Practical Systems Engineering

by the Systems Engineering Community
re-teaching.org

November 11, 2014

Contents

Chapter 1

Introduction

Requirements Management and Engineering (RE&M) is taught, both in industry and academia. The availability of open source SE-tools, and Eclipse-based tools in particular, created some interest for using those tools for teaching.

As RE&M is often seen as a discipline of Systems Engineering, our scope is systems engineering, with an **initial** focus on requirements engineering.

1.1 Tag des Systems Engineering 2014

These materials were distributed for the workshop “Modellgetriebene Systementwicklung mit Eclipse”, held by Michael Jastram on November 12, 2014 in Bremen. Please note:

- The workshop covers more than what is included here. In particular, it covers various aspects of formal modeling, which you can read in the Rodin Handbook.
- What you are reading here is part of a community project, hosted at re-teaching.org.
- These materials are openly licensed (see Section 1.7). Please support our work by collaboration and by donating your time.
- This is work in progress. Please visit re-teaching.org for the latest updates.
- This work is supported by Formal Mind GmbH. Please check out their services, and their Eclipse-based products for systems engineering and requirements engineering.

1.2 Vision

The vision of this project is to create:

1. A set of teaching materials that is actively used;
2. Which is embedded in a larger SE context; and
3. Which explicitly focuses on applying RE.

1.3 Software vs. Systems Engineering

For the purpose of this tutorial, a system has interfaces with other software or hardware, while software is (more or less) stand-alone. Using this definition, we see systems engineering simply as an extension to software engineering, but with interfaces that can be unreliable.

1.4 Standards

ISO 29110 looks promising as the foundation for the method. Eclipse-based tools in general, and ProR for requirements engineering in particular, will be used. We are currently looking for a suitable case study, ideally using something that already exists. The focus will be on the creation of shared teaching materials.

1.5 Tools

A central idea of this project is the use of freely available tools, as we cannot expect students to invest in expensive tools. Tools will be based on Eclipse. Figure 1.1 shows on the left a simplified V-Model, depicting the tools we plan to use.

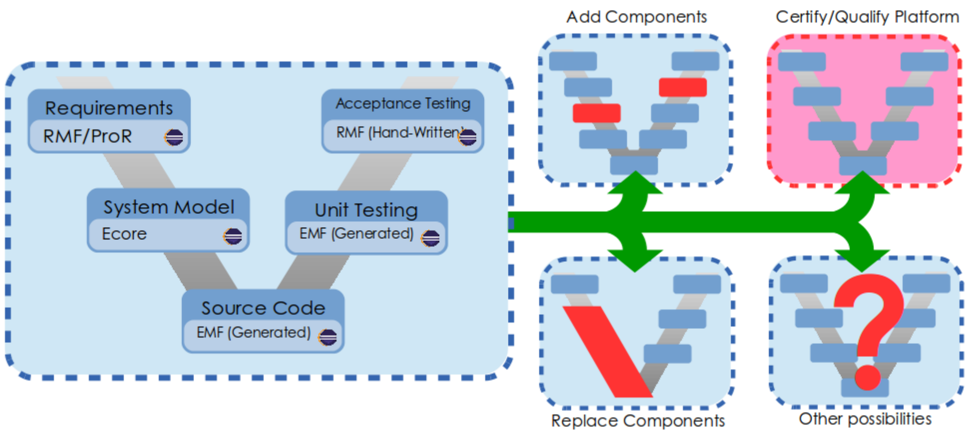


Figure 1.1: Tools used in this course

Information. In a "real" project, there would be many more tools and artifacts. We will keep tools and artifacts to a minimum, in order not to overwhelm the students.

As you can see, we plan on building a **minimal, complete, Eclipse-based** software engineering platform. It is loosely organized according to the V-Model.

1.5.1 Modularity and Extendability

Figure 1.1 depicts on the right how this toolchain can be adapted to your needs.

Openness helps drastically to make this possible:

Open Standards make it possible to replace individual tool components, without disturbing the toolchain as a whole. Open Standards that we use include ReqIF, Java and JUnit.

Open Software allows the toolchain to be tailored and seamlessly integrated in a way that is very difficult to do otherwise.

1.6 Background

This project started in July 2014 as a discussion on LinkedIn. Thank you to all contributors!

1.7 License

This content is licensed as Apache 2.0. If you contribute to the corresponding gitHub repository, you implicitly license the content that way.

Chapter 2

Tutorial

This chapter contains a mini-tutorial that has been used by Michael Jastram for the TdSE 2014 talk Modellgetriebene Systementwicklung mit Eclipse.

2.1 Overview

This tutorial covers the development of a small traffic light system, as shown in Figure 2.1.

