Graphical Editor for Diagnostic Method Development

by

Sudharshan Ravi and Quang Vu

LIU-IDA/LITH-EX-A--14/032--SE

2014-06-04
Final Thesis

Graphical Editor for Diagnostic Method Development

by

Sudharshan Ravi and Quang Vu

LIU-IDA/LITH-EX-A--14/032--SE

2014-06-04

Supervisor: Anders Fröberg (IDA), Jakob Palmheden (Scania), Erik Thorstenson (Scania)
Examiner: Erik Berglund
Abstract

The adage *A picture is worth a thousand words* conveys the notion that a complex concept can be understood with just a single picture. Thus visualising data allows users to absorb and use large amounts of data quickly. Although textual programming is widely used, it is not best suited for all situations. Some of these situations require a graphical way to program data.

This thesis investigates the different modeling frameworks available within the Eclipse ecosystem that allow the reuse of existing XML schema models and the creation as well as editing of diagnostic methods. The chosen frameworks were used to build a graphical editor that allows users to create, edit and use diagnostic methods graphically.
# Contents

1 Introduction .................................................. 4  
  1.1 Background .................................................. 4  
  1.1.1 Scania AB .................................................. 4  
  1.1.2 SDP3 - Scania Diagnos & Programmer 3 ............... 5  
  1.1.3 PT - Production Tool ..................................... 5  
  1.1.4 Diagnostic Method & Model ............................... 5  
  1.1.5 XML Editor - XMetaL ..................................... 5  
  1.1.6 Current Visualisation Software ............................ 6  
  1.2 Problem Description ......................................... 6  
  1.2.1 Current Visualisation Software ............................ 6  
  1.2.2 XMetaL .................................................... 7  
  1.3 Goal .......................................................... 8  
  1.4 Motivation .................................................... 8  
  1.5 Approach ...................................................... 9  
  1.6 Scope and Limitations ....................................... 9  

2 Theory ......................................................... 10  
  2.1 Metamodel-based transformation ............................. 10  
  2.2 Domain-specific editor ....................................... 11  
  2.2.1 Interface Design Guidelines .............................. 11  

3 The Chosen Frameworks .............................................. 14  
  3.1 Eclipse ...................................................... 14  
  3.2 JAXB ........................................................ 16  
  3.3 Parsers ....................................................... 17  
  3.3.1 Document Object Model - DOM ............................ 17  
  3.3.2 Simple API for XML Parsing - SAX ....................... 18  
  3.4 EMF ........................................................ 18  
  3.5 Graphiti ..................................................... 20  
  3.6 KIELER ....................................................... 22  

4 Implementation .................................................. 25  
  4.1 Import & Export .............................................. 26  
  4.1.1 Import of XML Schema .................................... 26
## CONTENTS

4.1.2 Import of Existing XML files ........................................... 28  
4.1.3 Diagram file ................................................................. 28  
4.2 Visualisation ................................................................. 29  
4.2.1 Editor Interface ......................................................... 29  
4.2.2 Data Visualisation ..................................................... 33  
4.3 Editing .......................................................... 38  
4.3.1 Palette tool box ...................................................... 39  
4.3.2 Button pad ............................................................. 40  
4.3.3 Property sheet ....................................................... 40  
4.3.4 Remove and Delete ................................................... 41  
4.3.5 Copy and Paste ....................................................... 42  
4.3.6 Update feature and Consistency Check ......................... 42  
4.4 Results .......................................................... 43  
5 Conclusion .......................................................... 44  
5.1 Implementation ......................................................... 44  
5.1.1 Interviews and Paper Prototyping ............................. 44  
5.1.2 Test implementation .............................................. 44  
5.2 Results .......................................................... 45  
5.2.1 Design Changes .................................................... 46  
5.3 Future Work .......................................................... 46  
5.3.1 Diagram Types and Views ....................................... 47  
5.3.2 Clean up of Eclipse plugin & Integration with PT ........ 47  
5.3.3 Validation of diagnostic methods ............................. 47  
5.3.4 Testing ............................................................. 48
Glossary

Diagnostic Method Scania’s troubleshooting instruction file for used with diagnostic program, specifying exact input and output as well as possible actions to take to help technicians finding faults in Scania’s products.


Essential Meta Object Facility (EMOF) Is a subset of OMG’s Meta Object Facility (a standard for describing how a metamodel should be built) that closely corresponds to the facilities found in object oriented programming language and XML.

Metamodel Is a model expressing a language to describe other models. For example, if a house is a model, then the roof, room and wall together are its metamodel.

Model In the context of this thesis, a model is a high level description of a data structure from which an instance can be generated.

Model Instance Is a specific instance of a model. For example, if Volvo car is a model then Volvo S60 is one model instance of this model.

Pictogram Element Common name for any visual element in the editor.

XML XML is a markup language that is used to store structured data.

XML Schema Definition (XSD) XML Schema is a language for expressing constraints about data and structured contents in XML documents. XML Schema defines a high-level model of constraints that XML documents must conform to in order to be called valid.

