereotype of dependency. I often use «represents» to trace between levels of abstraction; for example, a physical interface «represents» a logical interface.

It is reasonable to ask why there are different kinds of trace relations. From a theoretical standpoint, the different kinds of relation clarify the relation's semantic intent. In practice, different tables can be constructed for the different kinds of relations and make them easier to apply in real projects.

Traceability and agile

Many Agilistas devalue traceability; for example, Scott Ambler says:



Too many projects are crushed by the overhead required to develop and maintain comprehensive documentation and traceability between it. Take an agile approach to documentation and keep it lean and effective. The most effective documentation is just barely good enough for the job at hand. By doing this, you can focus more of your energy on building working software, and isn't that what you're really being paid to do?

See <http://www.agilemodeling.com/essays/agileRequirementsBestPractices.htm> for more information.

However, in my books on agile systems engineering and agile development for real-time, safety-critical software, I make a case that traceability is both necessary and required:



Traceability is useful for both change impact analysis and to demonstrate that a system meets the requirements or that the test suite covers all the requirements. It is also useful to demonstrate that each design element is there to meet one or more requirements, something that is required by some safety standards, such as DO-178B.

See Bruce Douglass, Ph.D, *Real-Time Agility* (Addison-Wesley, 2009): <https://www.amazon.com/Real-Time-Agility-Harmony-Embedded-Development/dp/0321545494> for more information.

In my agile projects, I add traceability relations as the work product content stabilizes to minimize rework, usually after most of the work has been done but before the system is verified and reviewed.

Purpose

The purpose of traceability is manifold, as stated earlier in this recipe. It enables model consistency and completeness verification as well as the performance of impact analysis. Additionally, it may be required by standards a project must meet.

Inputs and preconditions

The sets of elements being related are well enough developed to justify the effort to create trace relations.

Outputs and postconditions

Relations between related elements in different sets have been created and are captured in one or more trace matrices.

How to do it

Figure 5.37 shows the simple workflow for this recipe. Although the steps themselves are simple, not thinking deeply about the steps may result in unsatisfactory outcomes:

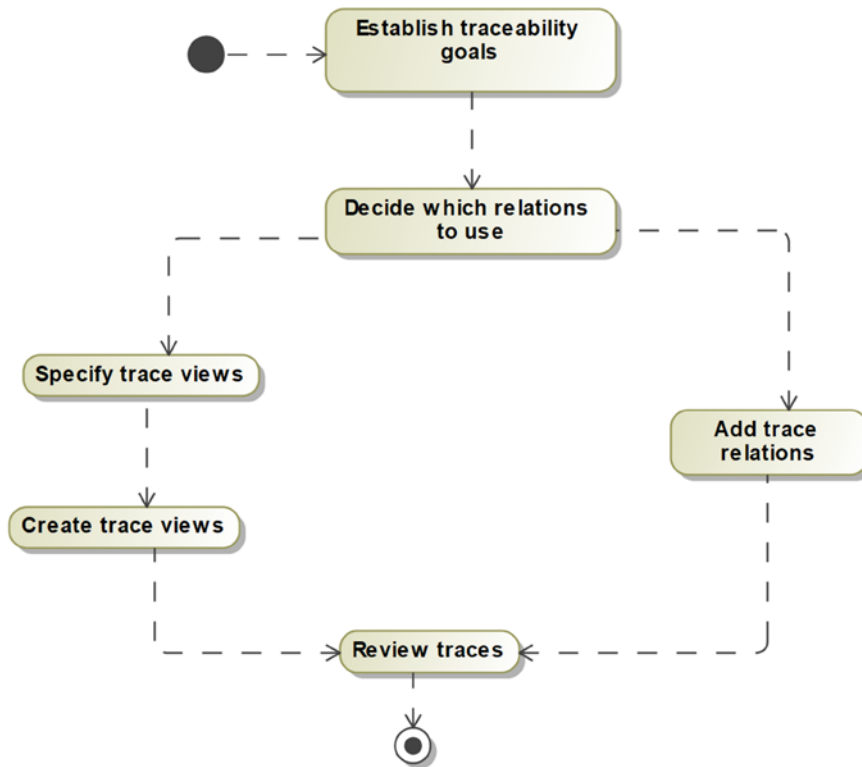


Figure 5.37: Create traceability

Establish traceability goals

Traceability can be done for a number of specific reasons. Different stakeholders care about demonstrating that the design meets the requirements, versus test cases adequately covering the design, versus ensuring that the stakeholder requirements are properly represented in the system requirements.

All of these are reasons to create traceability, but different trace views should be created for each purpose.

Common traceability goals in MBSE include tracing between:

- Stakeholder needs and system requirements.
- System and subsystem requirements.
- Subsystem requirements and safety goals.
- Facet (e.g., software or electronic) requirements and system requirements.
- Requirements and test cases.
- Design and test cases.
- Design and safety goals.
- Logical schemas and corresponding physical schemas, including:
 - Data schema
 - Interfaces
- Model products and standard objects, such as ASPICE, DO-178, and ISO26262.
- Design and implementation work products.

For your project, you must decide on the objectives you want to achieve with traceability. In general, you will create a separate traceability view for each trace goal.

Decide which relations to use

SysML has a number of different kinds of stereotyped dependency relations that can be used for traceability purposes. In modern modeling tools, you can easily create your own to meet any specified needs you have. For example, I frequently create a stereotype of dependency named «represents» to specifically address trace relations across abstraction levels, such as between physical and logical schemas.

Which relation you use is of secondary importance to establishing the relations themselves, but making good choices in the context of the set of traceability goals that are important to you can make the whole process easier. *Table 5.3* shows how I commonly use the relations for traceability:

Source	Target	Relation	Purpose
Requirement	Requirement	«trace»	Relate different sets of requirements, generally from later developed to earlier, such as system (source) to the stakeholder (target).
Use case	Requirement	«trace»	Show which requirements are related to a use case.
Requirement	Requirement	«deriveReq»	Relate a more detailed requirement (source) to a more abstract one.
Design element	Requirement	«satisfy»	Identify which requirements a design element satisfies.
Design element	Requirement	«allocate»	Identify which design elements (source) requirements (target) are allocated.
Model element	Requirement	«refine»	Provide a detailed, often behavioral, representation of a textual requirement, such as an activity or state diagram elaborating on one or more requirements.
Test case	Requirement, design element	«verify»	Relate test cases to either the requirements or design elements.
Comment	Model element	«rationale»	Provide an explanation for the existence or representation of one or more model elements.
Model element	Model element	«represents»	Relate elements that are intended to be the “same thing” but at different levels of abstraction, such as logical versus physical data schema.

Table 5.3: Common trace relation usage

Specify trace views

Once the element type and relations are decided, you must decide the best way to represent them. The first decision here is whether to use a matrix or a table.

A matrix is often the first choice for traceability. A matrix is a relation table between element sets, one set shown in the rows and the other in the columns. The contents of the cells in the matrix indicate the presence or absence of the desired relation.

This is a great view for determining the presence or absence of relations. For example, the «satisfy» matrix between requirements and design elements is very useful to determine absence; an empty requirements row means that the requirement isn't represented in the design while an empty design element column means that the design element isn't there to meet any requirements. For practical reasons, I prefer the smaller set of elements to be shown as columns and the larger set of elements to be shown as the rows. The downside of a matrix is that you can't see anything other than the name of the involved elements and the presence or absence of the desired relation. If you want to see other data, such as the text of the requirements, a list of the value properties of blocks, or the metadata tags, then a table is preferable.

Tables consist of rows of data, one per primary table element, along with information owned by or about that element about which you care. Table elements that have properties – such as textual specifications, value properties, or metadata held in owned tags – can display that data in an easy-to-read summary form. In the context of traceability, one of the kinds of properties the elements can own are relations, such as «trace» or «verify», and these can be shown in tables as well. In Cameo, this is done with the **Columns** tool above the table.

You can even create both tables and matrices.

Create trace views

Once the layouts are defined, create views that apply the layout to the specific set of elements.

Add trace links

You can use the trace views as a means for creating the trace views if you like, but you are not required to do so. I often create a use case diagram whose mission is to show the «trace» relation between one use case and a set of requirements, but then use a trace matrix to summarize the results of all such use cases. It is perfectly reasonable to simply create an (empty) matrix and add the relations directly to the matrix. The reverse can also be done: if you create the relations in a matrix, you can create a diagram, drag onto it the relevant elements, and expose those relations with the right-click menu option **Display > Display All Paths**. However you do it, this step adds the relations of the selected type to the appropriate elements in your model.

Review traces

Once the previous steps are complete, you can open the views and use the trace views to achieve your traceability objectives.

Example

In this example, we'll consider the relation of the system architecture to the system requirements. In the *Subsystem and component architecture* recipe of *Chapter 3, Developing System Architectures*, we created a subsystem architecture for the **Pegasus** system. While incomplete (it was for an early iteration after all), we created subsystems to meet those interfaces. Let's apply this recipe to develop a traceability view from the subsystems to the requirements.

Establish traceability goals

In this example, the purpose of traceability is to show all the requirements allocated to the iteration and allocated to a subsystem. If there are requirements in the iteration that are not allocated, then we can either directly allocate the requirements or create derived requirements that are then allocated.

Decide which relations to use

We will use the «satisfy» relation. It's also very common to use «allocate» to relate requirements to architectural elements. In this case, we will use «satisfy» between the subsystems and the requirements.

Specify trace views

We will use a matrix of **Block vs Requirement** «satisfy» relations from the subsystem blocks to the requirements for our view.

Create trace views

The matrix view uses the layout to display the specified data. In Cameo, the layout information is provided above the matrix (*Figure 5.38*):



Criteria	
Row Element Type: Block	Column Element Type: Requirement
Row Scope: Architecture Design Pkg	Column Scope: ReqsPkg
Dependency Criteria: Satisfy	Direction: Both
	Show Elements: With relations

Figure 5.38: Satisfy matrix view layout

Add trace links

There are a couple of ways to populate the «satisfy» relations from the blocks. In this case, I prefer to diagram the relations rather than work in the matrix directly. The reason is that on the diagram, I can see the text of the requirement, while in the matrix, I can only see the name of the requirement.

Figure 5.39 shows one of the requirement diagrams created to create the relations:

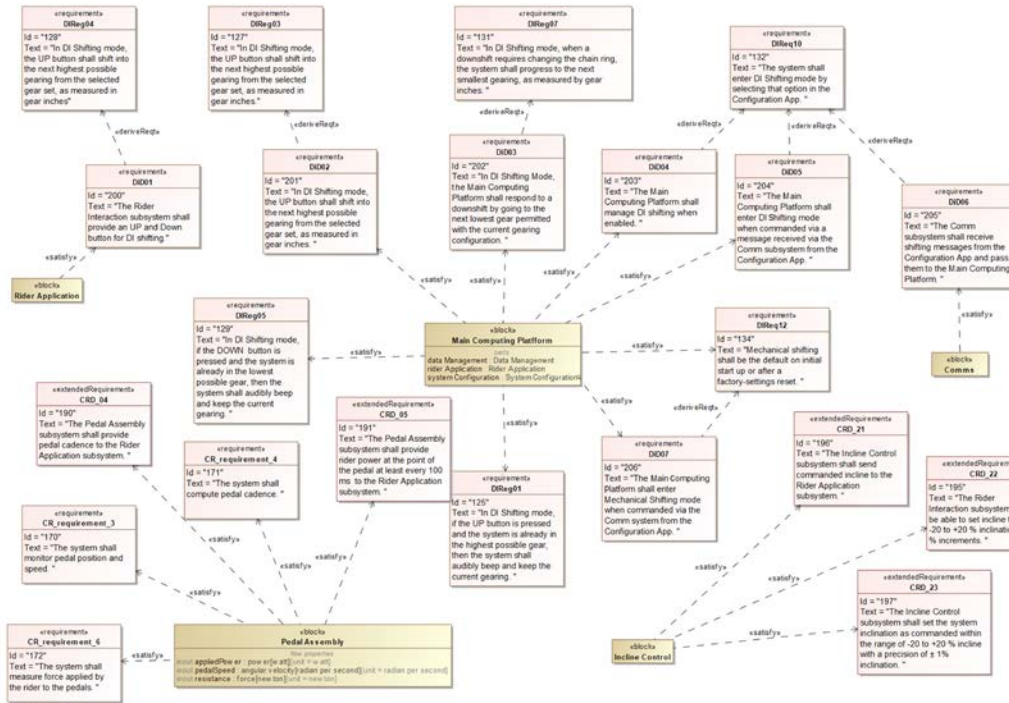


Figure 5.39: Example requirements diagram to add satisfies relations

Review traces

Figure 5.40 shows a portion of the resulting allocation relations from the subsystem elements to the requirements. The property *with relations* is used to only show rows and columns where a relation exists:

The screenshot shows a 'Subsystem-Requirement satisfy matrix' with the following structure:

- Legend:** Shows 'Satisfy' with a blue arrow icon.
- Table:**
 - Columns (Requirements):** DReq1, DReq5, DReq12, CR_requirement_1, CR_requirement_2, CR_requirement_3, CR_requirement_4, CR_requirement_5, CR_requirement_6, CR_requirement_7, CR_requirement_8, CR_requirement_9, CR_requirement_10, CR_requirement_11, CR_requirement_12, CR_requirement_13, CR_requirement_14, CR_requirement_15, CR_requirement_16, CR_requirement_17, CR_requirement_18, CR_requirement_19, CR_requirement_20, CR_requirement_21, CR_requirement_22, CR_requirement_23, CR_requirement_24, CR_requirement_25, CRD_01, CRD_02, CRD_03, CRD_04, CRD_05, CRD_06, CRD_07, CRD_08, CRD_09, CRD_10, CRD_11, CRD_12, CRD_13, CRD_14, CRD_15, CRD_16, CRD_17, CRD_18, CRD_19, CRD_20, CRD_21, CRD_22, CRD_23, CRD_24, CRD_25, DReq10, DReq11, DReq12, DReq13, DReq14, DReq15, DReq16, DReq17, DReq18, DReq19, DReq20, DReq21, DReq22, DReq23, DReq24, DReq25.
 - Rows (Subsystems):** Architecture Design Plug, Comms, Data Management, Incline Control, Main Computing Platform, Pedal Assembly, Pegasus, Rider Application, System Context.
- Grid:** Blue arrows indicate 'satisfies' relationships between subsystems and requirements.

Figure 5.40: Subsystem-Requirement satisfy matrix

It is common to also show a matrix view where no relations exist. This allows easy identification of requirements not allocated to design elements or design elements that have no relation to the requirements. See *Figure 5.41*:

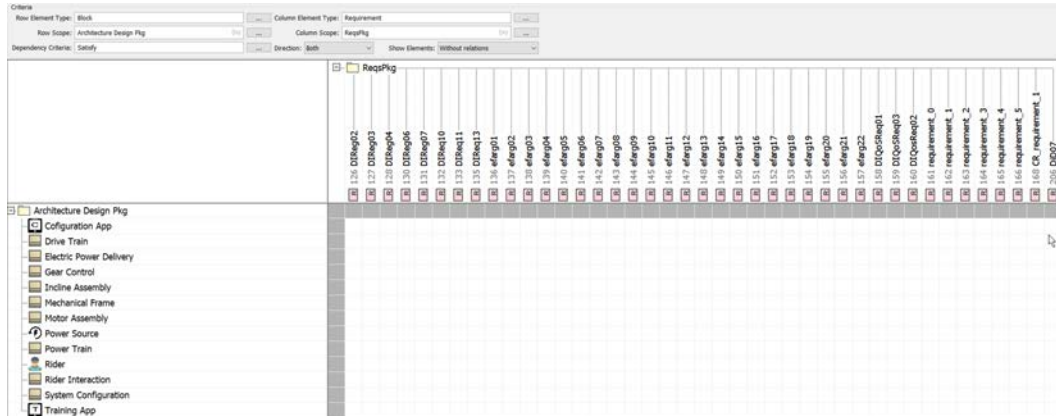


Figure 5.41: Subsystem-Requirements satisfy matrix – missing relations

Effective Reviews and walkthroughs

At the beginning of this chapter, I talked about reviews being the easiest but weakest form of model verification. This shouldn't be construed to mean that I don't believe reviews have value. Properly applied, reviews are a very useful adjunct to other, more rigorous forms of verification. Reviews can contribute to both syntactic ("compliance in form") and syntactic ("compliance in meaning") verification. They are relatively easy to perform and can provide input that is otherwise difficult to obtain.

Syntactic reviews, performed by quality assurance personnel, demonstrate compliance of the model to the project modeling guidelines, including the organization of the model, the presence of required elements and views, the structuring of those views, compliance with naming conventions, and more.

Semantic reviews are performed by **Subject Matter Experts (SMEs)**. Some of the questions such reviews can address include:

- Does the model appropriately cover the necessary subjects (breadth)?
- Does the model include an appropriate level of detail (depth)?
- Is the level of precision, accuracy, and fidelity appropriate?
- Does the model accurately represent the physics of the situation?

- Are relevant standards properly taken into account?
- Is the technology accurately represented?
- Are all primary (“sunny day”) conditions considered?
- Are exceptional and edge cases (“rainy day”) cases adequately addressed?
- Is there anything of significance missed or inappropriately modeled?
- How will the correctness be verified?
- Is the information (typically the views) consumable by the stakeholders?

Reviews can be a valuable adjunct to other verification means. However, it is easy, and far too common, to perform them poorly in ways that are both expensive and ineffective. Reviews are very expensive because they typically have several people engaged for a significant number of hours. It is imperative that the meeting time is used efficiently. It is common for people to come unprepared, wasting others’ time with questions to which they should already have answers. It is also common to “go down the rabbit hole” in exploring solutions – that is, out of scope of the review. It is enough to identify the problem and move on. If a reviewer has a suggestion for a solution, they can present it to the author *after the meeting*. Further, because authors are putting their egos on the line, it is important that review comments address the product aspects and not what the author did wrong (“The product has this defect”, not “You made a mistake here”). Subsequent reviews of the same product should be limited solely to the resolution of action items, unless new information appears that invalidates the previous review.

The recipe described here is adapted from the notion of *Fagan inspections*, which were originally a means for software source code review (https://en.wikipedia.org/wiki/Fagan_inspection). I’ve adapted them to apply to engineering work products in general and especially to models.

Purpose

All models are abstractions but some are *useful abstractions*. A review ascertains whether they are the *right abstractions* covering the *appropriate topics* with adequate precision, depth, and breadth and represent the system *correctly* in such a way that the model *fulfills its intended purpose*.