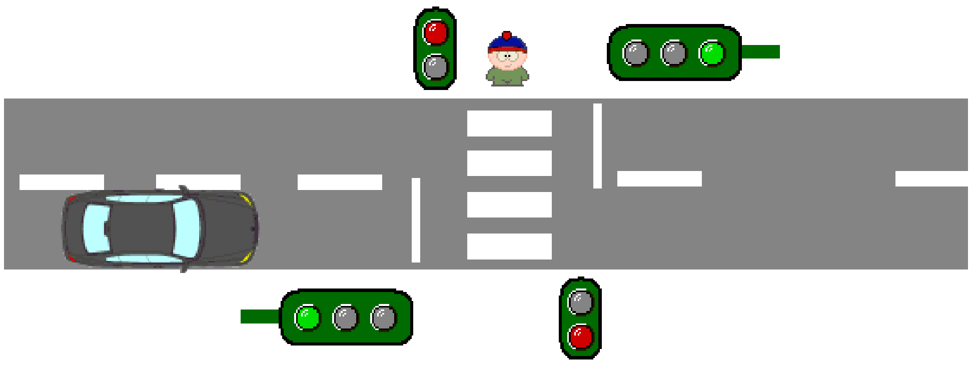


Figure 2.1: We will model a simple traffic light system.

2.2 Tool Installation

As of this writing, a complete toolchain is not yet available. The following describes the installation from various components.

2.2.1 Eclipse

The basis for the toolchain are the Eclipse Modeling Tools. Please download for your platform and extract to a convenient location and start it.

Information. It may be not a bad idea to start with Polarsys instead, as it already includes Papyrus.

Warning. On Linux, please edit eclipse.ini and add the following parameter **at the top (two lines)**:

```
--launcher.GTK_version
2
```

2.2.2 RMF and Formal Mind Essentials

Next install the RMF (requirements) tools, but the repackaged version from Formal Mind:

- Use this update site: http://update.formalmind.com/studio
- Unselect "Group items by category"
- Select **only** "Formal Mind Studio (Feature)"
- Complete the installation.

Information. The software is currently not signed, which will generate a warning. Please continue with the installation, in spite of this.

2.2.3 Java FX

If you want to use rich text in requirements, you need support for Java FX. Follow these steps:

- Use this update site: http://download.eclipse.org/efxclipse/runtime-released/1.1.0/site
- Optional: Unselect "Group items by category"
- Select **only** "Runtime Bundle Collector Feature"
- Complete the installation.

2.2.4 RMF-EMF Traceability

In the context of a public research project (itea openETCS), a traceability plug-in for connecting arbitrary EMF models has been developed.

- Use this update site: http://openetcs.ci.cloudbees.com/job/openETCS-tycho/lastSuccessfulBuild/artifact/tool/bun
- Unselect "Group items by category"
- Select **only** "ProR Tracing Feature"
- Complete the installation.

2.2.5 Additional Modeling Components

You can install additional components for modeling via `Papyrus via Help | Install Modeling Compo nents`. For this tutorial, useful components include:

Ecore Tools. Support diagram notation for Ecore models.

Papyrus. Supports UML and SysML.

We had some problems with installing the Ecore Tools. If you cannot see a diagram in Section **??**, then follow these steps:

Install software from this update site: http://download.eclipse.org/ecoretools/updates/releases/2.0.1/luna Select Ecore Diagram Editor

2.2.6 Team Support

Eclipse supports a number of team environments. We recommend the installation of the egit plugin, allowing to work with git repositories. The installation is described in the formalmind Studio Handbook.

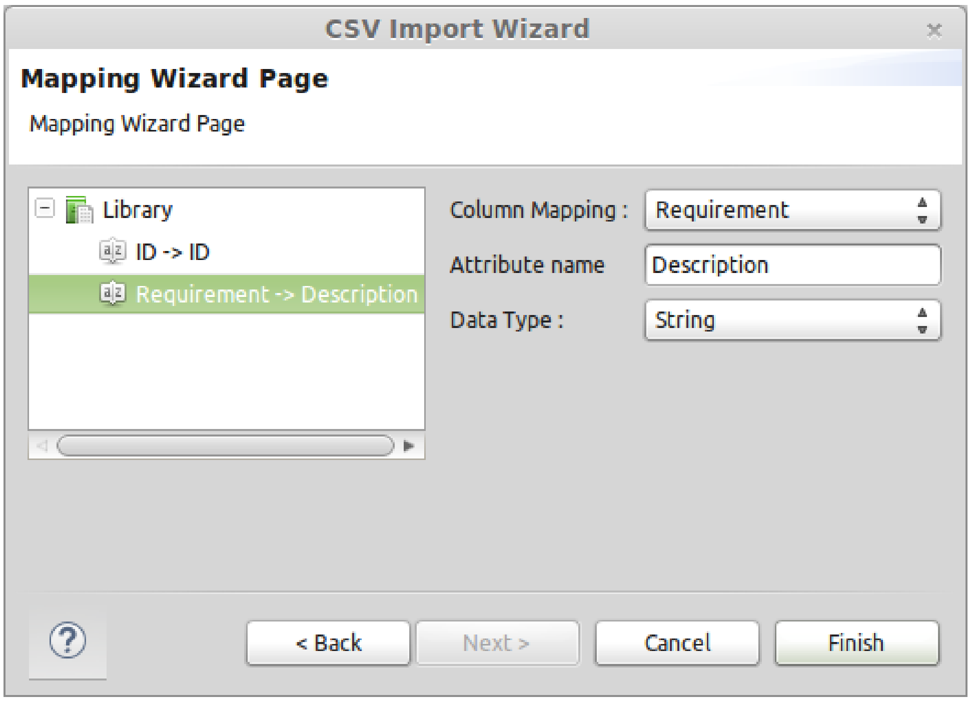
2.2.7 Tool Configuration

We recommend to switch to the ProR perspective, to get started.