XML Schema Model A model described in XML Schema Definition / Metamodle.
List of Figures

1.1 XMetal ............................................. 6
1.2 Current Visualisation Software ...................... 7

2.1 Metamodel-based transformation[16] .................. 10

3.1 Overview of Chosen Frameworks ..................... 15
3.2 The Eclipse Platform ................................ 15
3.3 JAXB Binding Process ................................ 17
3.4 Oracle DOM API ...................................... 17
3.5 EMF unifies XML, UML and Java ..................... 18
3.6 EMF Three-tier ....................................... 19
3.8 ResourceSet, Resource, EObject ...................... 21
3.9 Graphiti ............................................. 23
3.10 Kieler Infrastructure for Meta Layout ............... 24

4.1 Construction Steps ................................... 25
4.2 EMF Metamodel ....................................... 26
4.3 XML Schema .......................................... 27
4.4 Imported Ecore Class .................................. 27
4.5 XMetaL ............................................... 30
4.6 Paper Prototype ....................................... 30
4.7 Navigator View ....................................... 31
4.8 Editor View .......................................... 32
4.9 Properties View ...................................... 32
4.10 Miniature View ....................................... 33
4.11 Diagram Type Agent .................................. 34
4.12 Pictogram Hierarchy model ......................... 34
4.13 Outer Container Shape .............................. 35
4.14 Attribute ............................................ 35
4.15 Inner Container Shape .............................. 36
4.16 Connection ........................................... 36
4.17 By-name Connection .................................. 37
4.18 Graph Layout ........................................ 38
4.19 Palette Tool Box ............................................. 39
4.20 Button Pad .................................................. 40
4.21 Property Sheet .............................................. 41
4.22 Object Delete .............................................. 42
4.23 Attribute Delete .......................................... 42
4.24 Graphical Editor ......................................... 43

5.1 Paper Prototype Delete ................................. 46
5.2 Implemented Delete ...................................... 46
Chapter 1

Introduction

Although textual programming is widely used, it is not best suited for all situations. Some of these situations require a graphical way to program data. To begin with, an introduction and background to the project task is given. This is followed by the theory of graphical editor and its program structure is discussed in relation to the task. The next chapter describes the methods and frameworks that are used in designing and building the graphical editor. The chapter Implementation presents the results of the thesis work. Lastly, conclusions are made, and future work possibilities are presented.

1.1 Background

In this section we provide an introduction to the thesis in order to give the reader a better understanding of technologies currently in use, the problems with the existing visualisation software and the issues that need be addressed in a new graphical editor.

1.1.1 Scania AB

Scania AB is a large international company that manufactures heavy vehicles (such as trucks and buses) and diesel engines for heavy vehicles, marine and general industrial applications. This thesis was performed at Scania’s head office located in Södertälje, Sweden. Scania provides a host of support services and software to complement its main products and one of these is the Scania Diagnos & Programmer 3 or SDP3.
1.1.2 SDP3 - Scania Diagnos & Programmer 3

Scania Diagnos & Programmer 3 (Scania SDP3) is a diagnostic software tool used in conjunction with the Vehicle Communication Interface (VCI2) tool for the maintenance of Scania buses, trucks and other heavy vehicles. It is used for fault code reading, assisted troubleshooting, parameter setting and software updating.

1.1.3 PT - Production Tool

SDP3 Production Tool (PT) is an Eclipse RCP application used for diagnostic method development. It is used to produce and integrate information needed by SDP3 for troubleshooting and configuration of the electronic systems in a vehicle by giving method developers and documentation writers the ability to collect large amounts of data and to describe diagnostic methods and information for Scania’s products.

1.1.4 Diagnostic Method & Model

For building different step-by-step wizards for aiding technicians when diagnosing products in Scania’s product line, for example, an engine or an exhaust system, a diagnostic method XML file is needed by SDP3. There are many variations of diagnostic methods for different purposes but typically a diagnostic method has the same structure as any XML file with contents resembling a flow-diagram, consisting of different steps and conditions that lead to different outcomes.

To make sure all diagnostic methods are written according to a specific structure, Scania has defined a handful of XML Schema models available in PT that each diagnostic method has to conform to before they can be deployed.

1.1.5 XML Editor - XMetal

XMetal is the XML editor that is used by method developers to write the data and diagnostic methods. XMetal has a user interface similar to a word processor with an editing area where users write XML code and side palettes which show elements and attributes that can be added to the document.

Features such as a spellchecker, a thesaurus and the ability to track changes made to a documents allow method developers who are not familiar with
XML syntax to create diagnostic methods without a lot of prior training or knowledge.

1.1.6 Current Visualisation Software

The current visualisation software was built in order to visualise diagnostic methods and provide an overview of these methods to method developers. The current visualisation software is an integrated part of PT.

1.2 Problem Description

At the present, method developers and writers at Scania use XMetal to write diagnostic methods and the current visualisation software to visualise the diagnostic methods in a UML-like graphical diagram. However, these two programs have a lot of flaws and leave much to be desired.

1.2.1 Current Visualisation Software

- Is only a graphical viewer which means that it is not possible to edit the contents of the diagram in any way.
1.2. PROBLEM DESCRIPTION

CHAPTER 1. INTRODUCTION

Figure 1.2: Current Visualisation Software

- Visualizes objects in much finer detail than necessary and since there is no way to alter how objects are visualized, the diagram quickly becomes overpopulated with uninteresting data.

- Once the objects are drawn, they cannot be altered. This means that the layout of the objects is static and cannot be customized.

- Differentiates between class and relations and visualises them differently but these visual representations often contain many overlapping and crossing of edges which makes it hard to differentiate between the visual information in the diagram especially as the number of nodes increases.

- Is based on Eclipse Graphical Editing Framework (GEF) technology with poor documentation that hinders future development. Furthermore, Scania does not want to continue development of this software.

1.2.2 XMetaL

XMetaL is a lightweight and easy to use XML editor however like all commercial text editors it has some fundamental drawbacks.