The recipe provided here is very general and can be applied to any technical work product, but our focus will be on the review of models and their content and views.

Note that this recipe describes how to conduct a review only. There are other reasons to have meetings with technical work products, including the dissemination of information held within the work product. Those may be valid and important meetings to have but they are not reviews.

The purpose of a review is to *verify the content of a collection of project data, either syntactically, semantically, or both.*

Inputs and preconditions

A body of work is completed to the extent that the project can profitably conduct a review.

Outputs and postconditions

The primary outcomes are either a quality assurance record that the review was performed and no significant defects or issues were found, an action list of issues for the product authors to resolve, or both.

How to do it

Figure 5.42 shows the workflow for the recipe. A couple of main points in the figure should be emphasized.

First, the reviewers are expected to come to the review having examined the work products in detail. This is not a meeting for *explanation*, although questions of clarification are permitted. Second, solutions to identified issues are not discussed; suggestions can be discussed directly with the author *after* the review. Lastly, subsequent reviews are limited to the resolution of the action items from previous reviews:

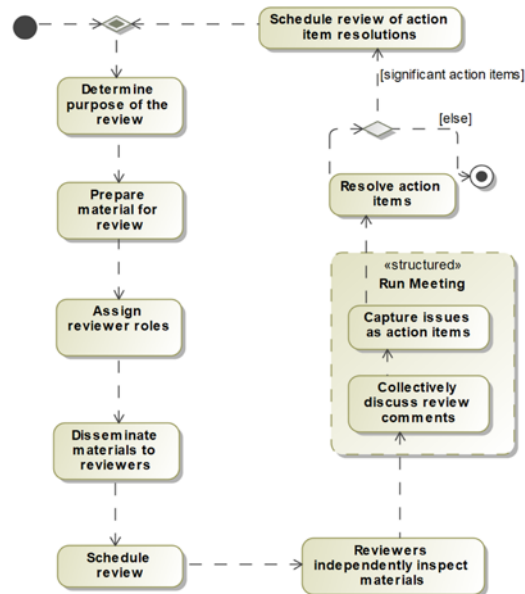


Figure 5.42: Conduct review workflow

Establish the purpose of the view

Real models are too large to be reviewed in their entirety in a single review. Each review must have a specific purpose or intent that can be performed in a reasonable timeframe and consider only the subset of the model relevant to that purpose. Some common review intents are performing the following actions:

- Evaluate and understand systems engineering handoff models.
- Evaluate the use cases and their details.
- Evaluate architectural models.
- Evaluate design models.
- Evaluate the adequacy and coverage of a test plan and its details.
- Evaluate the result of the actions taken after a previous review.

Prepare materials for review

In this step, the material is gathered together for review. There are almost always multiple artifacts that are reviewed together. For example, a review of a use case model usually includes the following artifacts:

- Relevant use case diagrams
- Corresponding requirement text
- Use case sequence diagrams
- Use case state machines and activity diagrams
- Quality of service constraints
- Use case descriptions
- List of actions from previous reviews (if applicable)

For the review of a design model, common artifacts include:

- For each use case in the design model:
 - Block definition diagram(s) showing the collaboration of elements realizing the use case
 - Internal block diagram(s) showing how the collaborating elements connect
 - Original use case sequence diagram(s)
 - Elaborated design sequence diagrams showing the interaction
 - State machine diagrams for stateful classes

- Activity diagrams for complex algorithms
- List of actions from previous reviews (if applicable)

In Cameo, **smart packages** are special packages that contain elements by reference. An element can be added manually (such as by drag and drop) or automatically by defining criteria for inclusion. This feature can be used to gather up the packages, model elements, and diagrams relevant to a particular review.

Assign reviewer roles

Reviews have a number of roles that people can fulfill during the review. These need to be assigned to actual people to play those roles:

- **Review coordinator:**

This role organizes and runs the meeting, ensuring compliance with the review's rules of conduct, such as "Direct comments to product, not the author" and "Discuss solutions with the author after the meeting."

- **Product owner/presenter:**

This is generally one of the authors of the material under review. This person is technically knowledgeable and can answer questions of clarification during the meeting.

- **SME reviewer:**

This role is expected to provide a critical eye from a semantic viewpoint, whether it is about the specific technology, tool use, or engineering.

- **Quality Assurance (QA) reviewer:**

The QA role is there to ensure syntactic correctness and review the work product for compliance with relevant standards, including both internal standards (such as modeling guidelines and process definition) and external standards (such as industry or regulatory standards).

- **Scribe:**

The scribe takes the meeting minutes and creates and updates the *action list*, the list of actions and issues to be addressed after the meeting. The action list will serve as the basis for subsequent reviews of the work product.

Disseminate materials for review

The materials should be in the hands of the reviewers no less than 2 nor more than 14 days prior to the review.

Schedule review

The room for the review – along with any special equipment required (e.g., projector, whiteboards, internet connection, computer, modeling tool availability, etc.) – is scheduled and invitations issued to the reviewers. This may also be done virtually.



Both Cameo Magic Draw (via **Collaborator**) and Rhapsody (via **Model Manager**) support online review of models, removing the need to generate work products for people who do not have access to the tools.

Reviewers independently inspect materials

The time to read the materials is *not* within the review meeting itself! The reviewers should come to the meeting with comments and issues they want to address. For complex work products, there may be value in the authors presenting a tutorial of the information to be reviewed. However, this should be a separate meeting, done prior to the review.

If the work product being reviewed is a model, then either the reviewers must have access to the relevant modeling tool, a web-based review environment (such as Cameo Collaborator), or reports must be generated for the purpose of the review that show the model content in an accessible format, such as Word, PowerPoint, or PDF.

Collectively discuss reviewer comments

Each issue or comment from a review should be heard and addressed by the product owner. If it is determined to be an issue, the issue is specified on the list of actions to be resolved.

DO NOT discuss solutions during this meeting. If someone has a solution they want to offer, have them do it outside the meeting with the work product owner.

Capture issues as action items

The primary outcome of the review is either an approval OR a set of action items to be resolved. Some action items might require modification to the work product, while others might require exploration or a trade study.

The action item list must be captured and will be used as the agenda for any subsequent reviews of the same work product set.

Resolve action items

As a result of the list of actions identified in the review, the product owner must modify the model or in some way address each of the issues. Then, if the changes are not trivial, a follow-up meeting is planned (repeat the recipe from the first step).

Plan review of action item resolutions

Unless the action items are considered trivial and not worth following up, a subsequent meeting should be scheduled specifically to address what was done to resolve the action items, and, typically, nothing more.

This subsequent review is performed in the same fashion as the original review.

Example

Let's consider doing a review of one of the use cases from the Pegasus system, *Emulate Front and Rear Gearing*, presented as an example in the *Functional analysis with state machines* recipe.

Establish the purpose of the view

The primary outcomes of use case analysis are:

- Ensure a good set of requirements.
- Define system interfaces necessary to support the related requirements.

The purpose of the review is to examine the work products created to perform that analysis and its outputs to ensure that the analysis was done adequately for the need and that the work products fulfill their purpose.

Prepare materials for review

The purpose of the review will be to look at the work products around:

- The use case and associated actors and their key properties (such as description):
 - Use case diagram showing actors and related use cases.
 - Traced requirements – use case diagram showing traced requirements.
- The use case sequence diagrams:
 - Specification scenarios.

- Animated scenarios.
- The use case execution context:
 - BDD/IBD showing the use case block and actor blocks.
- Use case block specification:
 - State and/or activity diagram for a use case or use case block.
 - State and/or activity diagrams for the actors or actor blocks.
 - Value and flow properties owned.
- Data schema:
 - BDD showing information known or relevant to the use case, including value types, dimensions (quantity kind), and units.
- The interfaces needed to support the use case:
 - BDD showing the interface blocks and their supported flow properties, operations, and event receptions.

Collectively, these diagrams and views form the work product to be reviewed. They are shown in *Figure 2.32* through *Figure 2.42* in *Chapter 2, System Specification*.

Assign reviewer roles

This step identifies the various roles. The review coordinator and the scribe are singletons – there is only one of those in the review. There must be at least one product owner (author) to present the materials and answer questions. The QA reviewer is optional but, if present, is usually a single person. There may be multiple SME reviewers, however. For the purpose of this example, let's assume 2 software, 1 mechanical, 2 electronic, 1 system engineer, and 1 marketer are invited as SMEs.

The identified personnel must be notified of their responsibilities.

Disseminate materials to reviews

In this case, some of the SMEs may not have access to the modeling tool and may not be skilled in SysML (I'm looking at you, marketing). For those reviewers, we can print out a report of the parts of the model relevant to their concern, such as a requirements table for the marketing SME. Other SMEs who have access and skill with the modeling tool can either be granted access to the model in a shared repository or even emailed the model.

Schedule review

Scheduling the meeting is generally easy but finding a time available for all attendees can be a challenge.

Reviewers independently inspect materials

This is a place where compliance can be spotty, especially when the workflow is introduced. In my experience, it is common for people who “meant to get around to” reviewing the materials to show up unprepared and expect to be walked through the materials they were expected to already examine. It is better to cancel a review if most people haven’t reviewed the materials and reschedule than to waste everyone’s time.

Here, the reviewers are expected to examine the use case, requirements, structural and behavioral views for syntactic conformance (in the case of the QA reviewer) and semantic correctness (in the case of the engineering SMEs). The marketer here is serving as a proxy for the customer (training athlete) and should look for the appropriateness and quality of the requirements.

Sidebar: Recording review comments in the model in Cameo.

There are many ways to make and record comments. Here is a method I like to use when reviewing Cameo models.

In the model, I create a **_Reviews** package to hold the review comments. In this package, I create nested packages, one per reviewer or review. In these nested packages, I add **Problem** elements identifying the issues found. Since **Problem** is a kind of **Comment**, I can anchor the problem to the element manifesting the concern, whether it is a diagram, block, operation, value property, state, action, etc. Then in the specific review package, I create a table summarizing my comments along with the **annotated elements**. This latter column can be added with the **Columns** tool for the generic table. Such a table is shown in *Figure 5.43*:

#	△ Box	Annotated Element
1	Model overview diagram doesn't describe model structure.	0 Pegasus Model Organization
2	Parts of the subsystem should be exposed on this diagram	Pegasus Context
3	The units of speed should be in RPM not radians per second	inout pedalSpeed : angular velocity[radian per second]
4	These operations do not have defining methods	<input type="radio"/> enable() <input type="radio"/> disable()
5	These use cases are missing meaningful descriptions	<input type="radio"/> Adjust Incline <input type="radio"/> Automatically handle crossover down shift <input type="radio"/> Configure Bike <input type="radio"/> Automatically handle crossover up shift <input type="radio"/> Control Resistance

Figure 5.43: Review comment table

Collectively discuss reviewer comments

The comments often vary widely in terms of usefulness. Syntactic comments about spelling and standards conformance are useful, to be sure. Most useful are comments about semantic content. This will not only improve the product but also improve understanding and agreement among the attendees.

I believe in what is known as “ego-less” reviews, but they can seem a little awkward at first. The idea is that comments only refer to the work product and never to the author. You would never say, “You made a mistake here” to the author, but rather, “This requirement seems incorrect” or “There is a missing value property here.” This is an attempt to avoid hurt feelings and focus on the technical aspects of the work product.

Here are some (hypothetical) comments and issues that might be raised by the reviewers:

- **QA reviewer:** The use case diagram doesn’t have a **mission statement**, as defined by the modeling guidelines standard.
- **QA reviewer:** The requirements standard says that each use case should specify the reliability, in MTBF, of the service delivery. There is no requirement to specify reliability traced to the use case.
- **Marketer:** Requirement **efarg17** says, “The default number of teeth for 12 cassette rings shall be 11, 13, 15, 17, 19, 21, 24, 28, 32, 36, 42, and 50. I think the gearing for the cassette should be from 12 to 28 instead.”
- **Marketer:** Requirement **efarg19** says that “the default starting gear should be the smallest possible gear.” I think that’s ok for the very first ride, but subsequent rides should start up in the last gear the rider used.
- **SME 1:** The use case block has a value property **cassette** of type **int**. It isn’t clear whether the value being held is the number of teeth of the currently selected cassette or which cassette is in use. Please either add a comment or rename the value property to make that clear.
- **SME 2:** The name of the state **ChangeGearing** for the use case block state machine is misleading. It is actually waiting for a gear change, not performing a gear change while in that state.

The product owner can respond to these comments, and a discussion of the comment ensures. Some action items may be resolved as not requiring action because of a corrected misunderstanding on the part of the reviewer or as inconsequential. The group may decide that other issues must be acted upon, and these end up as action items.

Capture issues as action items

The scribe records the action items and may add annotations depending on the urgency or criticality of the concerns. After the meetings, the action items are typically sent to all attendees so they can ensure their issues are properly recorded or understood.

Resolve action items

The product owner can then address the concerns. In this case, they can add the missing mission statement to the use case diagram, add reliability requirements, change the name of the value property to `cassetteIndex`, and the name of the state of `WaitingToChangeGear`.

Plan review of action item resolutions

Since the changes are minor, the group can decide that a subsequent review isn't needed.

Managing Model Work Items

There are many ways to manage to do items or work items in a backlog. This recipe is one specific way to simply and easily manage work items within a model. I've found this an effective way to track and enact changes in models on projects.

Purpose

The purpose of this recipe is to allow model authors to track, manage, and enact work items for a model.

Inputs and preconditions

The input for this recipe is an existing model that has work to be performed on its elements.

Outputs and postconditions

The primary output is an updated model that contains not only the changes made but also a record of changes, along with their priority and status.

How to do it

Figure 5.44 shows the workflow for this recipe.

There are two flows within the recipe. The first adds new work items and assigns their properties, while the second resolves the work items:

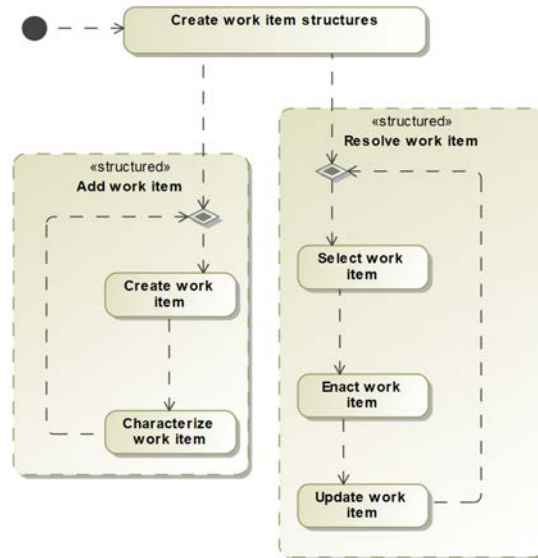


Figure 5.44: Manage work items

Create work item structures

This action adds the structures to represent, organize, and visualize the work items. We must define a work item as a type of element we can create and add relevant properties to it. The most obvious properties of such a type would be its status and its priority. The status of the work item identifies whether or not it is accepted or rejected, and whether it is waiting to be worked on, currently being worked on, or resolved. Additionally, a diagram or table should be defined so that we can easily see the work to be done.

Create work item

Once the work item types, structures, and visualization are defined, they can be used to characterize work items; as they are identified, they can be added to the list

Characterize work item

In this step, the properties of the work item are clarified. These properties are likely to include at least the work to be done, the status of the work item, and its priority.

Select work item

The other side to this recipe is performing the work. The first step is to select the work item to be resolved. Ideally, the work items are resolved on a higher-priority first basis.

Enact work item

This step is “simply” modifying the model to resolve the work item.

Update work item

Once the work has been performed, the status of the work item should be updated. Additionally, you may want to record how the work item was resolved.

Example

Create work item structures

In this example, we define a **Work Item Profile** that creates **Work Item** (a stereotype of **Comment**) and defines metadata properties **status**, **priority**, and **resolution**. The first two tags are typed by enumerations and the last is typed by **String**. It is possible to add other metadata, such as the type of work item (such as user story or spike), the author of the work item (for example, a particular quality assurance reviewer or subject matter expert), the date of the work item creation, and the date of the work item resolution:

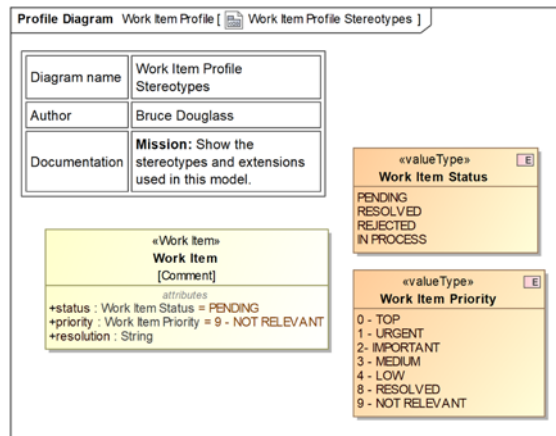


Figure 5.45: Work item profile

The reason for adding prefix numbers to the priority enumeration literals is so that you can sort the table of work items in a reasonable order, so that the **TOP** items appear before the **URGENT** items, which appear before the **IMPORTANT** work items, and so on.

Next, we must also define a place in our model to hold the work items and related elements. In this case, I create a package named **Work Items** to hold the work items to be added and to contain the visualization of those work items.

Finally, we must provide a visualization for the set of work items. Here, we will add a generic table of work items showing the target of the work items (if known) and the properties of the work item. Since **Work Item** is a stereotype of **Comment** – and we can anchor comments to model elements (including diagrams) – we can add **Annotated Elements** in the table as a column. Thus, as we add a work item, we can potentially anchor the work item to relevant model elements (if known), providing valuable guidance for its later resolution. Such a table, with a sample work item, is shown in *Figure 5.46*:

#	Body	Annotated Element	status	priority	resolution
1	This element is missing a useful and meaningful description.	Configuration App	PENDING	2-IMPORTANT	

Figure 5.46: Work item table

Create work item

Now that the structures are in place, we can start adding work items. This might come from inspection and review (see the previous recipe *Effective walkthroughs and reviews*) or some other source such as user stories, spikes, defect reports, or technical work items from the project backlog.

Characterize work item

Because **Work Item** is a stereotype of **Comment**, you may add anchors to elements, although when the work item is created, the elements to be modified may not be obvious or known. The **status** should be initially set to **PENDING** and the priority of the work item should be specified. The easiest way to add an anchor is to place the work item on a relevant diagram and add the anchor. It is simple to create diagrams in the **Work Items** package and drag the relevant elements onto that diagram and add the anchors there. This is my preferred approach so the presence of the work items doesn't "pollute" the other diagrams in the model. In Cameo, you can drag diagrams onto other diagrams, making it easy to add anchors to the model element to be modified in a diagram.