2.3 Import Requirements

Typically, you already have requirements available in some form. ProR includes a simple CSV-Importer that allows you to import existing requirements. Follow these steps:

- Create a new Project via `File | New | Project... | General | Project`
- Call it `tdse-1`
- Create a new Requirements Model via right-click on the project, selecting `New | Reqif10 Model`
- Call the Model `Trafficlight.reqif`
- Import the .csv file via `File | Import | formalmind Studio | CSV`
- Create a mapping for the two columns to String attributes, as shown in Figure 2.2


CSV Import Wizard

Mapping Wizard Page

Mapping Wizard Page

Library
- ID -> ID
- Requirement -> Description

Column Mapping : Requirement

Attribute name Description

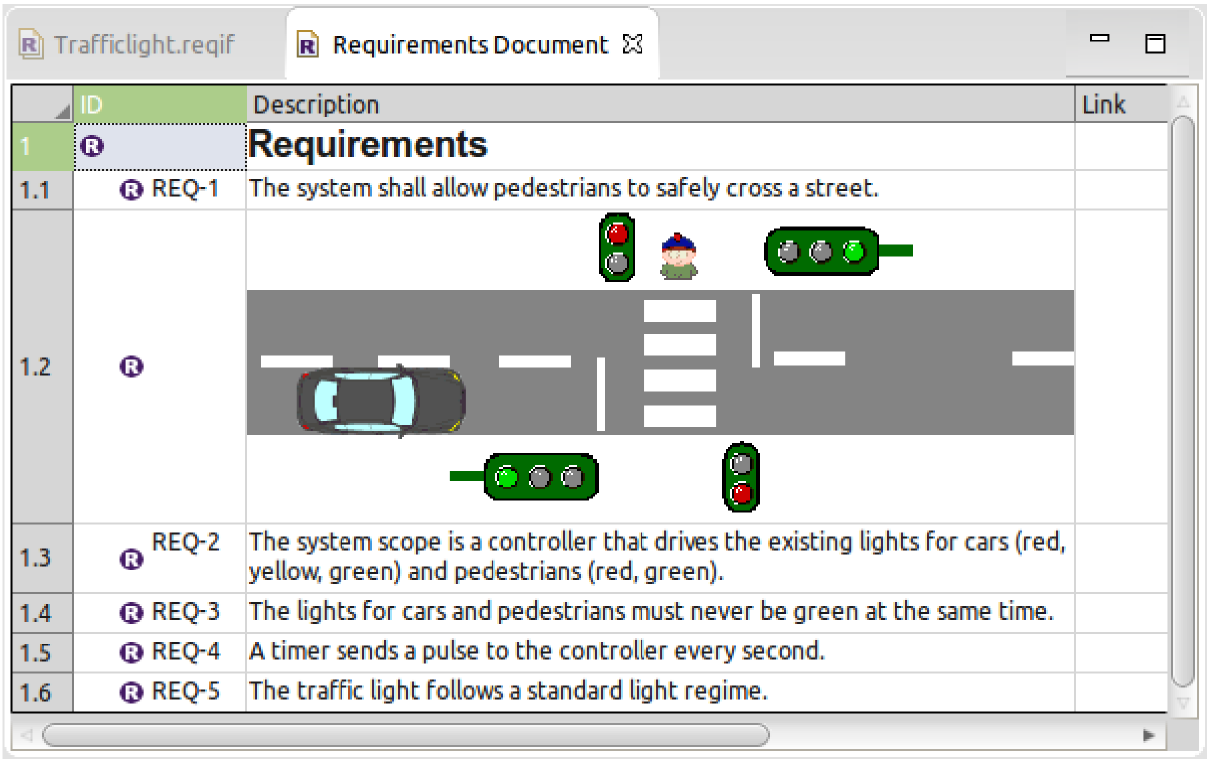
Data Type : String

Figure 2.2: Result: The requirement is now a sibling of the chosen requirement.

After the import, the new requirements have been added to the existing requirements specification. There are a number of recommended improvements, for instance:

- Add a SpecObjectType for Headlines and configure the Headline Presentation, so that you can structure the text
- Once you create headlines, you can arrange requirements as child elements (instead of siblings) under them.
- You can create an information SpecObjectType, using XHTML and no IDs. Use one of these to insert Figure 2.1 into your specification (trafficlight.png).
- Configure the ID Presentation to automatically create IDs for requirements, and center-align the ID.
- Use the ID as a label (if available) by adjusting the Label Configuration.

The resulting specification is shown in Figure 2.3.



	ID	Description	Link
1		**Requirements**	
1.1	REQ-1	The system shall allow pedestrians to safely cross a street.	
1.2			
1.3	REQ-2	The system scope is a controller that drives the existing lights for cars (red, yellow, green) and pedestrians (red, green).	
1.4	REQ-3	The lights for cars and pedestrians must never be green at the same time.	
1.5	REQ-4	A timer sends a pulse to the controller every second.	
1.6	REQ-5	The traffic light follows a standard light regime.	