- XMetaL is a commercial and closed program with a non-configurable
API that does not allow developers to freely modify and add new functionality to the editor.

- XMetaL is designed for general purpose XML editing and not specifically for developing diagnostic methods. This makes a lot of its functionality unnecessary and confusing.
- Since XMetaL is a textual editor, visualization of relations between objects as well as the flow of data can not be achieved which make it very hard to develop larger and more complex diagnostic methods.

1.3 Goal

This master thesis focuses on answering the following question:

- Is it possible to build a graphical editor from available modeling frameworks within the Eclipse ecosystem that allow the reuse of existing XML schema models and editing of diagnostic methods?
- If it is possible and there does exist a framework or a set of frameworks which allow the realization of the desired editor, Scania would be interested to know in detail how the graphical editor can be built using the chosen framework(s), their strengths and weaknesses and if possible everything is best described in the form of a test implementation.
- If there does not exist any suitable framework(s) for realizing such an editor, Scania would be interested in finding out the limitations in their existing modelling language as well as software and suggestions for any changes that would make it possible to build one in the near future.

1.4 Motivation

Scania wants its employees to work in the most efficient manner possible using tools that improve their effectiveness and deliver results of high quality. The current way to write diagnostic methods is complicated, hard to learn by novice developers and incomprehensible by anyone other than the method developers themselves. Scania wants a different way to program diagnostic methods that will improve speed, quality and increase collaboration between all people involved and strongly believe that a graphical editor is the solution.
1.5 Approach

We will first conduct an investigation of available frameworks and choose one or a set of technical solutions available for the Eclipse platform that together can be used for solving the different building blocks necessary for construction of a domain-specific graphical editor.

If the available frameworks can be used to realize an editor then we will build a test implementation that demonstrates how the frameworks work together, their role, importance and how an editor can be built in such a way that satisfies the needs of the users through well elaborated interviews and paper prototyping.

If there does not exist any framework(s) or if it is not possible to build an editor based on current model infrastructure, then reasons for this will be outlined as well as suggestions of possible changes to overcome these obstacles.

1.6 Scope and Limitations

In this thesis, we are limited to working within the Eclipse ecosystem which means that we will only focus on frameworks that support this platform.

Due to time constraints, this thesis will not focus on solutions that will require us to develop everything from scratch as this will put it far beyond the scope of this thesis. Thus framework solutions will be preferred whenever possible for quicker development of the required application.

We will only look into open and configurable solutions since these solutions allow developers to freely customize and extend new functionality.
Chapter 2

Theory

2.1 Metamodel-based transformation

Metamodel-based transformation is a process of transforming a specific model from one metamodel specification to another. During the construction of the editor, it is vital that the existing XML Schema model can be transformed into either Java classes or other metamodel specifications derived from Java for easier access and instantiation. The task of transforming one model to another can often be described as shown in Figure 2.1.

While Scania’s model is quite simple and resembles what we would normally see in an UML class diagram, the XML Schema metamodel used for describing them is very comprehensive. Hence working with XML Schema model directly is not a good approach. In order to be able to visualize data, we need a more uniform metamodel for describing the current model so that every model element can be categorized and visualized correctly based on their common qualities.
2.2 Domain-specific editor

Vlissides [17] defines a domain as a physical or abstract context in which objects are manipulated and an editor as a program that supports the manipulation of objects in a domain. Currently, the process of editing diagnostic methods means working directly with the serialized XML file. We want to achieve a higher level of abstraction that allows the user to modify the domain data through a simpler visual model.

Building a user-friendly editor is not a simple task as it is easy to misunderstand the needs and wishes of the user. In this section, we will explain some fundamental interface design principles that should be followed as guidelines during construction of the user interface and describe how paper prototyping can aid in the process of understanding the user and verifying the design of the user interface.

2.2.1 Interface Design Guidelines

Designing the user interface for the graphical editor is an important but not simple task. [13] states that users perceive what they expect, and thus user interfaces should be designed with this in mind. According to [13], there are four fundamental guidelines that should be followed while designing the interface; Understand the goals, Avoid ambiguity, Be consistent and Familiarity[13].

Understand the goals

Understanding the users’ goals is the first thing that should be performed at the start of user interface design. To this end, conducting interviews with users is a good way to get an understanding of their needs and wishes.

Avoid ambiguity

In order to avoid ambiguity that can lead to users making errors while using the system, all the interaction and functionality in the system should do what it states. For example, Remove and Delete should have clear distinction between what they do. This is an important thing to get right as the whole system will be unusable otherwise.

Be consistent

It is important to be consistent in all parts of the system. [13] states that
information and controls should be placed in distinct locations, controls and data that serve the same function on different objects should placed in the same position.

**Familiarity**

[13] states that when using a new system, users take familiar paths whenever possible rather than exploring new ones. Thus, by incorporating parts of the existing solution whenever possible will make it easier for existing users to get familiar with the new design.

**Paper prototyping**

It is essential to test and verify the design of the user interface of the editor before implementation. Paper prototypes are better than more complex and polished prototypes or so called hi-fi prototypes due to several reasons:

- Paper prototypes are easy to build and can be constructed much faster when compared to hi-fi prototypes built using multimedia tools or high-level languages. [14]

- The hand-made appearance of paper prototypes makes users think about the content of the interface rather than the appearance. When using hi-fi prototypes, the focus of users can easily be on the appearance of the interface rather than the content. [14]

- Hi-fi prototypes can cause users to have set expectations about the look and feel of the system that will be very hard to change. [14]

Paper prototyping provides a lot of benefits to aid in interface design such extracting vital information from users regarding the interaction and fundamental functionality of the interface as fast and early in the development phase as possible. They also illuminate the designers about flaws in the interface that might be missed in user interviews.