Figure 5.47 shows an example:

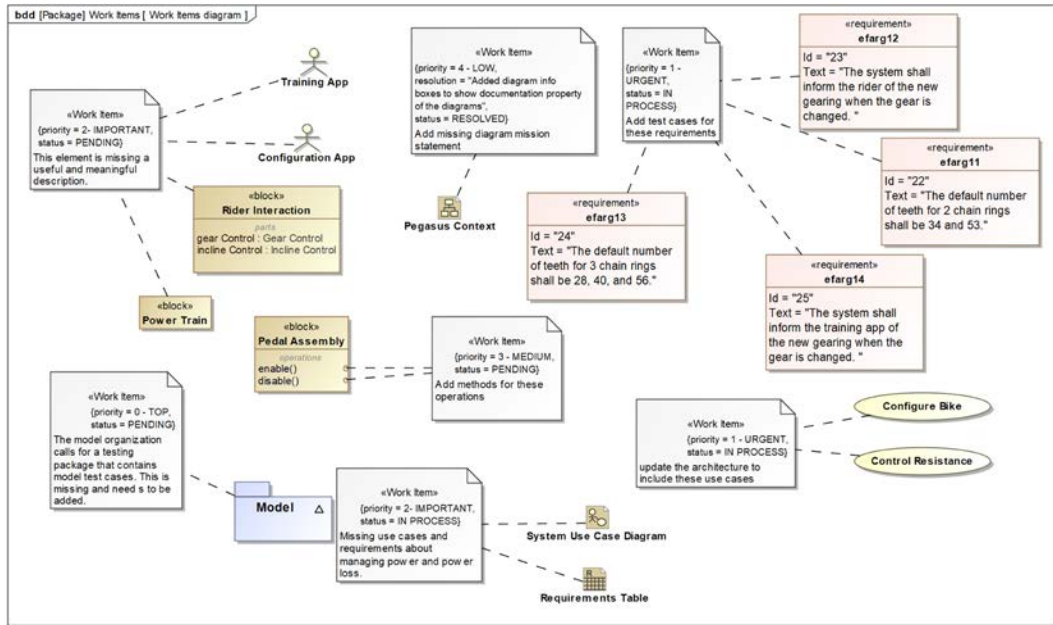


Figure 5.47: Adding anchors to model elements

The desired visualization for the work items is the work item table. Figure 5.48 shows an example, sorted in priority order:

#	Body	Annotated Element	status	priority	resolution
1	The model organization calls for a testing package that contains model test cases. This is missing and needs to be added.	Model	PENDING	0 - TOP	
2	Add test cases for these requirements	efarg11 efarg13 efarg14 efarg12	IN PROCESS	1 - URGENT	
3	update the architecture to include these use cases	Configure Bike Control Resistance	IN PROCESS	1 - URGENT	
4	Missing use cases and requirements about managing power and power loss.	System Use Case Diagram Requirements Table	IN PROCESS	2 - IMPORTANT	
5	This element is missing a useful and meaningful description.	Configuration App Training App Rider Interaction Power Train	PENDING	2 - IMPORTANT	
6	Add methods for these operations	enable() disable()	PENDING	3 - MEDIUM	
7	Add missing diagram mission statement	Pegasus Context	RESOLVED	8 - RESOLVED	Added diagram info boxes to show documentation property of the diagrams
8	The architecture acceptance criteria are not stated		REJECTED	9 - NOT RELEVANT	This is handled in the DOORS module.

Figure 5.48: Working work item table

Select work item

To resolve the work item, we must first select it.

Enact work item

This step involves updating the model to enact the required changes. The first step is to change the work item status to **IN PROCESS**.

Update work item

Once completed, the work item can be updated. At the very least, the **status** property is changed to **RESOLVED**. You might want to also update the priority to **8 – RESOLVED** so that it isn't mixed in with the work items yet to be resolved.

Test Driven Modeling

For the last recipe in the book, I'd like to present a recipe that is central to my view of the integration of modeling and agile methods. The archetypal workflow in software agile methods is **test-driven development (TDD)**, in which you do the following:

```
Loop
  Write a test case
  Write a bit of code to meet that test case
  Apply test case
  If (defect) fix defect
Until done
```

It's an appealing story, to be sure. Each loop shouldn't take more than a few minutes. This is a key means in agile methods to develop high-quality code: test incrementally throughout the coding process.

TDD aligns with this Law of Douglass:



The best way not to have defects in your system is to not put any defects into your system.

Law of Douglass #30

See <https://www.bruce-douglass.com/geekosphere> for further information.

This points out that it is far easier to develop high-quality systems if you avoid entering defects rather than putting defects in and trying to identify, isolate, and correct them sometime in the future. TDD is a way to do that.

The same kind of approach is possible with respect to developing models, particularly executable models, an approach I call **Test-Driven Modeling (TDM)**. These models needn't be limited to design models; they can also be models of architecture, requirements, or any kind of computational model. Such models can be developed in small, incremental steps and verified as the model evolves.

When it comes to modeling, this is *not* what people typically do. Most people build huge monolithic models before deciding to verify them. I remember one consulting customer who had a state machine with 800+ states. They had put it together and then attempted to “beat it with the testing stick” until it worked. They gave up after several months without ever getting it to even compile, let alone run properly. Then they called me.

The first thing I did was discard the expensive-yet-unworkable state machine and developed an equivalent using this TDM recipe. Each small iteration, known as a **nanocycle** in the *Harmony aMBSE* process (see *Agile Systems Engineering* by Bruce Douglass (Morgan Kaufmann, 2016) for further information) is executed anywhere from 30 to 60 minutes. In less than a week, the completed state machine was compiling, running, and working. It was then possible to uncover a number of subtle, yet important, specification defects from their source material.

Like I said, TDM is not the way most people create models. Let's change that.

Purpose

The purpose of this recipe is to quickly develop a demonstrably correct model by frequently executing and running the model.

Inputs and preconditions

The input is a well-defined modeling objective that can be rendered with executable modeling constructs.

Outputs and postconditions

The output is a well-constructed, executable, and verified model.

How to do it

Figure 5.49 shows the basic flow. I would take care to note that I don't really care whether the test case is created just prior to the “Model a bit” step or just after; it is essentially at the same time. Classic TDD has the test case definition first, but it doesn't really matter to me.

The primary point is that the test case isn't defined weeks to months after the design work is done but is done continuously as the work product is developed:

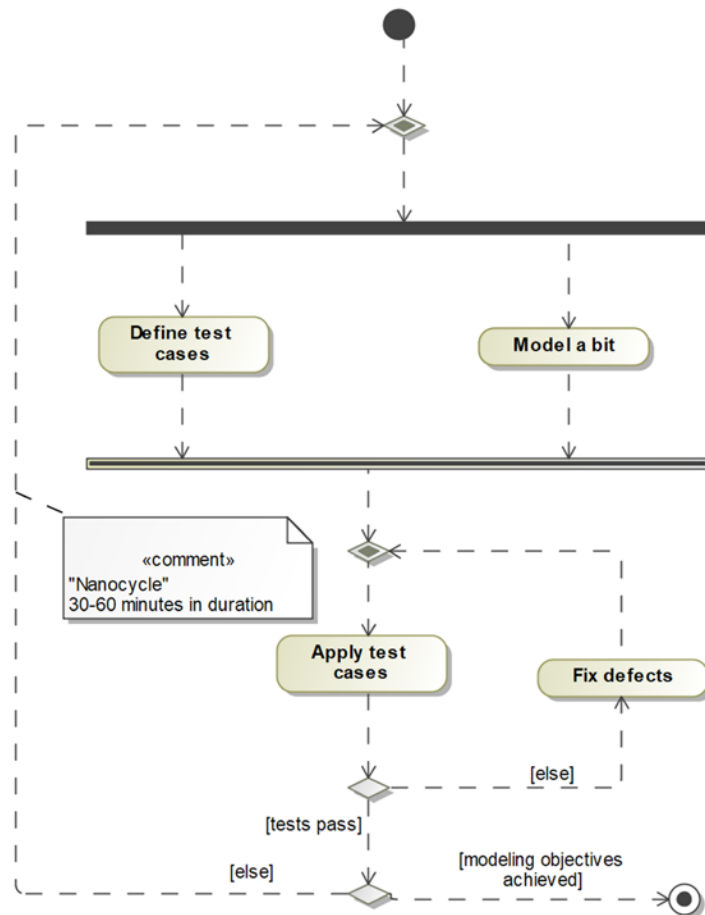


Figure 5.49: TDM

Define the test case

This step can come immediately before or after the *Model a bit* step, but logically they are done at about the same time. The test case defines a **Minimal Testable Feature (MTF)**, sometimes known as a **Minimal Verifiable Feature (MVF)**, that is to be modeled in this particular nanocycle. The MTF in this context is a small evolution of the model from its current state, *enroute* to the complete model that satisfies the initial modeling objective. The tendency is to make the nanocycles too large, so keep them short. The properties of the MTF to strive for are:

- The feature can be modeled in a few (<30) minutes.
- The feature tests differently from the previous model's condition.
- The feature contributes to the modeling objective that the sequence of nanocycles is intended to achieve.

For example, suppose the modeling objective is to construct an executable use case. Each nanocycle would add one or more requirements to the use case model, such that the state machine adds a few transitions, actions, or states. A state machine with 30 states and 40 transitions might be done as a series of 10–15 nanocycle iterations, each of which has a small MTF evolution.

Model a bit

In this step, the model is evolved by adding a small number of features: a value property or two, perhaps a couple of transitions, a state, or a small set of other model elements.

Apply the test case

At this point, you have a (small) model evolution and a (small) test case. In this step, you apply the test case to ensure that it is modeled correctly.

Fix defects

If the model fails the test, it is fixed in this step.

The nanocycles repeat until the initial modeling objective is satisfied.

Example

I want to introduce my favorite example for TDM. I've used this example in numerous courses and found it to be instructive and accessible to my students. I've even done the experiment: I told some groups to develop the model without giving them guidance (and they all tried to do it in a single step) or gave them an explicit instruction to do it in several separate steps, with a short description of the MTF increments. The results are that, in the time frame allotted, 100% of the unguided groups failed while 80% of the nanocycle groups succeeded.

The problem

Consider an intersection **Traffic Light Controller (TLC)**. The TLC controls traffic (via lights) on a primary road and a secondary road. The roads contain both through-lane traffic and left-turn lane traffic. In addition, the TLC also controls pedestrian traffic along those roads as well (again, via lights). The basic control rules are:

- In the absence of both pedestrians and left-turn cars, the through lights operate entirely based on time:
 - Turn lane lights remain red; pedestrian lights remain DON'T WALK.
 - All lights are red for RED TIME.
 - The through lanes (both direction for a given road) go green for GREEN TIME, while cross traffic lights are red while cross traffic remains stopped.
 - The through lanes (again, in both directions) go yellow for YELLOW TIME.
 - The through lanes all turn red and the cross traffic gets to go next, using the same flow.
 - Repeat forever.
- If cars are present in the left turn lane for a given road, the behavior is slightly different:
 - When a road is ready to set through traffic green but there is a car waiting in that road's left turn lane, the turn lanes (both directions on the same road) go first, and the through lane remains red:
 - Turn lanes go green for GREEN TIME
 - Turn lanes go yellow for YELLOW TIME
 - Turn lanes go red for RED TIME
 - Now the through lanes can continue normally
- If pedestrians are waiting to cross a road (that is, they want to cross the cross-traffic road), then the behavior is modified:
 - When the through light goes green, the pedestrian light displays "Walk" for WALK TIME.

- The pedestrian light then displays flashing “Don’t Walk” for RUN TIME.
- The pedestrian light then displays a solid “Don’t Walk” and the through light now goes yellow for YELLOW TIME.
- The through lane proceeds normally.

I like this example because it is easy to describe, it is conceptually simple, and it is very familiar.

Since this is a short, iterative cycle, we’ll just run the cycle multiple times in the example:

- MTF 0: Single through traffic light
- MTF 1: Primary and secondary through lights
- MTF 2: Add turns lights
- MTF 3: Add pedestrians

Define test case: MTF 0: Episode III: The Revenge of the Through Light

Let’s define the first MTF step.

Test case: Single light should go through the colors in the right sequence with the right timing – red for RED TIME, green for GREEN TIME, and yellow for YELLOW TIME.

Model a bit: MTF 0: Episode III: The Revenge of the Through Light

Evolution: Add a single traffic light that cycles through its colors.

This model is pretty simple. It consists of two blocks: **Traffic Light Control System** (which will also serve as our tester) and **Traffic Lane**, with a single, directed composition from the former to the latter. We need to define the timings – I used value properties in the **Traffic Lane** block with the **readOnly** property set. I also added an enumerated type **COLOR_TYPE** to define the different colors.

A `setcolor(color: COLOR_TYPE)` just prints out the value of the color to standard output for the purpose of this model. This simple model is shown in *Figure 5.50*:

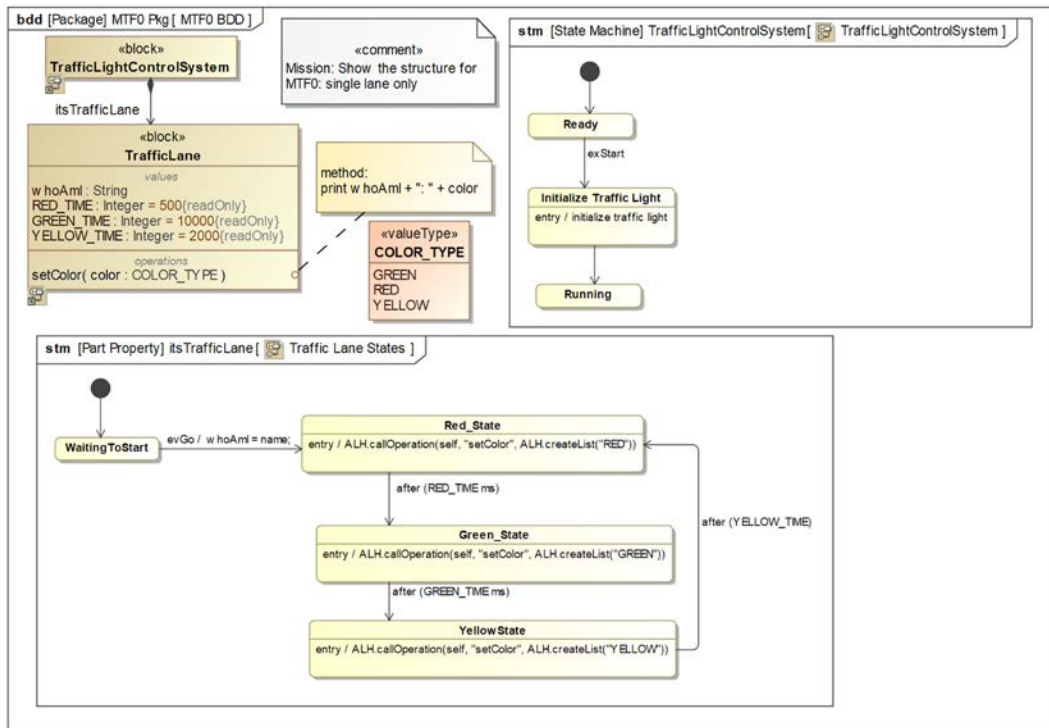


Figure 5.50: Structure and behavior for MTF 0

Apply test case: MTF 0: Episode III: The Revenge of the Through Light

Now let's run the model. In Cameo, right-click the **Traffic Light Control System** block and select **Simulation > Run**. When we run, we manually insert the `exStart` event to the **Traffic Light Control System** to kick it off, and we can view the execution outcomes in *Figure 5.51*:

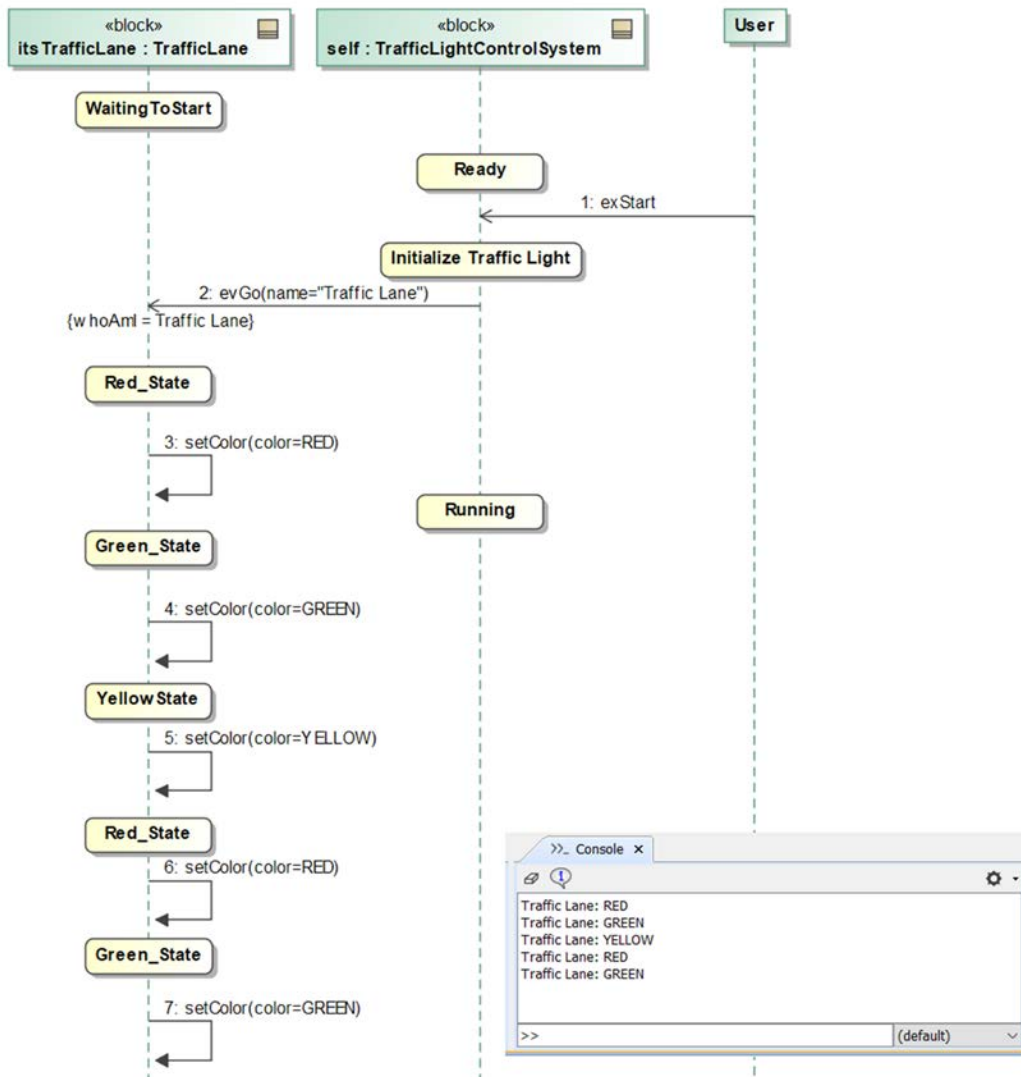


Figure 5.51: Running MTF 0

We've now completed our first nanocycle. It was very short – and it didn't do much – but we quickly got it to work.