Figure 2.3: The spec after completion of all steps so far.

2.4 Glossary

A glossary helps keeping track of terminology. In this section, the glossary management from formalmind Studio is introduced, which supports color highlighting in the requirements text.

Note that this kind of glossary is a "dead end", in the sense that it cannot be used beyond its purpose. Contrast that with a model-based data dictionary, as described in Section 2.5.

The glossary is kind of cumbersome to configure. Therefore, we included a correctly configured Sample Project. Note that you need both the Highlighting and Keyword Highlighting presentations, in that order.

Figure 2.4 shows the glossary, and its application to the requirements, which have been rewritten to use the terminology of the specification.

In the screenshot, REQ-3 is being edited. This results in the word *green* being underlined in red, indicating that it is a recognized glossary entry. The syntax highlighting diapears when not in edit mode. Square brackets make a glossary term explicit. If a term is marked that way that is not in the glossary, then it is shown in red.

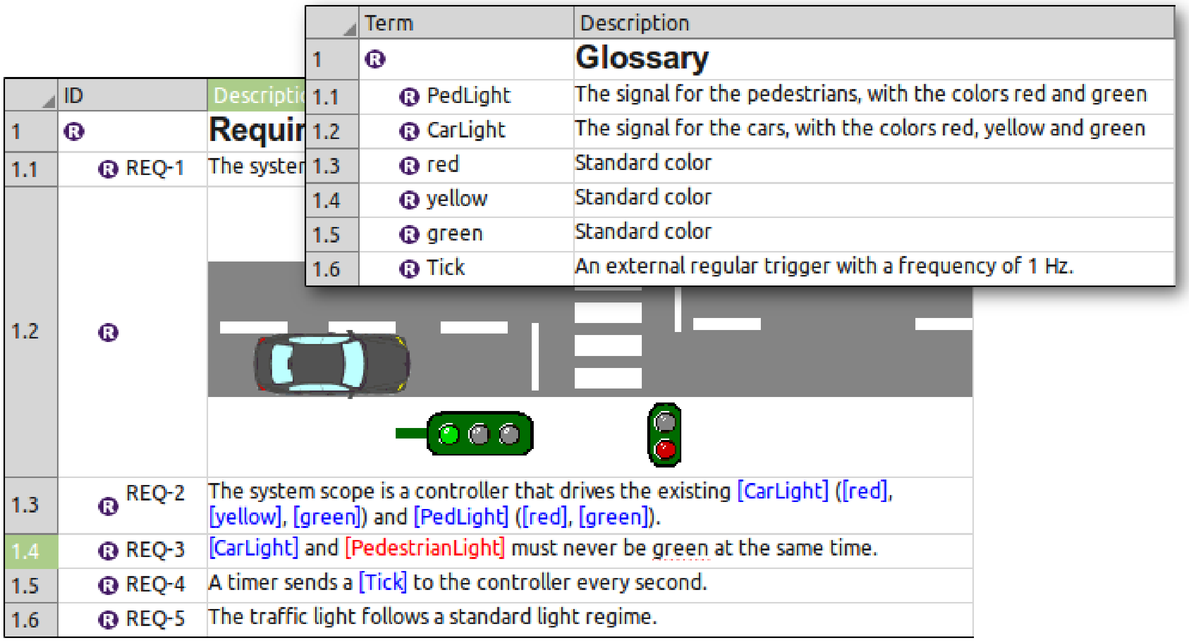
2.5 Data Dictionary with Ecore

Ecore is the modeling language of the Eclipse Modeling Framework. It has some similarities to UML Class diagrams, and is therefore well-suited for creating a precise data model. It has the following advantages:

Easy to learn. Especially if you already know class diagrams, you should be able to quickly learn Ecore.

Code generation. EMF allow the generation of Java code from Ecore models. You can even generate a GUI based on a tree view.

Test stub generation. EMF allows the generation of test code stubs, making it easy to cover the unit test level.



	Term	Description
1	Ⓡ	**Glossary**
1.1	Ⓡ PedLight	The signal for the pedestrians, with the colors red and green
1.2	Ⓡ CarLight	The signal for the cars, with the colors red, yellow and green
1.3	Ⓡ red	Standard color
1.4	Ⓡ yellow	Standard color
1.5	Ⓡ green	Standard color
1.6	Ⓡ Tick	An external regular trigger with a frequency of 1 Hz.

	ID	Descripti
1	Ⓡ	**Requir**
1.1	Ⓡ REQ-1	The syster
1.2	Ⓡ	
1.3	Ⓡ REQ-2	The system scope is a controller that drives the existing [CarLight] ([red], [yellow], [green]) and [PedLight] ([red], [green]).
1.4	Ⓡ REQ-3	[CarLight] and [PedestrianLight] must never be green at the same time.
1.5	Ⓡ REQ-4	A timer sends a [Tick] to the controller every second.
1.6	Ⓡ REQ-5	The traffic light follows a standard light regime.

Figure 2.4: Glossary Management in action.

On the other hand, it has its limitations. In particular, it is not really possible to model dynamic aspects of the system.

Warning. While it is possible to mix this approach with the glossary management described in Section 2.4, we do not recommend it, as it would lead to redundancy. Redundancies should be avoided (DRY-principle: Don't Repeat Yourself).