Rettig [14] provides guidelines on how paper prototyping should be conducted with users. These guidelines should be followed in order to successfully perform paper prototyping and get reliable results.

Preparing for the paper prototyping is made up of three steps [14]:

- **Select your users** - recruit users that represent the whole range of characteristics in the target audience.
• **Prepare scenarios** - based on user interviews, design scenarios that focus on a small fundamental set of function but broad enough to be meaningful.

• **Practice** - to remove all the *bugs* in the paper prototype, practice runs should be conducted.

Four people with different roles are recommended to conduct paper prototyping [14]:

• **Greeter** - welcomes users and tries to put them at ease.

• **Facilitator** - only person who is allowed to speak freely during paper prototyping. Does three things; gives users instructions, encourages users to express their thoughts and makes sure everything is done on time.

• **Computer** - simulates the behavior of a computer by moving pieces of the paper prototype around depending on user input.

• **Observer** - observe and take notes about suggestions and changes.

According to Rettig [14], it is good practice to perform paper prototyping in iterations with users as changes are made to the design whenever possible.
Chapter 3

The Chosen Frameworks

In this chapter, we will describe the frameworks that were used and go through some important concepts in this thesis as well as providing the reason why they were chosen and what they can contribute to the construction of our editor. The goal is to give readers who are new to these technologies a gentle introduction to all the frameworks. First, we will talk about the Eclipse development platform for which our implementation of the graphical editor is based on. Then we will continue by explaining in short several frameworks that can be used for import and export of existing data and how EMF fits in the picture. Third, we will talk about the Graphiti plugin that we used for the visualization and domain data manipulation and why KIELER is needed. Figure 3.1 gives an overview of the chosen frameworks and how they work together to form the graphical editor.

3.1 Eclipse

Eclipse is an Java open source software project with the goal to provide a highly integrated tool platform for software developers. Today, Eclipse is a widespread development framework with contributions from many independent and organizational contributors such as Google, IBM and Ericsson among many others. Most developers might only know Eclipse through its famous Software Development Kit (SDK) which includes one of the leading Java development environments (JDE) but not many know that Eclipse by itself without all of it plugins is nothing more than an empty shell, that would look and work very much like a file browser.

In an Eclipse application, the most basic building block is a plugin. The
Eclipse platform itself and all of its tools is a composition of one or many plugins. Each plugin provides specific functionality that can be invoked by the user or reused by other plugins. In Eclipse, a plugin is managed by two components, the OSGI Service Platform which allows it to be installed or removed without restarting the application and the Platform Runtime which manages the lifecycle of the plugin and its interaction with
other plugins [3].

From a packaging point of view, a plugin is a self-contained program containing everything needed for it to be fully functional, such as source files, images and other resources. In the heart of every plugin, there are two important files called META-INF/MANIFEST.MF, and plugin.xml and these specify dependencies and extension capabilities of the plugin among many other things. When integrating different plugins to construct our application, it was through these files the internal dependencies and extensibility between involved plugins was specified and declared.

The reason for choosing Eclipse as development platform was simple. First and foremost, it is free and open-source with a widely installed base of supported plugins. Secondly, the Eclipse modelling project is by far the most comprehensive one available since it provides modeling tools for developers, with support from both commercial and academic organizations from all over the world. In addition to this, Eclipse’s support community is invaluable, especially for getting in contact with people directly involved with the framework in question which was done many times during this thesis[3].

3.2 JAXB

JAXB which stands for Java Architecture for XML Binding is a lightweight JAVA framework that allows developers to map Java classes to XML data through a process called marshalling and from XML data back to Java classes called unmarshalling. To be able to read XML data JAXB generate java classes based on the corresponding XML Schema model and create data structures necessary for manipulating that XML. After all necessary classes have been generated JAXB can be used for both reading data from XML files to Java objects as well as writing Java objects back to XML accordingly.

However using JAXB would mean that all XML model classes have to be generated before it can be used, while this process only required once per schema the number of generated classes could be very high. Changes made to the XML schema would require corresponding changes to the generated Java classes and a recompilation of the application. As the result of these drawbacks JAXB were not considered for this thesis.
3.3 Parsers

3.3.1 Document Object Model - DOM

Another way to extract XML data would be to use DOM parser to gather all necessary information and construct Java models classes necessary to build the editor.

As depicted in figure 3.4 a DOM parser reads XML data and represents the whole document in a in-memory tree that can later be used to access and traversed generically.
3.3.2 Simple API for XML Parsing - SAX

Unlike DOM, SAX parser is an event based parser that doesn’t require the whole XML file to be loaded into memory before data can be read. The reading process is event-based and triggered when the parser encounter a matching opening tag. This way large files can be read more efficiently compare to DOM parser.

While DOM and SAX parsers offers a very efficient and generic way to read XML data it is not possible to write, manipulate and serialize objects back to XML using these framework. Hence a persistent framework is required to use in conjunction with these parser to cover both import and export. In this thesis DOM and SAX parsers were not considered, however these technologies are very often used in other frameworks, it is beneficial to have a basic understanding about their structure.