Define test case: MTF 1: Episode IV: A New Hope (For the Secondary Through Lane)

Let's define the next MTF step.

Test case: Both roads should cycle through the lights, with cross traffic held red while the current lane cycles through.

Model a bit: MTF 1: Episode IV: A New Hope (For the Secondary Through Lane)

Evolution: Add a second composition to the through lane to add the secondary road; the system now contains two traffic light instances. Modify the state machine of the traffic light so that the two instances of **Traffic Lane** can properly coordinate. In this case, we add proxy ports **pOutThru** and **pInThru**, both typed by the interface block **ibLane**. One of these ports is conjugated so that it can send and receive the same event, though via different ports. See *Figure 5.52*:

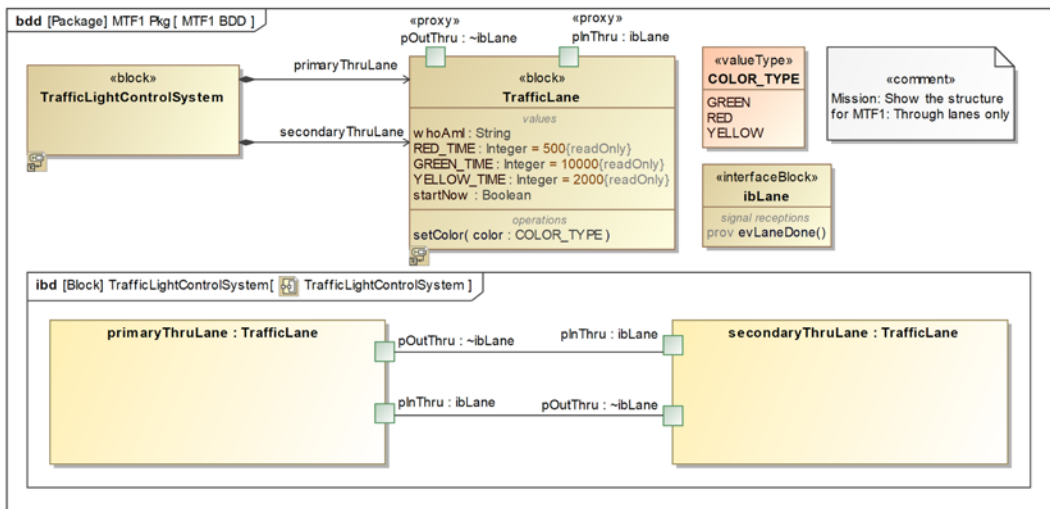


Figure 5.52: Structure for MTF 1

In the figure, we take advantage of Cameo's ability to show diagrams on other diagrams to show the internal block diagram contents on the block definition diagram.

The state machine for the **Traffic Light Control System** is exactly the same except that it now sends the **evGo** event to both instances and the signal has an additional attribute. The first (Boolean) value tells the instance whether it should go first or wait. The second gives the instance its identity for the **whoAmI** value property. *Figure 5.53* shows the state machine for **Traffic Light Control System** along with the behaviors to send the **evGo** signal to initialize the two traffic lights. *Figure 5.54* shows the **Traffic Lane** behavior:

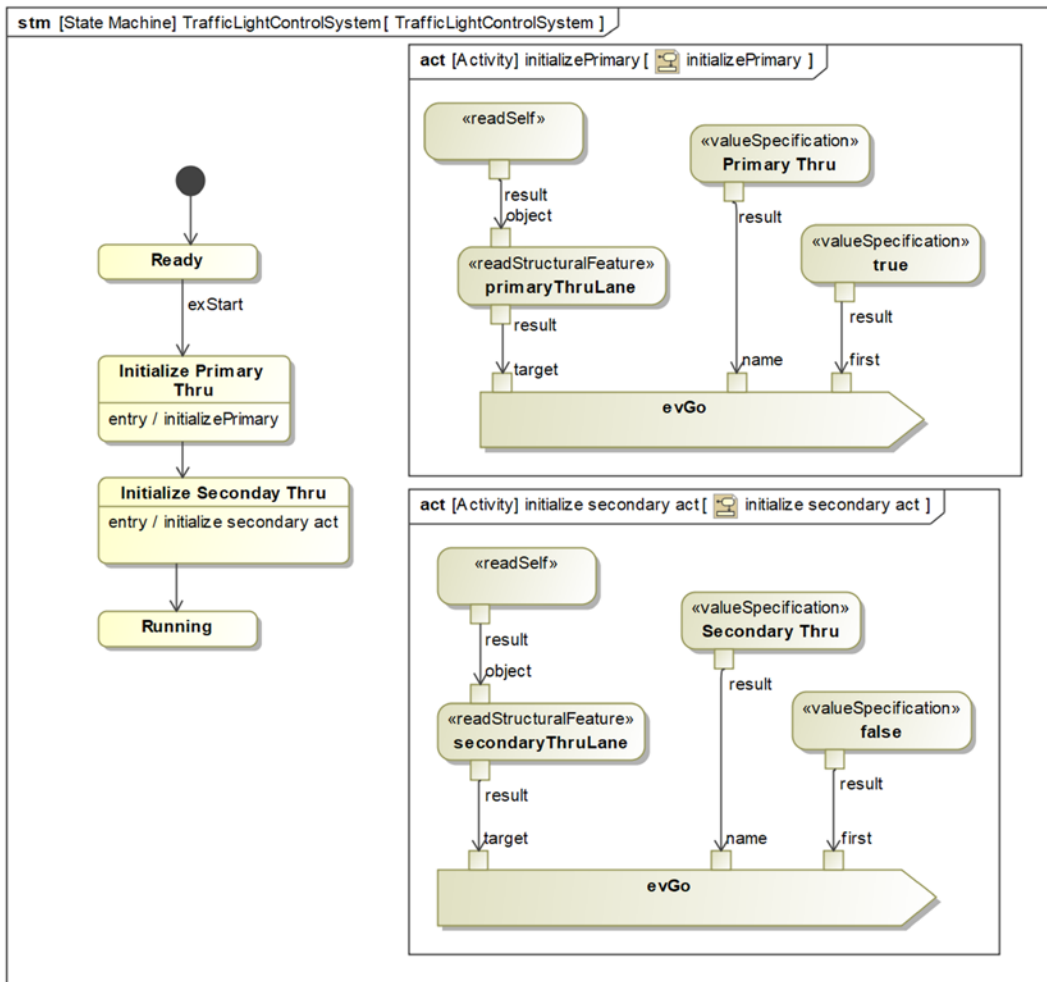


Figure 5.53: Traffic Light Control System state machine for MTF 1

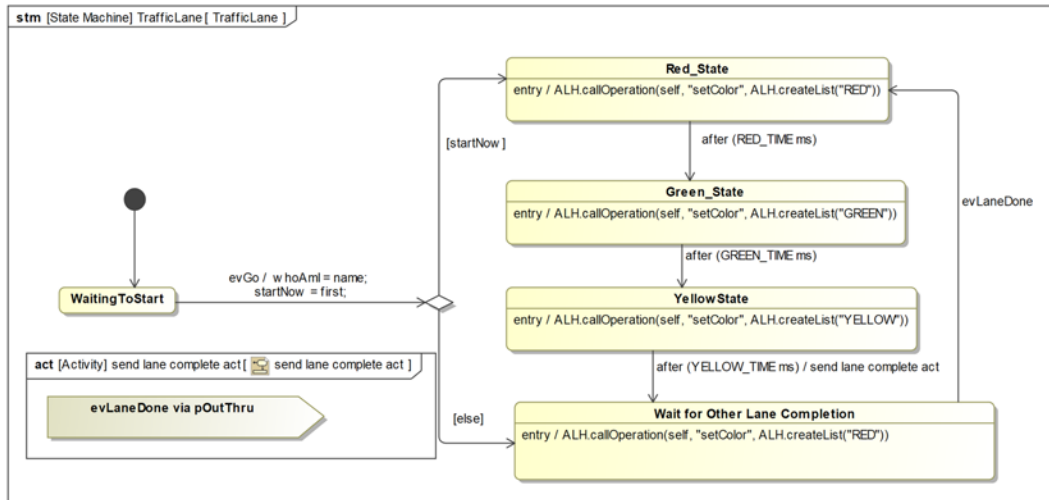


Figure 5.54: Traffic Lane state machine for MTF 1

Apply test case: MTF 1: Episode IV: A New Hope (For the Secondary Through Lane)

Now the model is getting interesting. Is it right? Let's run the test case and see.

Figure 5.55 shows the outcomes, both in the animated sequence diagram and the output window.

The scenario is a bit small to read, but you can see in the output window that it operates as expected:

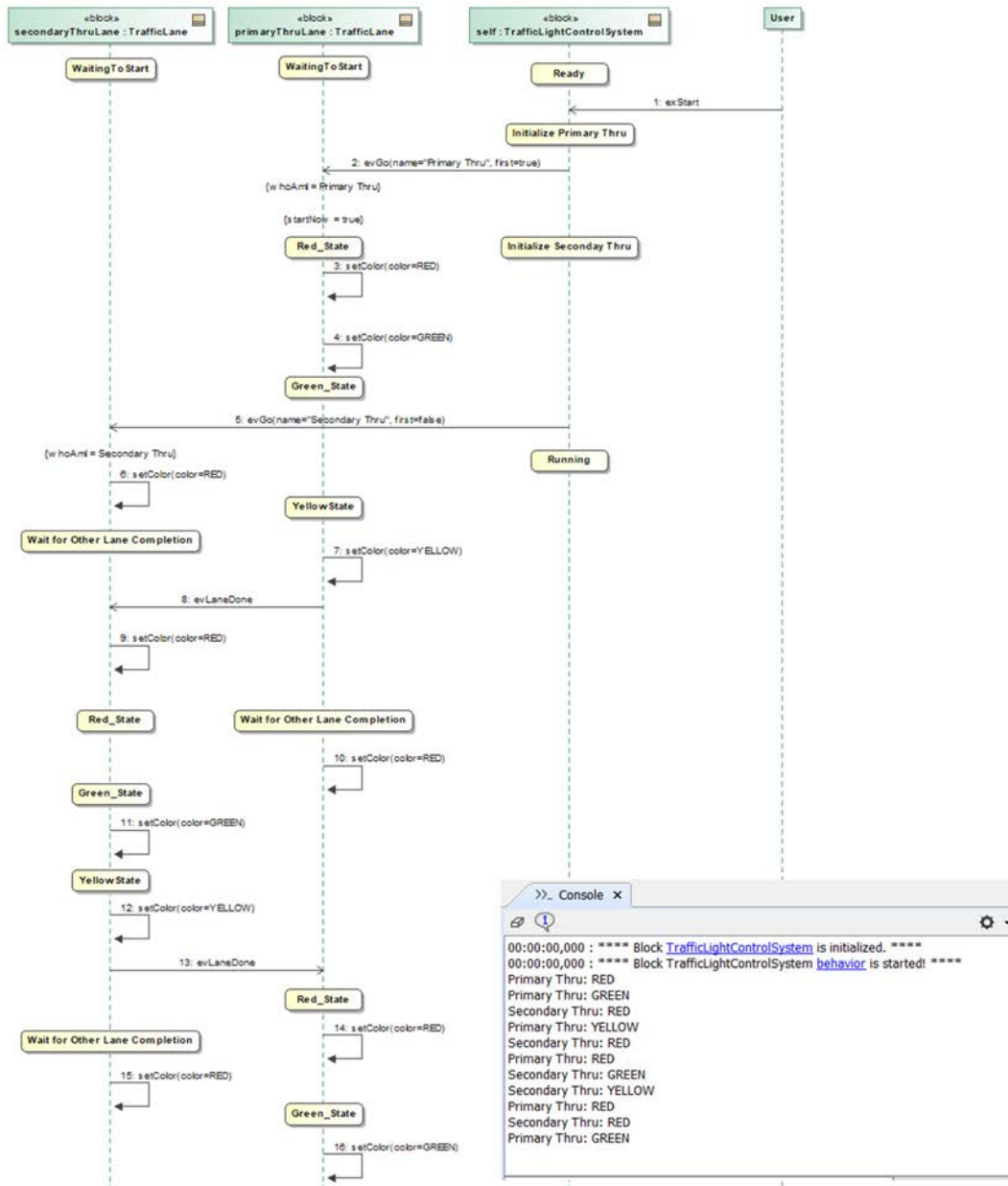


Figure 5.55: Running MTF 1

Fix Defects: MTF 1: Episode IV: A New Hope (For the Secondary Through Lane)

There are no defects to fix in this nanocycle.

Define test case: MTF 2: Episode V: The Turn Lane Strikes Back

Test case: When a car arrives on a road *regardless of when in the cycle that occurs*, the system properly allows turn lanes to cycle through, followed by the through light on the same road.

Test case: In subsequent cycles, the system “forgets” the turn lane was previously activated.

Model a bit: MTF 2: Episode V: The Turn Lane Strikes Back

Evolution: Add turn lanes and update state machines to manage them.

We must add two more lights to the system to support the left turn lanes, and make them composite parts of the **Traffic Light Control System**. In this case, the through lights and the turn lights are actually different types because while the through lights must coordinate with the cross-traffic through light, the turn lane coordinates in a different way with its corresponding road through light. We’ll make the **Thru Lane** and **Turn Lane** lights different subclasses of **Traffic Lane**, and remove the state behavior from the **Traffic Lane** base class. The structural model is shown in *Figure 5.56*:

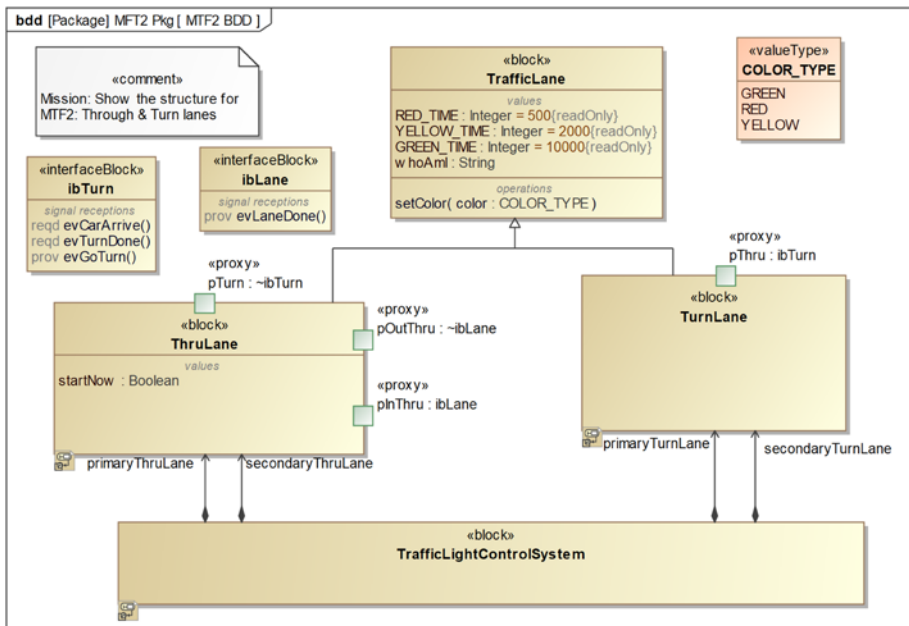


Figure 5.56: Structure for MTF 2

The **Traffic Light Control System** internal block diagram (Figure 5.57) shows the expected connections among the instances:

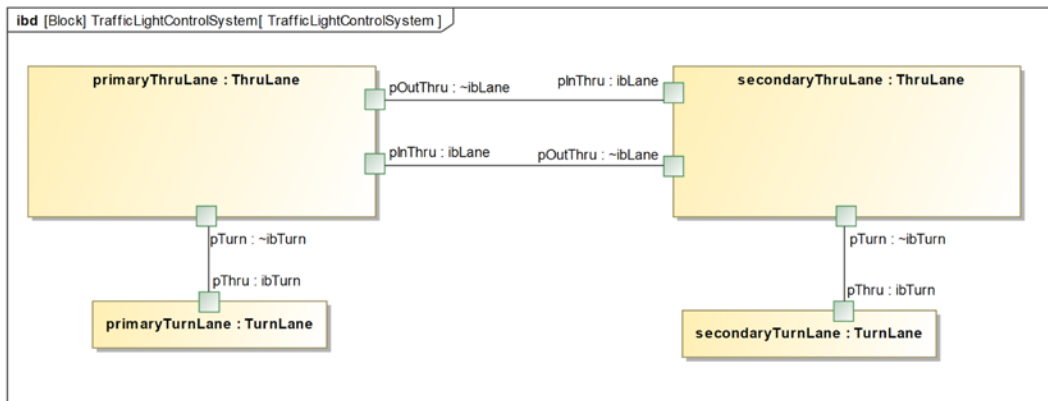


Figure 5.57: MTF 2 IBD

The state machine for **Traffic Light Control System** is modified so that it starts the turn lane lights in addition to the through lane lights. See Figure 5.58:

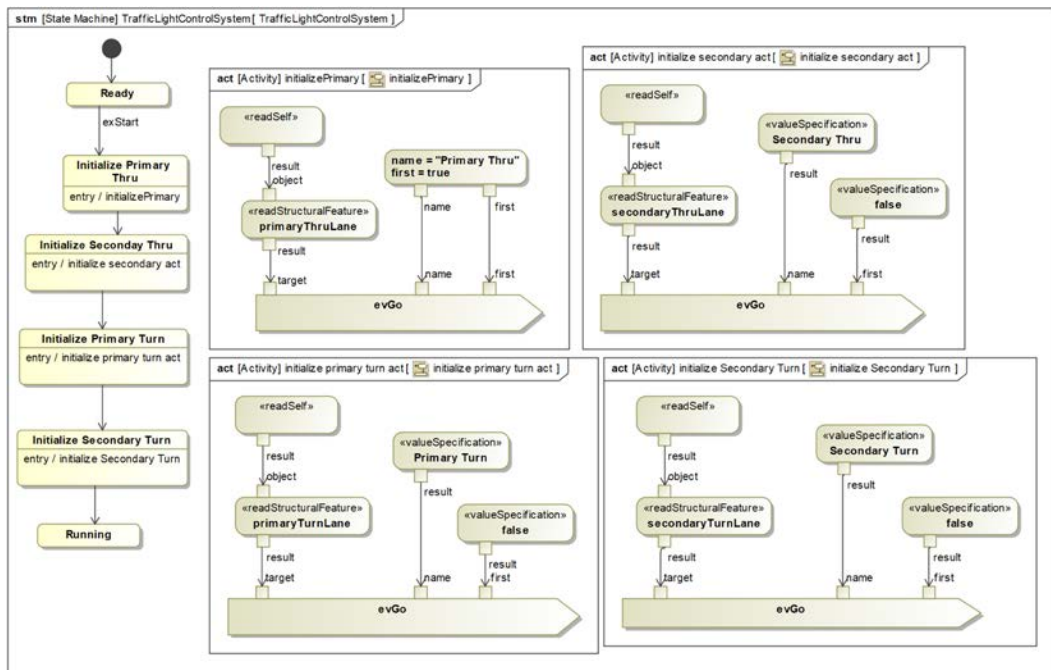


Figure 5.58: State machine for traffic light system MTF 2

More interesting are the modifications to the **Thru Lane** and **Turn Lane** lights. The state machine in *Figure 5.59* is for the **Thru Lane** and *Figure 5.60* is for the **Turn Lane**. The send actions on the state machines show the coordination of the through lanes as well as between the through lane and its connected turn lane:

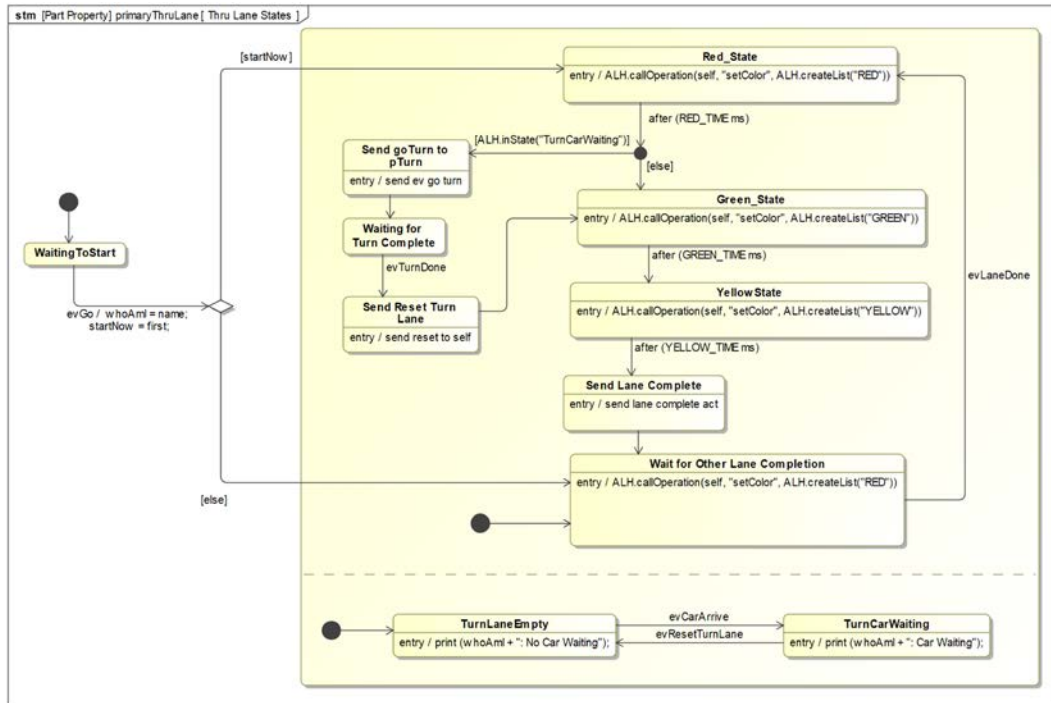


Figure 5.59: MTF 2 through lane states

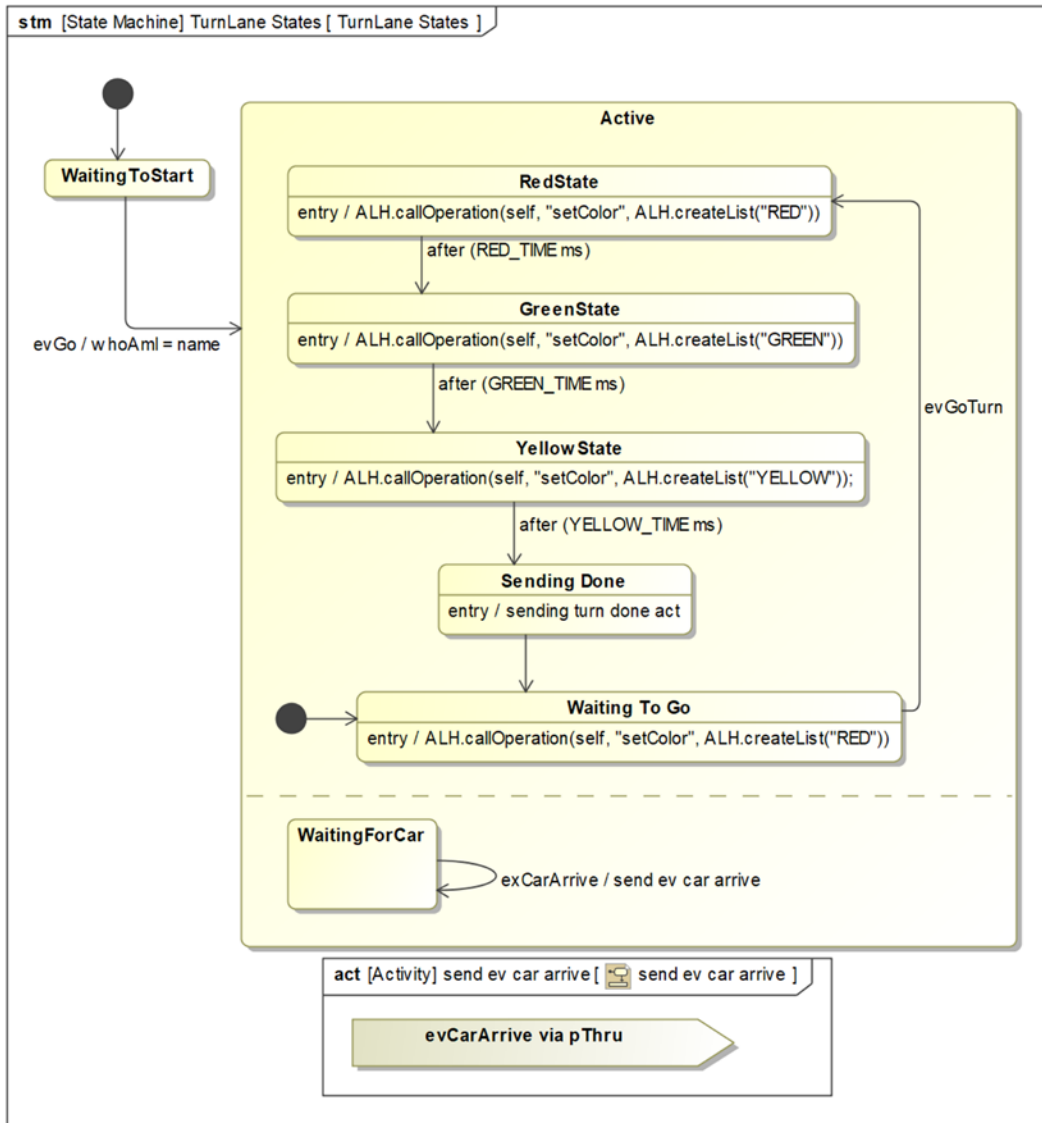


Figure 5.60: MTF 2 turn lane states

Apply test case: MTF 2: Episode V: The Turn Lane Strikes Back

Running the test case results in long sequence diagrams, so we will simply look at the output console window to verify the correctness of the execution. In this case, we'll start the system and while the primary through lane is green, we will have a car arrive for the secondary turn lane. The expected behavior is that the primary light will go green, yellow, and then red. Then, because there is a car waiting in the secondary turn lane, we expect to see the secondary turn lane go green, yellow, and then red. Following that, the secondary through lane should go green, yellow, and red before the primary through lane again goes green.

When running the state machine, we note that we cannot send the event **exCarArrive** at the right time because of a **race condition**. Because of the order in which the behavior unfolds, the primary through lane – being initialized first – starts cycling through its behavior even while the **Traffic Light Control System** is initializing the other parts. This is a classic race condition.

To fix this problem, we will change the behavior of the state machines:

- **evGo** will no longer have a **first** parameter. Both through lanes will start in the **Wait for Other Lane Completion** state.
- At the end of initialization, the **Traffic Light Control System** will send the **evLaneDone** signal to the **Primary Thru Lane** part to kick things off.

The updated behavior for the **Traffic Light Control System** is shown in *Figure 5.61*:

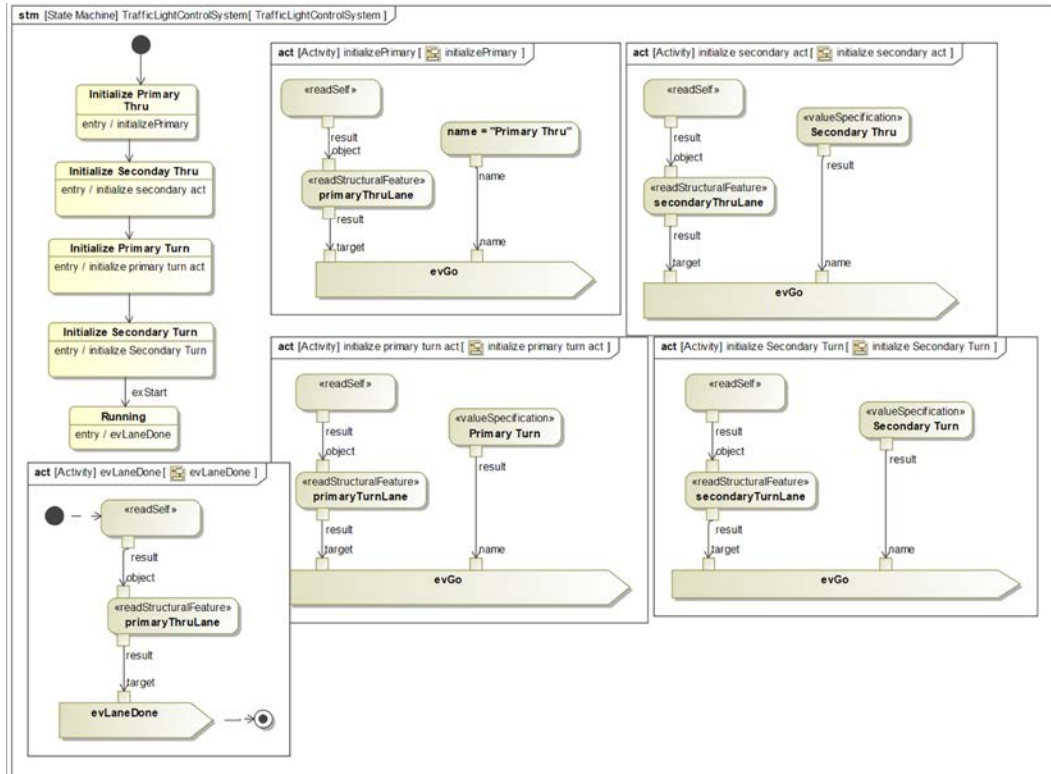


Figure 5.61: Traffic light control system updated behavior

The change to the **Thru Lane** is simpler. See *Figure 5.62*:

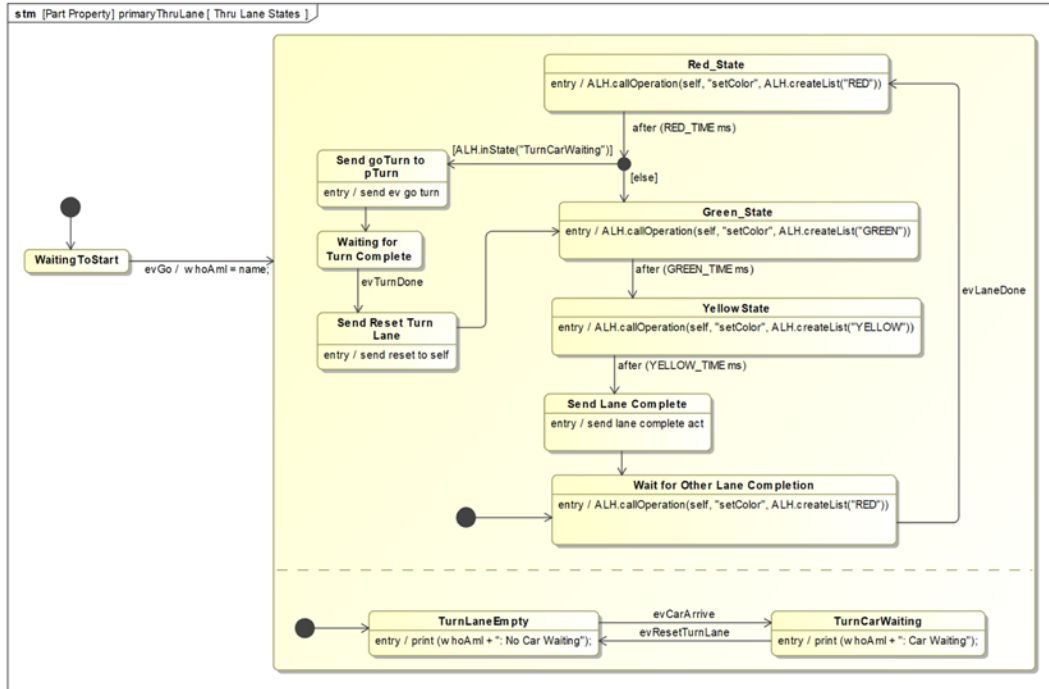


Figure 5.62: Updated thru lane behavior

Now we can run the model and while the **Traffic Light Control System** waits for the **exStart** event, we can send the **exCarArrive** signal to the **secondary turn lane** part to simulate the arrival of a car in that lane.

In Cameo, when the context has multiple parts of the same type, it isn't obvious how to send the signal to the correct part. The trick is to look in the **Variables** section of the Simulation Toolkit and find the address of the part you want, match that address to a running behavior in the **Sessions** window, and then use the pull-down trigger list to send the event.

See Figure 5.63:

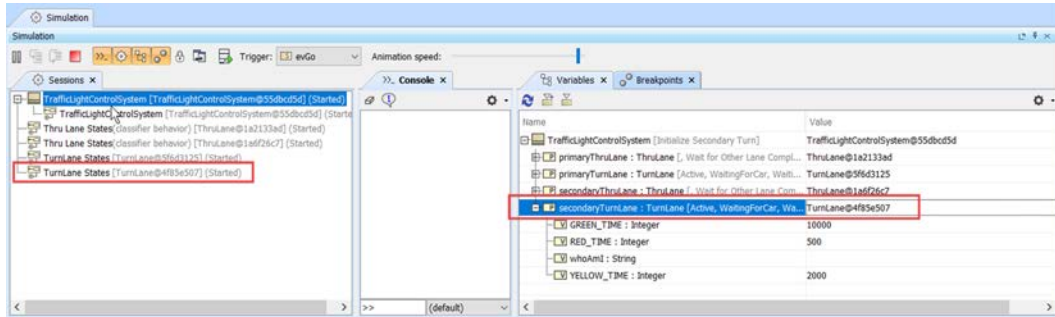


Figure 5.63: Sending the signal to the correct part

Figure 5.64 shows the console window output for the updated model:

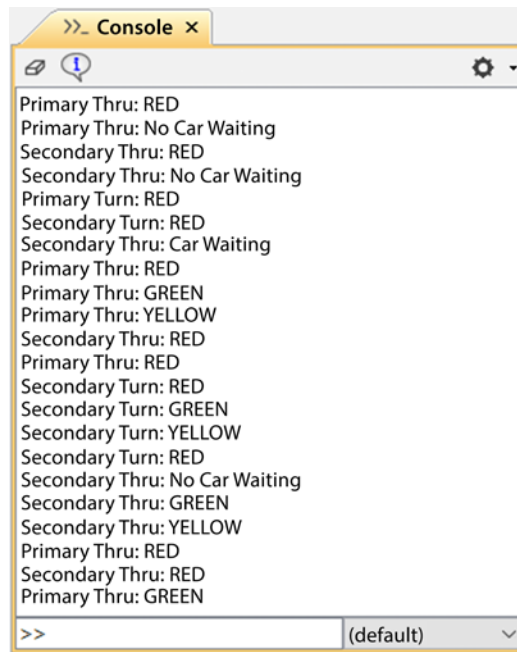


Figure 5.64: Running MTF 2

Fix Defects: MTF 2: Episode V: The Turn Lane Strikes Back

The race condition has been identified and resolved.

Define test case: MTF 3: Episode VI: Return of the Pedestrian

Test case: When a pedestrian arrives on the secondary road and the primary through light is green, the pedestrian gets the walk light at the same time that the secondary through lane gets a green light.

Test case: When a pedestrian arrives on the primary road and the primary through light is currently green, the pedestrian gets a walk light the next time the primary through lane gets a green light.

Model a bit MTF 3: Episode VI: Return of the Pedestrian

Evolutions: Add pedestrians and update state behavior to manage them.