2.5.1 Creating the Ecore Model

We recommend to create a new Ecore Modeling Project via `File | New | Project... | Eclipse Modeling Framework | Ecore Modeling Project`. This way, everything will be properly configured for code generation and other cool stuff. The model we use is shown in the right pane of Figure 2.5.

As you can see, the editors are arranged so that the requirements editor and the Ecore editor are visible at the same time. This is necessary, as links are created by dragging model elements from the Ecore model onto the requirements.

Information. Linking via Drag and Drop is a feature taken from the openETCS project, where it is documented.

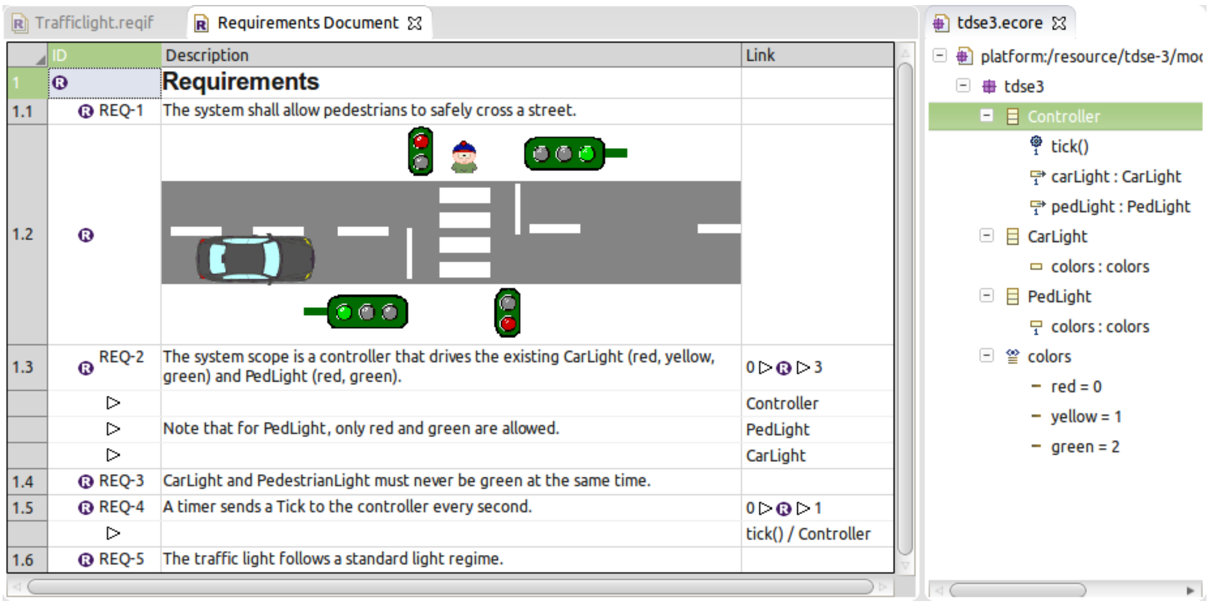
We provided a preconfigured Sample Project, that allows annotating traces, as also shown in Figure 2.5 (the second link of REQ-2).

2.5.2 Working with Diagrams

Some people prefer diagrams to the tree-view shown in Figure 2.5, and diagrams can make communication easier. If you installed the tool as described in Section 2.2, then you can create a diagram from the Ecore model as described here. Diagram and model will be synchronized, but it's possible to only show a subset of the model elements in the diagram.

A diagram should already have been created upon project creation. You can open it by going to the Model Explorer or Project Explorer, and opening the .ecore model by clicking on the

+



	ID	Description	Link
1	R	Requirements	
1.1	REQ-1	The system shall allow pedestrians to safely cross a street.	
1.2	R		
1.3	REQ-2	The system scope is a controller that drives the existing CarLight (red, yellow, green) and PedLight (red, green).	0 ▷ R ▷ 3
	▷		Controller
	▷	Note that for PedLight, only red and green are allowed.	PedLight
	▷		CarLight
1.4	REQ-3	CarLight and PedestrianLight must never be green at the same time.	
1.5	REQ-4	A timer sends a Tick to the controller every second.	0 ▷ R ▷ 1
	▷		tick() / Controller
1.6	REQ-5	The traffic light follows a standard light regime.	

Figure 2.5: On the left the requirements with links into the Ecore-based data model, shown on the right.

to the left of the file name. it should show the package name (tdse3), and upon opening it again, it should unveal "tdse3 class diagram". Doubleclicking should open the diagram editor, which will be empty, except instructions on how to add elements.