3.4 EMF

EMF stands for Eclipse Modeling Framework and is the core of the Eclipse Modeling Project. It is a Modeling, Data Integration & Code generation framework that provides a unified tool for developers that incorporate modeling, code generation and programming in Java into a very low-cost package.

Unlike similar modeling tools, in EMF there is no separation between the high-level modeling and the low-level programming since both are brought together to accomplish the same task. With EMF, one can start by defining the model either from Java interfaces, UML, XML Schema or Ecore.

![Figure 3.5: EMF unifies XML, UML and Java](image-url)
If an application is defined as a multitier architecture, EMF provides a wide range of functionality necessary for creating highly customized applications that can be extended to fulfill any specific needs.

At the Presentation Layer, EMF provides support for accessing data and change notifications to the Business Layer, as well as, redo and undo functionalities through EMF transactions [4].

At the Business layer, EMF provides a set of generic methods defined in the EObject superclass for modifying domain data. Changes to domain data can be invoked through EMF transactions which ensures global consistency and atomicity. Any changes to the data can be validated through the EMF validation framework for inconsistencies against specified business rules, which can be formulated in many different ways. For example, through coding in Java or Object Constraint Language (OCL) [4].

At the Persistence Level, EMF provides support for the serializing and persisting of the domain data in various formats through its EMF Resource concept. It facilitates fast and efficient loading and parsing of XML files with support for lazy loading (proxies) and cross-file references through URI fragments to reduce redundancy of the persisted data [4].

In this thesis, EMF will handle all back-end operations related to accessing and modifying of the domain data as well as loading, parsing and serialization of XML documents.

When working with EMF, there are some key concepts worth knowing.

- **Ecore** is a common metamodel for describing models in EMF. It is based on Essential Meta Object Facility (EMOF) which in most ways resembles how classes are defined in a UML class diagram.

The overall hierarchy of Ecore can be seen in figure 3.7.

- **EClass** represents a modeled class or type. It have a name, zero or more attributes and zero or more references.
• *EAttribute* represents a modeled typed attribute. It has a name and a type.

• *EReference* represents an association end between classes. It has a name, a boolean flag indicating containment and a reference to the target class.

• *EObject* is the root of all modeled objects. When loading XML files with EMF all of its contents are translated into EObjects.

• *Resource* is the container for loading and persisting of related document. A resource is typically contained by a resource set and is identified by its file location URI.

• *Resource Set* is a container for holding all related persistent documents.

• *URI fragment* is an optional address fragment to the file URI for random access objects inside a resource similar to XPath[4].

### 3.5 Graphiti

While EMF provides a lot of functionality for importing and editing model data, it does not provide an intuitive way for visualizing and editing the visualized data. However, the default generated code produced by the
EMF Code Generation Framework does provide a simple treeview for editing domain data but it is quite far from what we want to achieve.

Unlike EMF, which is perhaps the only back-end modeling framework available for Eclipse, there are quite a few open source Graphical editor plugins with support for EMF available. The first and biggest candidate is the Graphical Modeling Framework (GMF) backed by IBM and the other is Graphiti backed by SAP[6].

While GMF has been around since 2002 Graphiti is quite new, its initial contribution was from 2010 and is still in the Incubation Phase.

While both are based on the Graphical Editing Framework (GEF) and use the Draw2D graphical library for visualization, there are some fundamental structural differences between the two. The biggest difference is how their respective API is exposed to programmers.

Graphiti hides GEF and Draw2D in its own wrapper classes while GMF extends the existing GEF and Draw2D libraries and exposes it all to the developers. This means GMF provides more flexibility but also adds lot more complexity to its API. On the other hand, Graphiti trades some of the complexity for a more linear learning curve. Another drawback with
GMF is its highly theoretical documentation and complex examples. Unlike GMF, Graphiti provides a much more comprehensive list of concrete examples along with its theoretical documentation which makes it a better suited candidate. In this thesis, Graphiti will serve as the presentation layer responsible for the visualization and editing of domain data as well as the handling of user interaction.

In Graphiti there exists a clear separation between a pictorial object and its domain data. Pictograms (or shapes) reside in the diagram resource while domain data belongs to the XML resource [6]. Any functionality for modifying these resources are divided into features, most noticeably:

- **Add features**: For adding pictorial objects, this feature should only modify the diagram resource.
- **Create features**: For creation of domain data as well as adding a pictorial representation for it. Create features modify both the diagram and the XML resources.
- **Remove feature**: For only removing pictorial objects in the diagram resource.
- **Delete feature**: For removing both the pictorial object and the linked domain data in the diagram and the XML resources.
- **Custom features**: If the above features are not enough, a custom feature can be declared to do more general tasks. For example, set value or renaming.

Features are highly independent modules and Graphiti enforces this notion throughout its API. This makes the process of adding new functionality to Graphiti a very uniform process that can be carry out simultaneously.

### 3.6 KIELER

Although Graphiti provides a way to represent and modify domain data, it does not provide a graph layout functionality for the visualization of XML data. This makes it hard to visualise existing diagnostic methods with no complementing diagram resource especially when the number of visual objects grows to thousands of nodes and relation edges. Normally, to draw a directed graph the following steps are considered [8] [9].

1. **Circle removal** where circles are removed by temporarily remove or reverse some edges in the directed graph.
2. *Layer assignment* where each node is assigned to a depth level depending on different criteria, normally how many out edges it has.

3. *Vertex ordering* where vertices in each layer is ordered to create a good visual symmetry and minimize edge crossings.