This last nanocycle adds the pedestrian crosswalks, and updates the behavior of the **Traffic Light Control System** and the through lanes to properly interact with them. The BDD shows the evolving structure (Figure 5.65) and the **Traffic Light Control System** IBD shows the connected design (Figure 5.66):

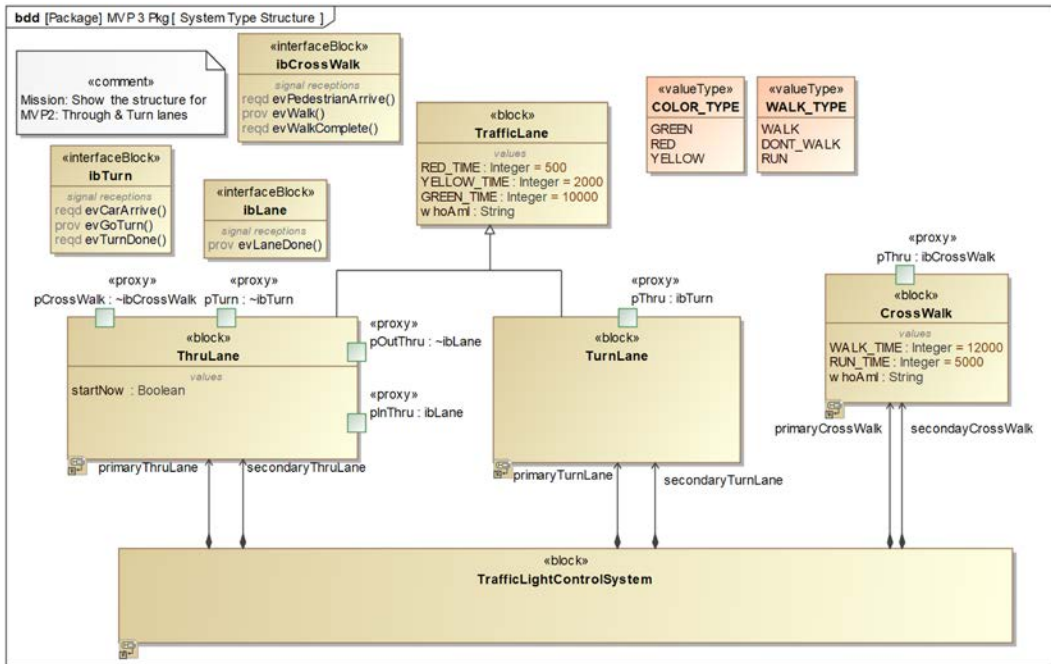


Figure 5.65: MTF 3 structure

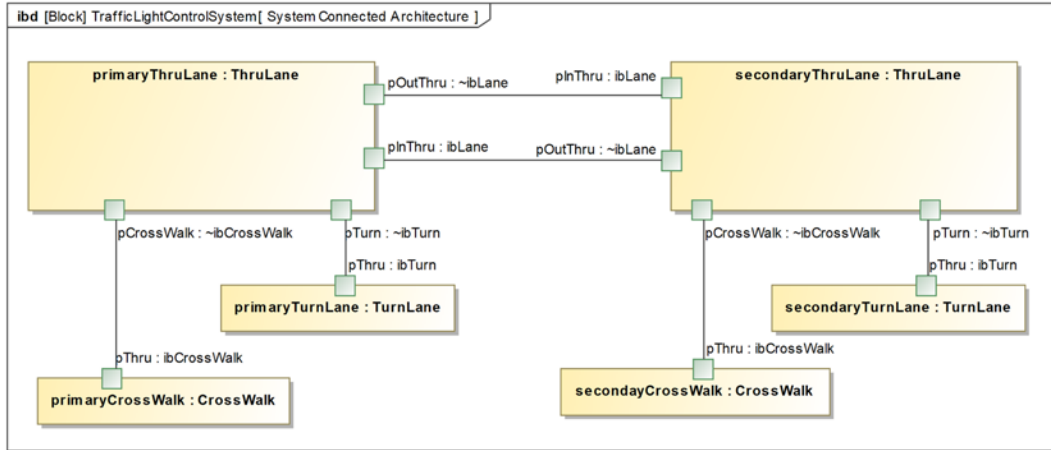


Figure 5.66: Traffic light system IBD for MTF 3

The **Traffic Light Control System** has a minor change to its state machine to initialize and start the **Cross Walk** instances (Figure 5.67):

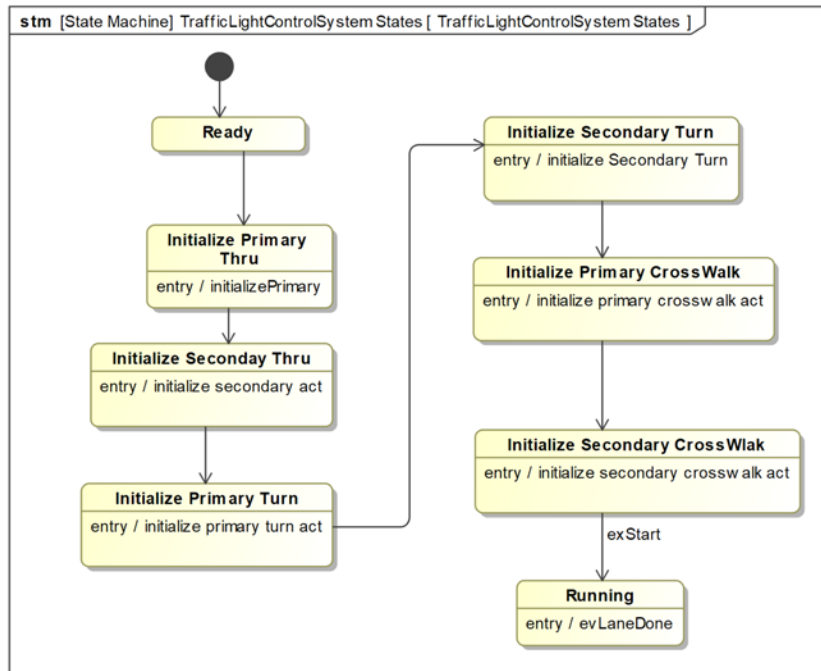


Figure 5.67: Traffic light system state machine for MV3

The state machine for the **Thru Lanes** adds an AND-state to keep track of whether pedestrians are waiting and a few states to handle the behavior when a pedestrian is crossing (Figure 5.68). If you compare this state machine with the **Thru Lane** state machine from MTF 2 (Figure 5.59), you can see that the updated state machine is a fairly simple evolution of its previous condition:

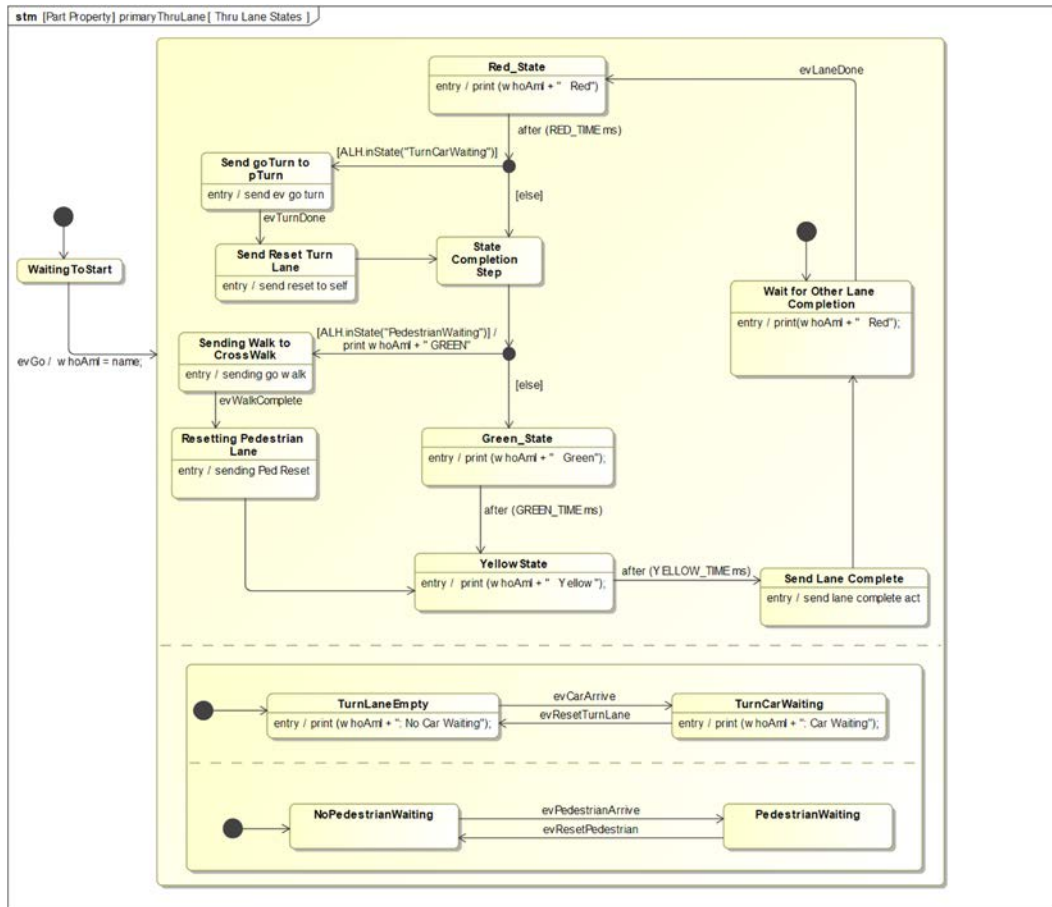


Figure 5.68: Thru lane state machine for MTF 3

Lastly, the state machine for the **Cross Walk** is straightforward, as can be seen in Figure 5.69.

Note that the **Turn Lane** state machine didn't change from MTF 2:

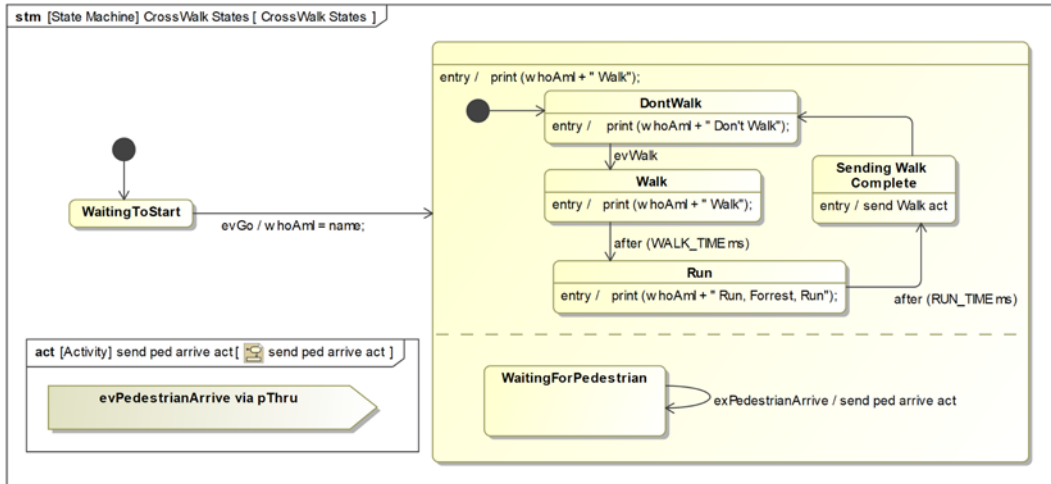


Figure 5.69: Cross walk state machine for MTF 3

Apply test case MTF 3: Episode VI: Return of the Pedestrian

The model is getting a bit complex to rely solely on the animated sequence diagrams for testing, so we elaborate the UI diagram by adding controls related to the pedestrian traffic and adding a simulation configuration to use as the user interface to monitor and control the simulation.

This MTF has two test cases; the output window in *Figure 5.70* shows the outcome for test case 1:

```

00:00:00,000 : ***** Block TrafficLightControlSystem is initialized. *****
00:00:00,000 : ***** Block TrafficLightControlSystem behavior is started! *****
Primary Thru Red
Primary Thru : No Car Waiting
Secondary Thru Red
Secondary Thru: No Car Waiting
Primary Turn Red
Secondary Turn Red
Primary Crosswalk Walk
Primary Crosswalk Don't Walk
Secondary Crosswalk Walk
Secondary Crosswalk Don't Walk
Primary Thru Red
Primary Thru Green
Primary Thru Yellow
Secondary Thru Red
Primary Thru Red
Secondary Thru GREEN
Secondary Crosswalk Walk
Secondary Crosswalk Run, Forrest, Run
Secondary Crosswalk Don't Walk
Secondary Thru Yellow
Primary Thru Red
Secondary Thru Red
Primary Thru Green
  
```

Figure 5.70: Output from MTF 3 test case 1 run

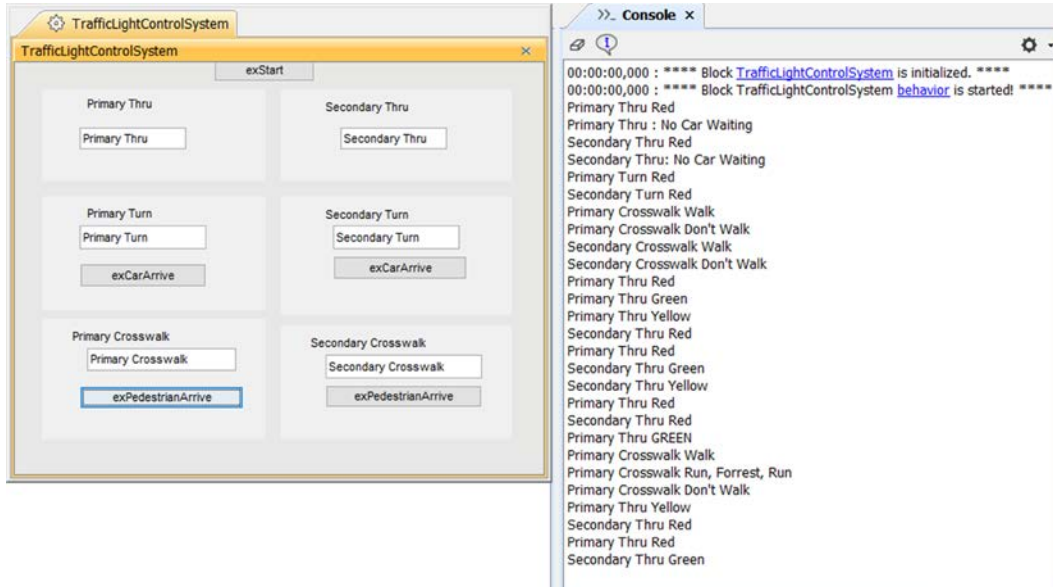


Figure 5.71: Output from MTF 3 test case 2 run

Fix defects MTF 3: Episode VI: Return of the Pedestrian

No defects were found. MTF 3 runs as expected.

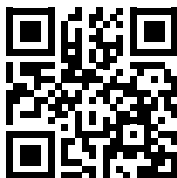
This example illustrates that it was pretty simple to construct a (somewhat complex) model easily by developing it in small iterations and verifying its correctness *so far* before moving on.

Now *that's* pod racing!... Er, TDM.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>



Appendix A: The Pegasus Bike Trainer

Overview

The Pegasus is an indoor cycling training system to support both casual and professional athletes in their cycling training. It has a large set of features to make indoor training effective and enjoyable. It simulates road feel, including automated control of power-on-pedal requirements and bike incline. It is expected that the bike will not be used in isolation, but will be used with an iOS, Android, or Windows device that receives, displays, records, and analyzes data from exercise sessions. Support is provided through standard low-power Bluetooth, ANT+, and ANT FEC interfaces.

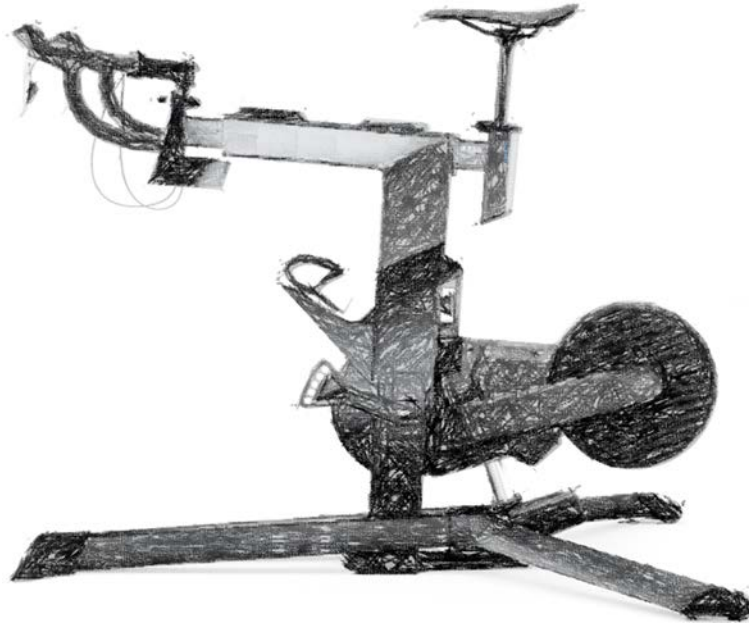


Figure A.1: Pegasus indoor training bike

Pegasus High-Level Features

This section describes a number of the key features and benefits of the Pegasus Bike Trainer.

Highly customizable bike fit

It fits users from 5 ft tall to 6'4 with a rider weight of up to 300 lbs. It has an adjustable crank length from 160 mm to 180 mm.

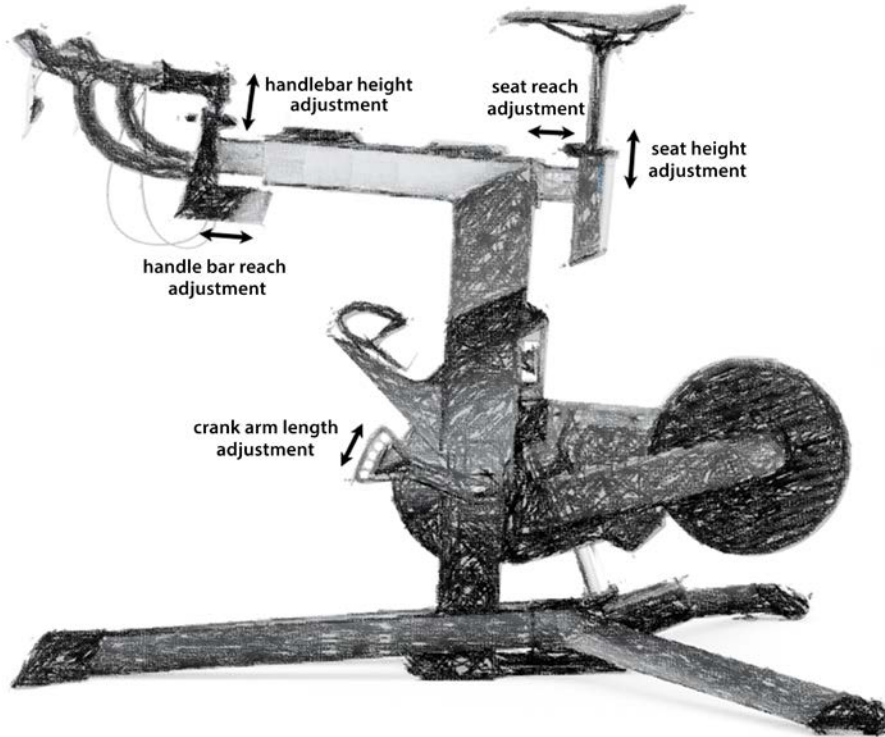


Figure A.2: Customizing bike fit

Monitor exercise metrics

Metrics monitored include speed, distance, power, cadence, time, and grade (incline). Data will be uploaded to connected devices at a rate of at least 5 Hz.