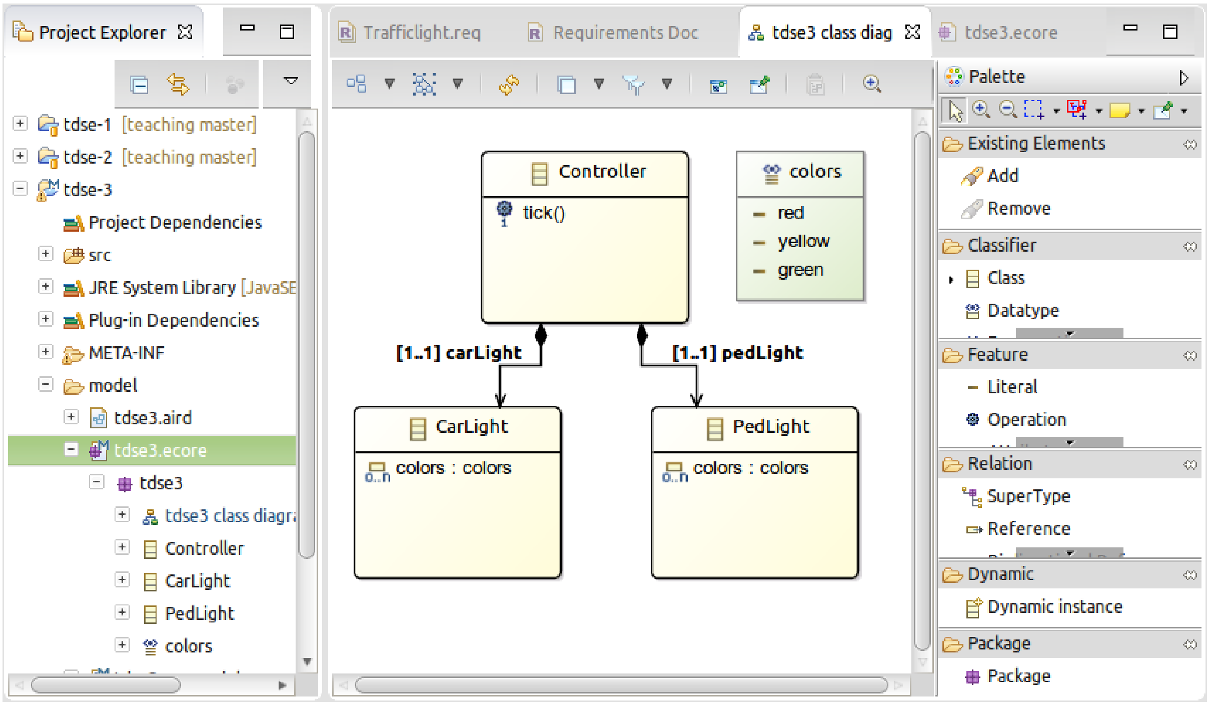


Figure 2.6: The diagram has been created by dragging elements from the Project Explorer (left) to the drawing area (middle). New elements can also be created by using the pallet on the right.

By dragging elements from the project explorer into the diagram area, we end up with the editor, as shown in Figure 2.6. Elements and labels can be customized as one sees fit. Changes to the model, including the creation of new elements, will be reflected in the tree view of the Ecore model as well. Changes to the Ecore model will be seen here, but newly created elements will not appear on the diagram by default. They have to be added manually.

2.6 Traceability

In the previous section, we saw for the first time links (or traces) in ProR. Traces are useful for many things, including change management.

Traces can are typed and can have an arbitrary number of attributes. In ProR, traces can be created via drag and drop, or via context menu. The latter is recommended, as it makes it easy to link multiple requirements at the same time, and allows to set the type with one click, as shown in Figure 2.7.

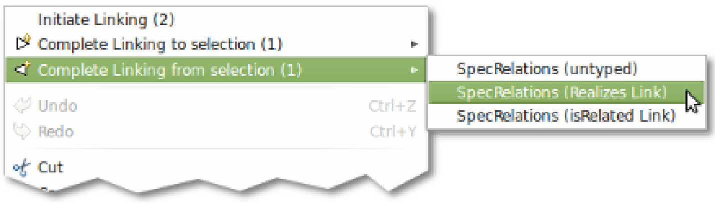
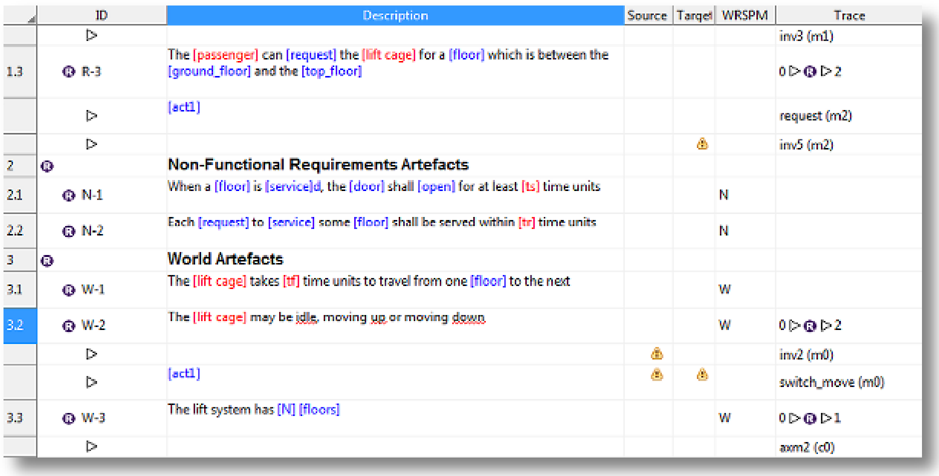

Initiate Linking (2)
Complete Linking to selection (1)
Complete Linking from selection (1)
Undo Ctrl+Z
Redo Ctrl+Y
Cut
SpecRelations (untyped)
SpecRelations (Realizes Link)
SpecRelations (isRelated Link)

Figure 2.7: Completing a link operation.