4. *Coordinate assignment* Where vertices are assigned normally to minimize the drawing area

5. *Drawing of edges* where edges of the graph is placed on the graph.

Most of these steps are, however, NP-hard (Non-deterministic Polynomial-time hard) and would require good heuristic to achieve acceptable results depending on the situation. To do this properly, extensive evaluation of different layout algorithms needs to be carried out. This is outside the scope of this thesis.

While conducting research on different layout algorithms and solutions for the above problem, an open source research project called KIELER from University of Kiel was found. Just like EMF and Graphiti, KIELER is an Eclipse plugin which can be easily integrated. The internal structure of the KIELER Layout can be described in the following [10].

Here *DiagramLayoutEngine* is a singleton class responsible for analyzing the diagram, generating its own representation KGraph of the graph, configuring and executing the layout algorithms and transferring the changes back to the diagram.
The interaction between KIML and external editors and viewers is handled by LayoutManagers while layout options is handled by LayoutOptionManager which applies a set of Layout Configurators on each node in the graph. Layout Configurators can be configured programmatically or statically through KIELER’s Eclipse plugin extensions point. In this thesis, the former approach is used and KIELER is integrated into Graphiti and configured to handle the layout command from users during use and upon import of existing diagnostic methods.
In this chapter we will explain how the editor was implemented based on the steps depicted in figure 4.1. As mentioned in the previous chapter, EMF was chosen for handling file import and export, we will first explain how the EMF converter has to be setup to support the transformation from XML Schema to Ecore as well as correct serialization format of the output data. Later, we will present the result from our interview and paper prototyping sessions and the proposed design of which our implementation will be based on. Lastly, we will describe in depth how different functionalities in the editor were implemented to support visualization and editing.
4.1 Import & Export

By default EMF is configured to support Ecore import and XMI as default serialization. To change this behaviour the correct Resource Implementation has to be registered to the global resource factory registry. While EMF provides a selection of Resource Implementations for XML and other file types, for our purposes only `GenericXMLResourceImpl` and `XSDResourceFactoryImpl` were used.

4.1.1 Import of XML Schema

For each element in the schema EMF conversion engine will generate one EClass according to EMF’s XSD to Ecore mapping specifications[4]. To make sure every tag is fully captured EXTENDED_META_DATA were manually enabled by default. This option force the engine to store original naming conventions to guarantee correct serialization later in the export process.

After import, an in-memory ecore model package called Epackage will be available for use. The contents of this package consist of EClasses representing the model class corresponding to the Schema. Each EClass might have one or more EStructuralFeatures of either Attribute/EAttribute or reference/EReference. Each EAttribute holds data of EDataType of simple data types such as String or int, while EReference pointing to other EClasses by containment or non-containment references. If the imported XML Schema have more than one root classes which is true in this case EMF will generate an additional class called `DocumentRoot` to contain all first level contents and set it as the root container. Figure 4.2 shows the structure of EMF’s metamodel used for conversion.

![Figure 4.2: EMF Metamodel](image)

Figure 4.2: EMF Metamodel
Figures 4.3 and 4.4 is an example of how XSD to Ecore conversion could look like.

```xml
<xsd:complexType name="CalibrationClass">
  <xsd:sequence>
    <xsd:element name="Calibrate" type="RefToServiceDefinition"/>
    <xsd:element default="1" minOccurs="0" name="IndexInResult" type="xsd:integer"/>
  </xsd:sequence>
  <xsd:annotation>
    <xsd:documentation xml:lang="en-GB">This is a description</xsd:documentation>
  </xsd:annotation>
</xsd:complexType>
```

Figure 4.3: XML Schema

As depicted in figures 4.3 and 4.4 apart from EstructuralFeatures EMF used Extended Meta Data annotations for mapping from XML Schema to Ecore, the main uses of this data model is to attach additional structural and serialization information to Ecore model classes.

Any changes made to the Extended Meta Data in the class mapping will change how the class instance is persisted in the XML output.

In this thesis, all EPackages are registered to the global EPackage registry. By doing so the converted model will become available throughout the application. Another alternative is to register the Ecore model packages locally to the resource set in order to limit their availability only to the enclosed resources. No matter the method chosen, the result is the same.
4.1.2 Import of Existing XML files

When importing existing XML diagnostic files, the Schema namespace location URI is available within the file itself. By default, EMF will look for any available Ecore Epackage in the registry with the matching namespace location URI, first by looking locally in the resource set followed by the global Epackage registry. If no package was found EMF will create an EPackage, register it and then use it to load in the XML resource.

EMF supports load and save options to customize parsing behavior of the resource implementation. In addition to GenericXMLResourceImpl’s own load options, KEEP_DEFAULT_VALUE was set manually to force the resource to persist default values to unset attributes when saved. Otherwise, attributes such as ref (which always have a default value) will be omitted from the XML output. There is a long list of other options that can be set for XML resources for additional customization and efficiency [4]. This can be helpful for optimization purposes in the near future.

After Import, a Graphiti diagram instance is created in the same resource set as the imported XML resource. This is very important that both belong to the same resource since all pictogram elements in the diagram will have direct reference to the actual domain data instances in the XML resource. EMF only allows such linking when both the linker and the linked resources belong to the same resource set.

To visualize the contents of the XML Graphiti’s add features were called to manually traverse the content tree and populate the diagram.

4.1.3 Diagram file

Since both the input XML Schema model and the output diagnostic method XML have to maintain the same content and structure due to backwards compatibility issues, visual information has to be stored in a separate file so it can be used in other sessions. To get around this problem a new file format called UIXML (often referred to as diagram resource) was created for storing such information.