Export/upload exercise metrics

The system monitors, stores, and reports metrics over Bluetooth, ANT+, and ANT FEC interfaces. It is assumed that the user will provide their own heart rate strap for capturing and reporting their heart rate to their iOS, Android, or Windows devices.

Variable power output

Output can be set from 0 to 2,000 watts with a heavy flywheel to provide power smoothing over changing effort with an accuracy of 1%. Resistance may be manually set by the user or externally controlled via the Bluetooth or ANT FEC interfaces. Resistance is provided by electromagnetic resistance.

Gearing emulation

The system simulates standard bicycle gearing including the for the user to “change gears” in the same way they would on a road bike. The user can select the kind of gearing emulated including standard mechanical index shifting as well as DI-2 electronic shifting. The system can emulate 1-3 front chain rings with 30 to 60 teeth. Rear cassette emulation supports 9-12 rings ranging from 10 to 40 teeth.

Controllable power level

The power level can be controlled from 0 to 2,000 watts and provide a similar user feel for the current cadence and power output as the user would experience on a road back with similar gearing. Both user-set output level (“level mode” or “resistance mode”) and externally set output level (“ERG mode”) operational modes are provided.

Incline control

The incline angle of the bike may be set by the user or via an external system over the Bluetooth or ANT FEC interfaces. The range of incline may be set in the range of -15 to + 20 degrees. This can be automatically by an external device to emulate “terrain following” as the user rides over virtual bike courses. Alternatively, the user is provided with a pair of buttons to raise or lower the incline manually.

User interface

A simplified user interface is provided for viewing and changing selected gears on both the front chain ring and rear cassette.

Gearing

User buttons on the “brake hoods” of the handlebars allow for increasing or decreasing the selected gears. The LED display on the system frame shows the currently selected front and rear chain rings.

Incline

Up and down buttons on the frame allow the user to select the bike incline. A small LED display shows the current incline.

Setup

Separate (provided) iOS, Android, and Windows apps provide configuration and setup instructions.

Ride

There is no interface provided for the exercise or performance metrics. It is assumed that the user will provide a compatible third-party device for this purpose.

Online training system compatible

Almost all users will use third-party training systems that support the required interfaces to monitor and control the workout. These systems provide a range of capabilities ranging from running downloadable workouts and fitness assessment protocols to social gaming environments for virtual riding and racing. The most popular systems include Strava™, Trainer Road™, Zwift™, and Sufferfest™.

These applications run on third-party iOS, Android, or Windows devices and support ANT FEC, ANT+, and/or low-energy Bluetooth communications protocols. The system is open with published interfaces so that additional third-party device vendors can support the Pegasus.

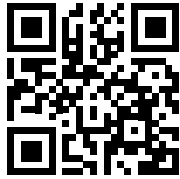
Configuration and OTA firmware updates

The Pegasus configuration app allows the user to register their product, search and download firmware upgrades, set the rider profile (height, weight, age, and gender), and the selected crank arm length. The app provides the means for initial calibration of the output settings via a guided protocol. This app is provided to the user and runs on iOS, Android, and Windows devices.

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/cpVUC>





packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

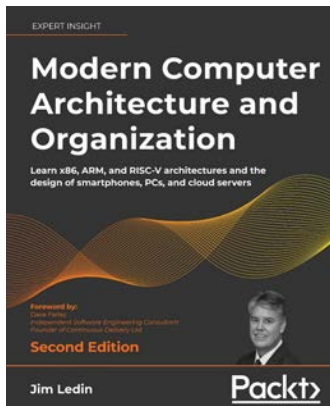
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

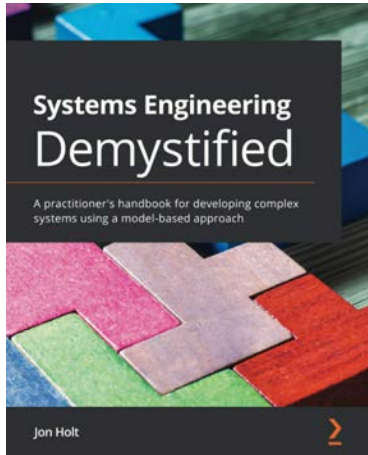


Modern Computer Architecture and Organization – Second Edition

Jim Ledin

ISBN: 9781803234519

- Understand the fundamentals of transistor technology and digital circuits
- Explore the concepts underlying pipelining and superscalar processing
- Implement a complete RISC-V processor in a low-cost FPGA
- Understand the technology used to implement virtual machines
- Learn about security-critical computing applications like financial transaction processing
- Get up to speed with blockchain and the hardware architectures used in bitcoin mining
- Explore the capabilities of self-navigating vehicle computing architectures
- Write a quantum computing program and run it on a real quantum computer



Systems Engineering Demystified

Jon Holt

ISBN: 9781838985806

- Understand the three evils of systems engineering - complexity, ambiguous communication, and lack of understanding
- Realize successful systems using model-based systems engineering
- Understand the concept of life cycles and how they control the evolution of a system
- Explore processes and related concepts such as activities, stakeholders, and resources
- Discover how needs fit into the systems life cycle and which processes are relevant and how to comply with them
- Find out how design, verification, and validation fit into the life cycle and processes

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share your thoughts

Now you've finished *Agile Model-Based Systems Engineering Cookbook, Second Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Index

A

Action Language for Foundational (ALF) 256

Agile methods 3, 4

agility

concepts 3

analysis model 302

animated sequence diagrams 447

architectural allocation 321

inputs 321

non-allocable requisites, decomposing 323

outputs 322

postconditions 322

preconditions 321

purpose 321

system requisites, allocation 323

Architectural Analysis Package 105

architectural design 270

Architectural Design Package 105

architectural merge 286

example 290

inputs 287

interfaces, copying from use cases
to system block 290

interfaces, merging 290

interfaces, updating to refer copied system
functions 290

issues, with merging specifications 286, 287

outputs 287

performing 287

postconditions 287

preconditions 287

purpose 287

system architecture 286

system context, creating 288

system data, copying from use cases
to system block 289, 290

system function conflicts, resolving 289

system functions, copying from use cases
to system block 289

architectural trade studies 272

architectural merge 286

assessment criteria, defining 275, 276

candidate solutions, identifying 274

inputs 272

key systems functions, identifying 274

MoE, assigning to candidate solution 278

outputs 273, 274

pattern-driven architecture 299

Pegasus smart bicycle trainer example 279

postconditions 273, 274

preconditions 272

purpose 272

sensitivity analysis, performing 278

solution, determining 278

- subsystem and
 - component architecture 310, 311
- utility curve, defining for assessment criterion 277
- weights, assigning to criteria 277
- architecture 85**
 - critical views 270
 - guidelines 271, 272
- Architecture 0**
 - electronic technical work items 100
 - inputs and preconditions 87
 - mechanical technical work items 100
 - outputs and postconditions 87
 - purpose 87
 - technical work items 100
 - workflow 87, 89
- Architecture 0, Harmony process views**
 - concurrency and resource view 85
 - dependability view 86
 - deployment view 86
 - distribution view 85
 - subsystem and component view 85
- Architecture 0, workflow**
 - computational needs, identifying 89, 90
 - computational patterns, selecting 90
 - dependability needs, identifying 90
 - dependability patterns, selecting 91
 - distribution needs, identifying 90
 - distribution patterns, selecting 91
 - electronic architecture, creating 91
 - engineering disciplines contributions, considering 89
 - example 92-100
 - existing technologies assets, considering 89
 - key technologies, identifying 89
 - mechanical architecture, creating 91
 - primary architectural optimizations, determining 88
 - scope of functionality, reviewing 88
 - subsystem architecture, creating 91
 - subsystem organizational concerns, identifying 89
 - subsystem patterns, selecting 90
 - technical work items, allocating to iterations 91
 - technical work items, creating 91
- asset diagram 217**
- assets**
 - types 221
- attack flow diagram 218, 219**
- authoritative source 380**
- automatic initialization of context and runtime objects**
 - reference link 447
- B**
- backlog management 6, 7**
 - inputs and preconditions 8
 - outputs and postconditions 8
 - purpose 8
 - use cases 13, 14
 - workflow 9
- backlog management, workflow**
 - effort, estimating 11
 - item, creating 10, 11
 - resolved work item, removing 12
 - work item, allocating in iteration backlog 12
 - work item, approving 11
 - work item, performing 12
 - work item, placing in project backlog 12
 - work item, prioritizing 11
 - work items, reorganizing 13
 - work performed, rejecting 12
 - work performed, reviewing 12
- backward traceability 495**

- baseline 115
- Bluetooth Low Energy (BLE) 393
- branch 116
- burn down chart 24
- Business Analysis Body of Knowledge (BABOK)
 - reference link 67
- business epics 39, 43
- C**
- Cameo prototype
 - reference link 199
- Cameo Safety and Risk Analyzer 198
- Cameo Teamwork Cloud 117
- Capabilities Package 104
- change management 113, 114
 - inputs and preconditions 115
 - outputs and postconditions 115
 - purpose 114
 - workflow 115
- change management, workflow
 - branch and merge 116
 - change activity, making 117
 - changes, verifying 117
 - elements, locking 117
 - elements, modifying 117
 - elements, unlocking 117
 - example 122-129
 - Lock and Release 116
 - model, opening 116
 - modification, determining 116
 - updated model, storing 117
 - work item, obtaining 116
- collaboration design 270
- collaboration parameter 300
- Collaborator 509
- Command-and-Control Battle Management (C2BM) platform 354
- Commercial Off-the-Shelf (COTS) 365
- Common Vulnerability Enumeration (CVE) 222
- Common Weakness Enumeration (CWE) 222
- compliance in form 441
- compliance in meaning 441
- component architecture 270
- computable constraint modeling 483
 - inputs and preconditions 484
 - outputs and postconditions 484
 - purpose 483
- computable constraint modeling, example 486
 - analysis of units, performing 490
 - available truths, identifying 487
 - mathematical formulation, constructing 488
 - necessary properties of acceptable solutions, determining 488
 - problem, constraining to relevant information 488
 - problem, rendering as set of connected constraint properties 488, 490
 - problem, understanding 486
 - sanity check, performing 493
- computable constraint modeling, workflow
 - analysis of units, performing 486
 - available truths, identifying 485
 - mathematical formulation, constructing 485
 - necessary properties of acceptable solutions, determining 485
 - problem, constraining to relevant information 485

- problem, rendering as set of connected
 - constraint properties 485
- problem, understanding 485
- sanity check, performing 486
- computable model 444**
- concurrency and resource architecture 270**
- Configuration Item (CI) 114**
- configuration management 113**
 - reference link 114
- constraints 485**
 - blocks 485
 - parameters 485
 - properties 485, 489
- Control Area Network (CAN) 394**
- Cost of Delay (CoD) 72**
- Critical Design Review (CDR) 443**
- critical views, architecture**
 - concurrency and resource architecture 270
 - data architecture 270
 - dependability architecture 270
 - deployment architecture 271
 - distribution architecture 270
 - subsystem/component architecture 270
- cut set 200**
- Cyber-Physical Security 213**
 - key concepts 213-215
- cyclomatic complexity 20**
- D**
- data architecture 270**
- Data Distribution Service (DDS) 391**
- data schema 248**
- Decision Analysis Matrix 273**
- dependability architecture 270**
- deployment architecture 271, 404**
 - engineering facets 404
 - interdisciplinary interfaces 421
 - privacy architecture 270
 - reliability architecture 270
 - safety architecture 270
 - security architecture 270
- Deployment Architecture Package 110**
- design patterns 299**
 - design context, defining 302
 - design solution, validating 302
 - dimensions 299
 - example 303
 - in architectural context 300
 - inputs 301
 - issues, determining 302
 - issues, solving 302
 - outputs 301
 - pattern roles 300
 - postconditions 301
 - potential pattern solutions, selecting 302
 - preconditions 301
 - purpose 301
 - selected pattern, instantiating 302
 - trade study, performing 302
- detailed design 270**
- Digital Indexed (DI) 396**
- DI shifting**
 - reference link 191
- distribution architecture 270**
- DO-178C**
 - reference link 464

E**effort estimation 58**

- inputs and preconditions 59
- outputs and postconditions 59
- purpose 58
- use cases 62, 63
- user stories and scenarios 59

ego-less reviews 513**enabler epics 39, 44****Engineering Change Order (ECO) 114****engineering facets**

- block, creating 408, 411
- deployment architecture 404-406
- example 411
- initial facet load analysis,
 - performing 409, 410, 420
- inputs and preconditions 406
- involved disciplines, identifying 407, 411
- non-allocatable requisites,
 - decomposing 408, 413-415
- non-allocatable subsystem features,
 - decomposing 408, 417-420
- outputs and postconditions 406, 407
- purpose 406
- requisites, allocating 408, 417
- subsystem features, allocating 408, 420

epic 39**example, of architectural allocation 323, 324**

- non-allocable requirements 324, 325
- non-allocable system functions
 - and data, decomposing 332-334
- system functions
 - and data allocation 330, 331
- system requirements allocation 328-330

example, of architectural merge

- interfaces, copying from use cases
 - to system block 297
- interfaces, merging 298
- interfaces, updating to refer copied system functions 297
- interfaces, updating to refer copied systems data 297
- system context, creating 291
- system data conflicts, resolving 296
- system data, copying from use cases
 - to system block 296
- system function conflicts, resolving 295
- system functions, copying from use cases
 - to system block 292-294

example, of design pattern 303

- design context, defining 303, 304
- design solution, validating 310
- issues, determining 304
- issues, solving 304
- potential pattern solutions,
 - selecting 304-308
- selected pattern, instantiating 309
- trade study, performing 309

example, of reference architecture

- architectural elements, adding 360, 361
- areas of deviation, identifying 357
- instance specifications, creating 361
- instance specification slots,
 - populating 361, 362
- reference architecture, selecting 356, 357
- relevant architectural elements,
 - subclassing 358, 361
- specific architectural elements,
 - redefining 359
- specific system block, defining 358
- specific system instance specification,
 - defining 361

executable model 444

F

facets 89, 90

Fagan inspections

reference link 505

Failure Means and Effect (FMEA) 198

Fault Tree Analysis (FTA) 198

Federating Models for Handoff 380, 381

example 386

inputs and preconditions 382

outputs and postconditions 382

purpose 381

SE model references, adding
to Shared Model 383-386

SE model references, adding
to subsystem model 385, 388

Shared Model, creating 383, 386

Shared Model references, adding
to subsystem model 386, 388

Shared Model, structuring 383, 386

subsystem model, creating 384, 387

subsystem model, structuring 384, 385, 388

federation 380

Final Operating Condition (FOC) 40

forward traceability 495

Fraction of Inspired Oxygen (FiO2) 205

free-form diagram 447

Functional Analysis Package 105

**functional analysis with activity model,
example** 157

execution context, defining 158

ports and interfaces, creating in execution
context 163-167

primary functional flow, identifying 160

related actors, identifying 158

Requirements_change 170

requisites set, updating 170, 171

requisites show, performing 171

requisites, validating 168, 169

requisites, verifying 168, 169

trace links, adding 171

use case, describing 158

use case, performing 171

use case scenarios, deriving 161, 162

**functional analysis with activity model,
workflow**

executable state model, creating 156

execution context, defining 155

ports and interface, creating in execution
context 156

primary functional flows, identifying 155

related actors, identifying 155

Requirements_change 156

requisite set, updating 156

requisites review, performing 157

requisites, validating 156

requisites, verifying 156

trace links, adding 157

use case, describing 155

use case, identify 154

use case, performing 157

use case scenarios, deriving 156

functional analysis with scenarios 134

inputs and preconditions 135, 153

outputs and postconditions 135, 153

purpose 134, 153

**functional analysis with scenarios,
example** 138

activities 153

executable state model, creating 146-149

execution context, defining 139

ports and interface, creating in execution
context 145

related actors, identify 139

Requirements_change 151

- requisites review, performing 152
- requisites set, updating 151
- requisites, validating 149-151
- requisites, verifying 149-151
- trace links, adding 152
- use case, describing 138
- use case, identifying 138
- use case, performing 152
- use case scenarios, capturing 141-144
- functional analysis with scenarios, workflow 135, 153**
 - executable state machine, creating 137
 - execution context, defining 136
 - ports and interface, creating in execution context 136
 - related actors, identify 136
 - Requirements_change 137
 - requisites review, performing 137
 - requisites set, updating 137
 - requisites, validating 137
 - requisites, verifying 137
 - trace links, adding 137
 - use case, describing 136
 - use case, identify 136
 - use case, performing 137
 - use case scenarios, capturing 136
- functional analysis with state machine 171**
 - inputs and preconditions 172
 - outputs and postconditions 172
 - purpose 172
- functional analysis with state machine, example 176**
 - executable state model, creating 179-182
 - execution context, defining 177
 - ports and interface, creating in execution context 179
 - related actors, identifying 177
 - Requirements_change 183
 - requisites review, performing 185
 - requisites set, updating 184
 - requisites, validating 183
 - requisites, verifying 183
 - trace links, adding 184
 - use case, describing 176
 - use case, performing 185
 - use case scenarios, generating 182
- functional analysis with state machine, workflow 172**
 - executable state model, creating 174
 - execution context, defining 174
 - ports and interface, creating in execution context 174
 - related actors, identifying 174
 - Requirements_change 175
 - requisites review, performing 175
 - requisites set, updating 175
 - requisites, validating 175
 - requisites, verifying 175
 - trace links, adding 175
 - use case, describing 173
 - use case, identifying 173
 - use case, performing 175
 - use case scenarios, generating 174
- functional analysis with user stories 185, 186**
 - guidelines 187, 188
 - inputs and preconditions 189
 - outputs and postconditions 189
 - purpose 189
- functional analysis with user stories, example 191**
 - buttons, using to shift gears 193
 - functional requisites, specifying 194
 - quality of service as requisites, identifying 194
 - related actors, identifying 192