Information. We wrote an article in the Formal Mind Blog on how to create links in detail.

2.6.1 Change Management

With change management, you want to know if a relationship has to be re-evaluated. There is a free extension available for this purpose, as part of the Formal Mind Essentials (which should be installed, if you followed the installation instructions). With this extension, two flags on the link are set if source or target change, respectively. This is shown in Figure 2.8.



	ID	Description	Source	Target	WRSPM	Trace
	▷					inv3 (m1)
1.3	Ⓡ R-3	The [passenger] can [request] the [lift cage] for a [floor] which is between the [ground_floor] and the [top_floor]				0 ▷ Ⓡ ▷ 2
	▷	[act1]				request (m2)
	▷			⚠		inv5 (m2)
2	Ⓡ	**Non-Functional Requirements Artefacts**				
2.1	Ⓡ N-1	When a [floor] is [service]d, the [door] shall [open] for at least [ts] time units			N	
2.2	Ⓡ N-2	Each [request] to [service] some [floor] shall be served within [tr] time units			N	
3	Ⓡ	**World Artefacts**				
3.1	Ⓡ W-1	The [lift cage] takes [tf] time units to travel from one [floor] to the next			W	
3.2	Ⓡ W-2	The [lift cage] may be idle, moving up or moving down			W	0 ▷ Ⓡ ▷ 2
	▷		⚠			inv2 (m0)
	▷	[act1]	⚠	⚠		switch_move (m0)
3.3	Ⓡ W-3	The lift system has [N] [floors]			W	0 ▷ Ⓡ ▷ 1
	▷					axm2 (c0)

Figure 2.8: When source or target of a link change, a corresponding flag is set on the link.

This mechanism also works with external models, e.g. the traceability the the Ecore model that was described in Section 2.5.

2.7 Modeling with SysML and Papyrus

An alternative to the RMF/Ecore approach is to use SysML with Papyrus. SysML has a requirements type, which would allow covering requirements and model in one overarching model. Further, SysML contains Sequence Diagrams, Activity Diagrams and State Machines, thereby supporting the modeling of dynamic behavior as well.

To use SysML, create a SysML Project via `File | New | Project... | Papyrus | SysML Project`. When asked for the diagrams to be created, sselect the "Block Definition Diagram" (roughly equivalent to a Class Diagram) and the "Requirements Diagram".

We will explore this further in the future, but consider SysML to be overkill at this point.

2.8 Code Generation with Ecore

The Ecore model we created in Section 2.5 can be used to generate a number of different pieces of code, by default Java:

Model Code. This is fairly simple code that manages getters and setters, relationships, persistence, notifications, etc.

GUI Code. This consists of an "Edit" model that can drive a user interface, but does not contain GUI code itself. A simple reflective user interface can be created as well, which looks very much like the Tree-based Ecore editor.

Test Code. The framework generates stubs for unit tests, which have to be implemented manually.

The generation is driven by a "Generator Model", which has already been created when we built the Ecore model. It resides in the same folder.

2.8.1 Model Code

We generate the Model code via Right-Click on the top-level element of the generator model and select `Generate Model Code`.

The code will be generated in the same project. Let's implement a little program for testing:

```
public static void main(String[] args) {
        Controller controller = Tdse3Factory.eINSTANCE.createController();
        while (true) {
                print(controller);
                controller.tick();
        }
}

private static void print(Controller controller) {
        System.out.println("cars: " + controller.getCarLight().getColors() +
                        "peds: " + controller.getPedLight().getColors());
}
```

Of course, this program will not run as is: the lights will be null, the method tick() has not been implemented, etc. This is left as an exercise to the reader.

2.9 Unit Testing

Similar to the generation of the model code, you can generate stubs for unit tests. Right-Click on the top-level element of the generator model and select `Generate Model Code`.

By default, the test code is created in a separate project. The inheritance hierarchy of the test classes reflects those of the model. Each class sets up and tears down a "fixture", an instance of the class under test.

Tests are also inherited: If you implement a test for a superclass in the corresponding unit test file, using the fixture, then the same test will be executed for the derived classes as well.

2.9.1 Mocking

Even at the unit test level, you may be confronted with the fact that the “real” thing to be tested is not available. This is particularly relevant for systems. Consider a software system, for instance, that relies on a web API to operate. On the unit test level, you would typically not want to communicate with the real API. It would be hard to determine whether the API is down or the test broken.