As mentioned earlier, a diagram resource must belong to the same resource set as the XML resource before it can be opened. By default, Graphiti assumes all domain data objects are in the diagram but in this thesis the diagram’s visual data and XML data are kept separated. Thus, it has to be loaded in before the diagram can be opened. It is important to make sure all linking in the diagram is consistent with the XML resource, modifying XML data outside the editor is highly discouraged and should be handled properly. This will be discussed in chapter 4.3.6.
4.2 Visualisation

4.2.1 Editor Interface

In order to cement the design of the user interface of the graphical editor, interviews were conducted with users followed by paper prototyping.

Interviews

Interviews with users were conducted to better understand their needs and wishes. The following questions were posed about the current process in writing diagnostic methods. The interviews were 30-45 minutes.

- Shortly describe the development process in creating a diagnostic method.
- What works well right now? The best functionality?
- What does not work so well right now?
- What current functionality do you want to transfer to the new editor?
- Is there any repetitive behavior? What is good or bad about this?
- How often do you make mistakes? What are these mistakes?

These interviews were conducted with method developers writing diagnostic methods for different vehicle systems. They were very illuminating and provided vital information required in creation of the paper prototypes to further concretize the design of the interface.

Building the Paper Prototype

The user interface design guidelines discussed in [ref] were followed when designing the interface of the editor. User interviews were conducted as per the first guideline Understand the goals. The biggest piece of information gained from these interviews was that the users liked and understood the interface of the XMetaL editor. Thus, as per the guideline Familiarity the design decision was made to use XMetaL as a template for designing the user interface of the graphical editor.

As seen in figure 4.5, XMetaL has a tool palette which dynamically suggests tools that can be added to the document which is based on the selected object. This functionality intelligently reduces the number of tools
shown and thus was appreciated by the users. Thus, this functionality and design has been incorporated into the paper prototype.
As seen in figure 4.6, the paper prototype consists of four views that together form the graphical editor. These four views are the Navigator view, Editor view, Properties view and Miniature view.

**Navigator View**  The Navigator view is a file explorer which catalogs and shows all the files in the file system. It recognizes the filename extensions related to the graphical editor which are .uixml, .xml and .xsd.

- The XML schema file (.xsd) is to generate a new diagnostic method. It creates a .uixml and a .xml file.
- The XML file (.xml) is the where the data of the diagnostic method is stored and can be used to generate a .uixml diagram file.
- The diagram file (.uixml) stores the visual data linked to the data in .xml file.

![Figure 4.7: Navigator View](image)

**Editor View**  The editor view consists of two parts; the editing area and the palette. As seen in the figure below, the editing area makes up a bulk of the view with the palette on the right side.

The editing area is the most important part of the editor, as it is where the diagnostic methods are drawn and shown. Visual objects can be added
to the editing area by dragging them from the palette. To make it easier for the user to access frequently used operations such as layout, delete, copy and paste on visual objects and the editing area, a context button-pad and a context-menu have been added to the interface.

Properties View  To separate the visualization and editing capabilities of the editor, the editor view only visualises the data whereas all modification to it can only be made through the properties view. As seen in the figure below, the properties are shown in a tree structure which can be edited.

Miniature View  The miniature view is used to provide an overview of the diagram. It makes it easy to change the focus of the current editing
area to any part of the diagram, especially when the diagram is large and clipped outside the viewing area.

![Miniature View](image)

Figure 4.10: Miniature View

**Paper Prototyping with Users**

Paper prototyping with users was conducted and lasted 30-45 minutes. The guidelines given by Rettig [14] were followed with some modifications.

Due to limited resources, the four different roles needed for paper prototyping were split between the authors. This change has been applied after careful consideration and preparation in order to minimize any negative effect.

The six users chosen for paper prototyping were method developers from different vehicle systems and products in order to have a diverse target audience as per Rettig [14]. Through paper prototyping, misunderstandings about the design were captured and unexpected behavior was noted. The result was positive and implementation can be carried out with no major changes made to the original design as shown in above figure 4.6.

**4.2.2 Data Visualisation**

For this project only one Diagram type contribution to Graphiti was registered through the editor’s Extension point with its own set of features provided by XMLEditorFeatureProvider. The implementation of Guide Diagram Type will serve two main purposes

1. To provide visualization of imported XML domain data EObjects.
2. To provide basic editing options conforming to the modeled specification.
Our diagram type contains a highly cohesive set of functionalities that rely a great deal on how the data is being visualized. Any modification in the way the pictograms is visualized will most probably require changes in code on many or all underlying features.

**Graphiti visual building blocks**

The most commonly used visual object in Graphiti is called PictogramElement which is an EMF model object. It is the high level of all visual objects that can be seen in figure 4.12. A PictogramElement is by itself a high level container model object for holding two other important container model objects.
1. Graphic algorithm container model which holds dimensional, positional and graphical data specifying how the PictogramElement will look like when the shape is being drawn by the underlying graphic engine.

2. Property container model holds all user-attached data. This makes it possible to attach any kind of String data to be persisted along with the PictogramElement which is located in the diagram resource. This is a very commonly used feature in KIELER in order to add additional information to pictogram elements, especially when they are unlinked.

With all available graphic algorithms, for example, basic shapes and lines, constructing suitable visual representation for the available domain data is very straightforward. Complex domain data objects which are objects that have one or more containment EReferences to other objects will have an independent shape depiction called *Outer Container Shape*.