- Requirements_change 195
- requisites review, performing 196
- requisites set, updating 195
- requisites, verifying 195
- trace links, adding 195
- use case, describing 192
- use case, identifying 191
- use case, performing 196

functional analysis with user stories, workflow 189

- <role> statement 190
- acceptance criteria, specifying 190
- quality of service as requisites, identifying 190
- related actors, identifying 190
- Requirements_change 191
- requisites review, performing 191
- requisites set, updating 191
- requisites, validating 190
- requisites, verifying 190
- trace links, adding 191
- use case, describing 190
- use case, identifying 190
- use case, performing 191

H

Handoff to Downstream Engineering

- activities 364, 365
- purpose 363

Handoff to Downstream Engineering, activities

- examples 365-369
- logical interface, migrating
 - to physical interfaces 389
- model federation 380
- preparation 372

Harmony Agile Model-Based Systems Engineering process (Harmony aMBSE) 85

Harmony for Embedded Software process (Harmony ESW) 85

Harmony process

- Harmony aMBSE 85
- Harmony ESW 85

hazard analysis 202

- inputs and preconditions 202
- outputs and postconditions 202
- purpose 202

hazard analysis, example 205

- causality model, creating 208
- conditions and events, describing 207
- cut set, identifying 209
- hazard, describing 206
- hazard, identifying 205
- problem statement 205
- related conditions and events, identifying 207
- results, adding in FTA 210
- safety concept, reviewing 211
- safety measures, adding 209
- safety requisites adding 212

hazard analysis, workflow 202

- causality model, creating 204
- conditions and events, describing 204
- cut set, identifying 204
- hazard, describing 203
- hazard, identifying 203
- related conditions and events, identifying 203
- requisites review, performing 205
- safety concept, reviewing 205
- safety measures, adding 204

safety requisites, adding 205
trace links, adding 205
use case, performing 205

I

Initial Operating Condition (IOC) 40

interdisciplinary interfaces 421

agreed-upon interfaces, storing in
configuration management 424, 437
allocations, reviewing to engineering
facets 422
examples 424
facet properties, identifying 422, 425
inputs and preconditions 421
interaction, defining among facet
properties 423, 426, 427
interaction metadata,
capturing 423, 428, 433
interactions, grouping across facets into
interfaces 423, 437
interface details, negotiating 423, 437
outputs and postconditions 421
port pairs, defining between facets 423, 425
purpose 421

Interdisciplinary Product Team (IPT) 406

Interface Control Document (ICD) 389

Interface Diagram 390

Interfaces Package 105

Interface Specification Table 390

Internal Block Diagram (IBD) 95

Iteration 0 78

areas of focus 79, 80
inputs and preconditions 78
outputs and postconditions 79
purpose 78
workflow 80

Iteration 0, workflow

architectural goals, identifying 82
example 83-85
high-level architecture, defining 83
product roadmap, creating 81
product vision, creating 81
providing, team with domain knowledge 81
providing, team with process knowledge 82
providing, team with skills and tools 81
release plan, creating 81
risk management plan, creating 81
team, selecting 81
team workspaces, setting up 82
tool installations, testing 82
tools, configuring 82
tools, installing 82

iteration plan 51

backlog, adjusting 54
effort, estimating for work tasks 53
inputs and preconditions 51
iteration mission, reviewing 53
outputs and postconditions 52
purpose 51
tasks, adding to backlog for iteration 53
team, adjusting 54
team loading, evaluation 54
use cases, breaking into user scenarios 53
use stories, breaking into tasks 53
workflow 52
work items, selecting from backlog 53

iteration plan, workflow

example 55-58

K

Key Performance Indicator (KPI) 19

L**League of Racers Extraordinaire (LORE) 358****logical data schema**

- creating 246
- defining 246-248
- example 248-250
- inputs and preconditions 251
- outputs and postconditions 251
- purpose 251

logical data schema, example 257

- blocks, identifying 258
- collaboration, creating 257
- interaction, defining 260-263
- properties for metadata, defining 266, 267
- relations, adding 259
- structure, defining 258
- type model, constructing 263
- units, defining 263
- value properties, identifying 260
- value type, applying to relevant properties 265
- value type, defining 265-267

logical data schema, workflow 252

- block, identifying 254
- collaboration, creating 254
- flow item to structure, defining 255
- flow message, defining 255
- flow property, identifying 254
- interaction, defining 254
- message, identifying 255
- message parameters, adding 255
- metadata of interest, identifying 257
- metadata aspects, specifying 256
- metadata properties, filling 257
- metadata tags, adding 257
- operation, defining 255

- quantity kind, defining 256
- relation, adding 254
- signal attributes, defining 255
- signal reception, defining 255
- stereotype, creating 257
- stereotypes to properties, applying 257
- structure, defining 254
- type, applying to relevant properties 256
- type model, constructing 255
- unit, defining 256
- value property, identifying 254
- value type, defining 256

logical interface, migrating to physical interface 389, 390

- examples 393
- ICD, representing 390
- inputs and preconditions 390
- interface visualization, creating 393, 403
- navigable links, adding 393, 401, 402
- outputs and postconditions 391
- purpose 390

logical interfaces

- physical realization,
 - defining 392, 393, 396-400
- referencing 391, 393

logical system interfaces

- inputs and preconditions 235
- outputs and postconditions 235
- purpose 235
- specifying 231
- SysML ports and interfaces 231, 232

logical system interfaces, example 237

- activity flow, creating 238-240
- execution context, creating 238
- flows, adding 244
- message parameters, adding 244
- naming conventions 244-246

- parameter and flow types, creating 244
- ports and interfaces, creating 244
- related actors, identifying 237
- use case, identifying 237
- use case scenarios, capturing 241, 243

logical system interfaces, workflow 235

- activity flow, creating 236
- execution context, creating 236
- message parameters, adding 237
- parameter and flow types, creating 237
- ports and interfaces, creating 237
- related actors, identifying 236
- UML flows, adding 237
- use case, identifying 236
- use case scenarios, capturing 236

Low-energy Bluetooth (BLE) 92

M

Mathematically-Addressable Problems (MAPs) 483

MBSE and MDD guidelines
reference link 441

Mean Time Between Failure (MTBF) 204, 279

Measure Performance Metrics 223

Medical Gas Mixer (MGM) 205

metadata 8, 246

metrics 20

Minimal Testable Feature (MTF) 521

Minimal Verifiable Feature (MVF) 521

model
purpose 103

model-based safety analysis 196
cut set 200, 201
key terms 197
profile 198-200

Model-Based Systems Engineering (MBSE) 4-6

model-based testing (MBT) 460, 461
inputs and preconditions 462
outputs and postconditions 462
purpose 462
reference link 461

model-based testing (MBT), example 465, 466
defects, fixing in SUT 482
system under test, identifying 466-471
test architecture, defining 472
test cases, applying 481
test cases relating to requirements 474
test cases, rendering 475-480
test cases, specifying 473
test coverage, analyzing 474
verdicts, rendering 481

model-based testing (MBT), workflow 463
defects, fixing in SUT 465
system under test, identifying 463
test architecture, defining 463
test cases, applying 465
test cases relating to requirements 464
test cases, rendering 465
test cases, specifying 464
test coverage, analyzing 464
verdicts, rendering 465

model-based threat analysis 212
Cyber-Physical Security 213
inputs and preconditions 220
modeling for security analysis 215
outputs and postconditions 220
purpose 220

model-based threat analysis, example 223
asset contexts, describing 224
asset contexts, identifying 224

- assets, describing 224
- assets, identifying 224
- attack chain, specifying 225
- causality tree, creating 228
- countermeasures, adding 229
- requisites review, performing 230
- security posture, reviewing 229, 230
- security requisites, adding 230
- trace links, adding 230
- use case, performing 230
- vulnerabilities, identifying 225
- model-based threat analysis, workflow 220**
 - asset contexts, describing 222
 - asset contexts, identifying 221, 222
 - assets, describing 222
 - assets, identifying 221, 222
 - attack chains, specifying 222
 - causality tree, creating 222
 - countermeasures, adding 222
 - requisites review, performing 223
 - security posture, reviewing 223
 - security requisites adding 223
 - trace links, adding 223
 - use case, performing 223
 - vulnerabilities, identifying 222
- model federation 380**
- model fidelity 103**
- modeling for security analysis 215, 216**
 - asset diagram 217
 - attack flow diagram 218, 219
 - security analysis diagram (SAD) 217
 - tabular views 220
- Model Manager 509**
- Model Overview Diagram**
 - example 106
- Model Overview Package 104**
- model simulation 444**
 - inputs and preconditions 445
 - outputs and postconditions 445
 - purpose 445
- model simulation, example 448**
 - behaviors, defining 450
 - creating 452
 - desired outputs/outcomes, identifying 450
 - outputs and outcomes,
 - analyzing 454, 455, 458, 460
 - purpose, defining 448
 - running 452
 - structural context, defining 448, 449
 - view, defining 451
- model simulation, workflow 445**
 - behaviors, defining 446
 - creating 447
 - desired outputs/outcomes, identifying 447
 - outputs and outcomes, analyzing 448
 - purpose, defining 446
 - running 447
 - structural context 446
 - structure, defining 447
- model version 114**
- model work items**
 - inputs and preconditions 514
 - managing 514
 - outputs and postconditions 514
 - purpose 514
- model work items, example**
 - characterizing 517, 518
 - creating 517
 - enacting 519
 - selecting 518
 - structures, creating 516, 517
 - updating 519

model work items, workflow 515
characterizing 516
creating 515
enacting 516
selecting 516
structures, creating 515
updating 516

N

nanocycle 520

O

Over-the-Air (OTA) updates 409

P

package 114

Pain Cave model
composition architecture 118
connected context 118
elements, behavior 120
simulation view 120, 121

pattern arguments 300

pattern-driven architecture 299

pattern instantiation 299

pattern mining 299

pattern roles 300

Pegasus 545

Pegasus key features 545
customizable bike fit 546
emulation, gearing 547
exercise metrics, exporting 546
exercise metrics, monitoring 546
exercise metrics, uploading 546
incline control 547
online training system compatible 548

OTA firmware updates 548
power level, controlling 547
user interface 547
variable power output 547

Pegasus smart bicycle trainer, trade study example
assessment criteria, defining 279, 280
candidate solutions, identifying 279
key system functions, identifying 279
MoE, assigning for candidate solution 283, 284, 285
sensitivity analysis, performing 285
solution, determining 286
utility curve, defining for criterion 282
weights, assigning to criteria 281, 282

Pegasus system architecture, example
abstract common services layer 318
abstract HW layer 318, 319
application layer 317
communications layer 318
mission statement, assigning 320
potential architectural patterns, reviewing 316
requirement allocation 321
subsystem and component architectural pattern, selecting 317
subsystem interfaces, creating 321
system features, allocating 321
systems functions and data, grouping into coherent sets 315, 316
UI layer 318

Pegasus System Model
overview diagram 112

physical interfaces
technology, selecting 391-395

planning poker 58
workflow 60-62

- polymorphic behavior** 354
- Preliminary Design Review (PDR)** 443
- preparation, of Handoff to Downstream Engineering**
 - examples 374
 - inputs and preconditions 372
 - interface data, organizing 373, 378
 - outputs and postconditions 372
 - purpose 372
 - requirements, organizing 373, 376
 - review model, for handoff
 - readiness 373, 380
 - subsystems, organizing 373, 375
- prioritized epics** 44
- priority** 66
 - influencing, factors 66
- priority poker** 67
- product owner/presenter** 508
- product roadmap** 38-40
 - agreement, obtaining 42
 - broad product timeframe, assigning 42
 - epics, allocating in product timeframe 42
 - epics, creating 42
 - epics, prioritizing 42
 - inputs and preconditions 39
 - outputs and postconditions 39
 - product themes, enumerating 41
 - purpose 39
 - updating 43
- product roadmap, for Pegasus system**
 - agreement, obtaining 45
 - epic, allocating into product timeframe 45
 - epics, creating 43
 - epics, prioritizing 44
 - product themes, enumerating 43
 - product timeframe, assigning 44
 - updating 45

- project risk management** 28
 - inputs and preconditions 29
 - outputs and postconditions 29
 - purpose 29
 - types 28

Q

- Qualities of Service (QoS)** 299
- Quality Assurance (QA) reviewer** 508
- quantity kind**
 - identifying 263, 264

R

- Real-Time Agility** 3
- reference architecture** 349
 - architectural element, adding 354
 - areas of deviation, identifying 353
 - compliance, demonstrating 355
 - example 356
 - features 349-351
 - inputs 351
 - instance specification slots, populating 355
 - outputs 351
 - postconditions 351
 - preconditions 351
 - purpose 351
 - relevant architectural elements,
 - subclassing 354, 355
 - selecting 353
 - specializing 348
 - specific architecture elements,
 - redefining 354
 - specific architecture instance, creating 354
 - specific architecture type, creating 353
 - specific system block, defining 354

- specific system instance specification, defining 355
- system instance specification, creating 355
- workflow 352
- release plan 46**
 - epics decomposing 47
 - epics high-level goals, identifying 47
 - example 48-51
 - inputs and preconditions 46
 - iteration missions, establishing 47
 - iteration plan, reviewing 48
 - iteration work items, prioritizing 48
 - outputs and postconditions 46
 - purpose 46
 - work items, allocating to iterations 48
- Request for Proposals (RFPs) 365**
- Requirements Package 105**
- resistance generation, smart bicycle trainer**
 - electric motor with flywheel 279
 - electrohydraulic 279
 - hydraulic with flywheel 279
 - Wind Turbine 279
- review**
 - inputs and preconditions 506
 - outputs and postconditions 506
 - overview 504, 505
 - purpose 505
- review coordinator 508**
- reviewers**
 - comments and issues 513
- review, example 510**
 - action item resolutions, planning 514
 - action items, resolving 514
 - issues, capturing as action items 514
 - materials, disseminating 511
 - materials, preparing 510
 - purpose, establishing 510
 - reviewer comments, discussing 513
 - reviewer roles, assigning 511
 - reviewers independently inspect materials 512
 - scheduling 512
- review, workflow 506**
 - action item resolutions, planning 510
 - action items, resolving 510
 - issues, capturing as action items 509
 - materials, disseminating 509
 - materials, preparing 507
 - purpose, establishing 507
 - reviewer comments, discussing 509
 - reviewer roles, assigning 508
 - reviewers independently inspect materials 509
 - scheduling 509
- revision 114**
- Rhapsody RAAML prototype**
 - reference link 199
- Risk Analysis and Assessment Modeling Language (RAAML) 198**
- risk list 29**
- risk management plan 30**
 - example 33, 36-38
 - outcome, assessing 33
 - potential source, identifying 31
 - replanning 33
 - risk, characterizing 31
 - risk list, adding in priority order 32
 - spike, identifying to address risk 32
 - spike, performing 33
 - spike work item, allocating to iteration plan 32
 - updating 33
 - work item, creating for spike 32

S

Scaled Agile Framework (SAFe) 67

scribe 508

security analysis diagram (SAD) 217

semantic reviews 504

semantic verification 441

smart packages 508

SME reviewer 508

spike 29, 32

Subject Matter Experts (SMEs) 504

**subsystem and component
architecture 310, 311**

data, allocating 314

example 315

inputs 312

mission statement, assigning 314

outputs 313

pattern, selecting 314

postconditions 313

potential architectural patterns,
reviewing 314

preconditions 312

purpose 312

requirements allocation 314

selecting 311

subsystem interfaces, creating 315

subsystem, modeling in SysML 311

system features, allocating 314

systems functions and data, grouping
into coherent sets 314

validating 315

workflow 313

subsystem interfaces

creating 338

creating, from use case scenarios 334, 335

example 339

inputs 335

outputs 335

owner, adding to hold white box
scenarios 336

postconditions 335

preconditions 335

purpose 335

scenario, creating 337

source scenario, replicating 337

subsystem interaction messages,
adding 337

subsystem ports, creating 338

updating, with messages 339

subsystem interfaces, example

interfaces, creating 347

interfaces, updating with messages 348

original scenario, replicating 341, 342

owner, adding to hold white box
scenarios 339

scenario, creating 339, 340

subsystem interaction messages,
adding 342-345

subsystem ports, creating 347

Subsystem Package 111

subsystems 89

cons 89

pros 89

success metrics measurement 19, 20

failure, reasons 27, 28

inputs and preconditions 21

outputs and postconditions 21

purpose 21

usage, example 24-27

workflow 21-23

- SW Design Package 111**
- syntactic reviews 504**
- syntactic verification 441**
- SysML ports and interfaces 231, 232**
 - continuous flows 232-234
- system requirement**
 - defining 133, 134
- System Requirements Review (SRR) 443**
- Systems Models 112**
 - organizing 101
 - Requirements package 111
- Systems Models, organizing**
 - example 112, 113
 - inputs and preconditions 101
 - outputs and postconditions 101
 - purpose 101
 - workflow 102-111

T

- tabular views 220**
- test-driven development (TDD) 519**
- Test-Driven Modeling (TDM) 519, 520**
 - example 522-543
 - inputs and preconditions 520
 - outputs and postconditions 520
 - purpose 520
- Test-Driven Modeling (TDM), workflow 520**
 - bit, modeling 522
 - defects, fixing 522
 - test case, applying 522
 - test case, defining 521
- textual requirements 132**
- traceability 493, 496, 497**
 - definitions 495
 - inputs and preconditions 497

- outputs and postconditions 498
- purpose 497
- trace links, types 495, 496

- traceability, example 502**
 - goals, establishing 502
 - links, adding 502
 - relations, deciding to use 502
 - reviewing 503, 504
 - views, creating 502
 - views, specifying 502
- traceability, workflow 498**
 - goals, establishing 498
 - links, adding 501
 - relations, deciding to use 499
 - reviewing 501
 - views, creating 501
 - views, specifying 500
- trace matrices 495**
- trace ownership 495**
- Traffic Light Controller (TLC) 523**
- trunk 116**

U

- UML Testing ProfileTM 2 (UTP 2)**
 - reference link 461
- Use Case Package 105**
- user interfaces (UIs)**
 - technology, selecting 392, 395, 396
- user interfaces (UIs), Pegasus key features**
 - gearing 547
 - incline 548
 - ride 548
 - setup 548

V

validation 441, 443

velocity 24

verification 441, 442

W

Weighted Shortest Job First (WSJF) 67

work items

mapping 46

work items prioritization 66

example 72, 78

inputs and preconditions 67

outputs and postconditions 67

purpose 67

workflow 68-71

WSJF approach 71, 72

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there- you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781803235820>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