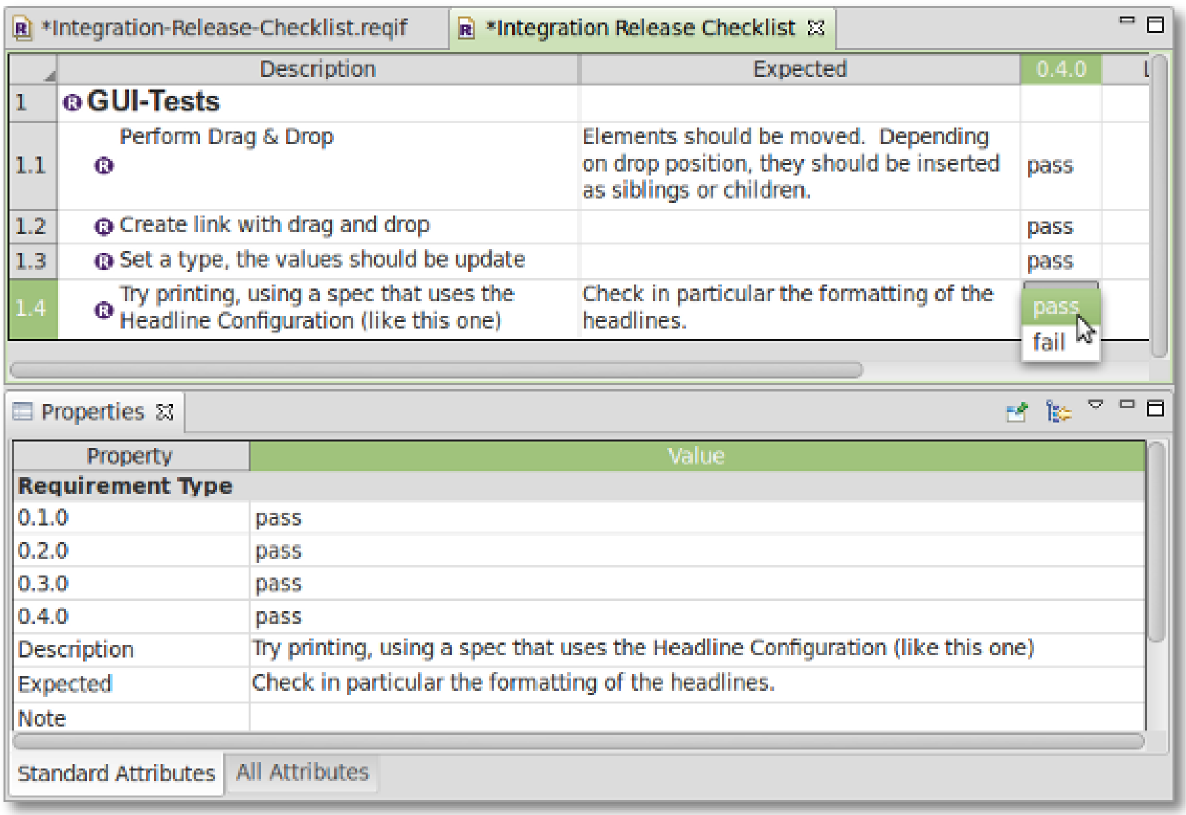
The solution for this is “mocking”: Writing code that mimics the real thing, but only those aspects relevant for the test. Conveniently, mocking also allows testing a service that misbehaves, allowing fault and error conditions to be tested as well.

There is a wide array of mocking frameworks available. We had good experience with JMockit, but there are many others.

2.9.2 Functional and Integration Testing

For the scope covered in this document, there is nothing between unit tests and acceptance tests. But in any project of non-trivial size, and in particular for systems engineering projects, there are functional and integration tests.

If a test can be automated, then it can typically still be realized by leveraging JUnit. In particular, an integration test can be quite similar to a unit test, except that it does not use mocking, but “the real thing”. This can be an API of a service that is called in real time, or it can be real hardware that responds.


*Integration-Release-Checklist.reqif
*Integration Release Checklist
Properties
Standard Attributes
All Attributes

	Description	Expected	0.4.0
1	GUI-Tests		
1.1	Perform Drag & Drop	Elements should be moved. Depending on drop position, they should be inserted as siblings or children.	pass
1.2	Create link with drag and drop		pass
1.3	Set a type, the values should be update		pass
1.4	Try printing, using a spec that uses the Headline Configuration (like this one)	Check in particular the formatting of the headlines.	pass / fail

Property	Value
Requirement Type	
0.1.0	pass
0.2.0	pass
0.3.0	pass
0.4.0	pass
Description	Try printing, using a spec that uses the Headline Configuration (like this one)
Expected	Check in particular the formatting of the headlines.
Note	

Figure 2.9: The diagram has been created by dragging elements from the Project Explorer (left) to the drawing area (middle). New elements can also be created by using the pallet on the right.

2.10 Acceptance Testing

With acceptance testing, we reach the top of the V again. Acceptance tests are hard to automate. There may be scenarios where it is possible, and then some techniques from the previous section could be employed. But more often than not, they have to be performed manually. In this section, we use ProR/RMF again to implement manual testing.

Information. There is a more elaborate description of this technique in the Formal Mind Blog

2.10.1 A Test Result Specification

Tests are defined in a separate document that is part of the same .reqif file. Corresponding to Section 2.3, the document can be styled with headlines, sections, etc.

But the important thing is to create an enumeration data type for recording the tests results, typically "pass" and "fail".

When the time comes to run a test, the tester creates a new attribute for the version under test. Now, each test can be performed, and the result recorded, as shown in Figure 2.9.

The nice thing is, that old tests will be preserved. The main view (Specification Editor) can be configured to only show the version of interest, but in the properties view, all past test results are still there to be inspected.