![Figure 4.13: Outer Container Shape](image)

An *outer container shape* consists of one ContainerShape holding a Rounded Rectangle graphic implementation with no attached properties. An empty outer shape contains one title row consisting of two Shapes (the simplest pictogram element container); one holding an icon and another holding a text field describing the name of the domain object model.

There are three properties attached to these Shapes, a VERTICAL_INDEX value of -2 and a HORIZONTAL_INDEX value of 0 for the icon and 1 for the text field. Next is a line separator consisting of a shape holding a polyline with attached VERTICAL_INDEX value of -1. These index values will be used later for indexing the layout of the shape. Each outer shape is linked with exactly one domain data object after its creation. However, it is possible to link multiple domain objects to one shape and vice versa if there is a need for it.

![Figure 4.14: Attribute](image)

For each EAttribute in a complex domain data object, three shapes are used to visualize its data similar to the title row with an additional text
column but with the attached VERTICAL_INDEX corresponding to its EAttribute ID value. Since EMF sorts all EStructural features in an EObject from top to bottom and assign IDs to all of them in the same order, using the same ID for vertical indexing of rows will guarantee that the order of all the attributes match the order in the Schema model.

![Figure 4.15: Inner Container Shape](image)

For each non-complex domain object, a container called inner shape consisting of one ContainerShape holding a Rectangle graphic implementation and one attribute row was used. Similar to the how EAttributes are visualized, the container is linked to the containment referenced domain data with the attached VERTICAL_INDEX corresponding to its EReference ID.

The creation of each set of shapes mentioned above is handled by AddOuterContainerFeature, AddInnerContainerFeature and AddAttributeFeature respectively.

**Connections**

Since all domain data is modeled with a strong focus on containment relationship, almost all but a handful of objects are contained by another through the use of containment references in a tree like structure similar to what can be found in UML diagrams. In this thesis, this relationship is depicted as a dashed grey arrow pointing from the linked pictogram element belonging to the parent object to the pictogram element belonging to the child object. The name of the reference EStructuralFeature is also attached to the connection to make it easier to identify the relationship.

![Figure 4.16: Connection](image)

Another possible relationship that can occur in the model is the by-name reference where a business object refers to another by the XML *ref* and *value* tags. Ref specifies the type of the referenced domain data type and
value specifies the name of that type’s instance which could be located either in the same XML resource or anywhere in Scania’s database. As of now, only by-name references to domain objects in the same file are visualized. References to files outside of the diagram’s resource set is still to be developed. Similar to containment relationships, by-name relationships are depicted as a solid blue arrow pointing from the pictogram belonging to the referencing object to the matching object with the specified name.

![Figure 4.17: By-name Connection](image)

The creation of connection elements are handled by AddContainmentFeatures and AddByNameReferenceFeature respectively. Unlike inner and outer shape, connections don’t have any linked business object to them.

**Shape Layout and Resize**

For calculating size and arrangement of outer and inner shapes, a visual tree was built based on the vertical and horizontal indexes attached with each shape when layout is being applied. Shapes in the tree are sorted by rows and columns respectively.

Vertically from -2 to any positive number with -2 belonging to the class name, -1 belonging to the line separation row and 0 to any positive number depending on the EStructuralFeature the row is representing.

Horizontally from 0 to 3 for EAttributes and 0 to the total number of instantiated references in one to many containment relationships for inner shapes. The size of the title and line separator is vertically fixed while all other rows are dynamically dependent on both width and height in relation to the outer shape containing them. By default, only outer shapes are resizeable which means that every time the user wants to resize an outer shape by dragging the corner of a shape, the layout feature is called calculate the new size. It is important to maintain this separation for consistent shape layout.
4.3 Editing

In our editor, all add features only modify the diagram resource, while create and custom features can modify both the diagram resource and the XML resource. While modification of the diagram data is handled by Graphiti, modification to the XML resource is only handled by extending AbstractCreateFeature. In this thesis, whenever a feature makes an attempt to modify existing XML domain data, it was done through EMF transactions to fully support concurrency modification and fail rollbacks in
order to guarantee the atomicity of actions as well as supporting undo and redo functionality.

There are different ways to create and modify data in the diagram. Creation of new shapes and attributes is done through the tool palette. All modification to the existing data is done through the Eclipse tabbed properties view. For other system wide actions such as copy, paste and delete the context menu was used. In addition to this, quick access to context specific functionalities such as drag creation of containment and by-name references were also added to context button-pad. The context button-pad belongs to each outer shape has the copy, delete and connection creation features as standard.

4.3.1 Palette tool box

When any outer shape in the diagram is selected, the palette is updated with a list of available creation features based on the model EClass behind the linked business object. Creation of any new shape or attribute generally consists of two steps:

1. Creation of the target business object in the xml resource.

2. Creation of visual representation and linking in the diagram resource by using the appropriate Add feature.

![Figure 4.19: Palette Tool Box](image)

Every creation feature in the tool palette also accesses model data and extracts existing documentation annotated in the model (if there is one) to provide tooltip help text when highlighted.
4.3.2 Button pad

To make it easier to access features that are more frequently used, copy, delete and update buttons were added to each outer shape button-pad. The button-pad is automatically activated when mouse pointer is hovering over any shape in the diagram. Right now button-pad belonging to a shape contains five different buttons, namely copy, delete, update, create containment and create by-name reference. The first three is activated through mouse click while the latter two will initiate a drag action to hint the user to complete the connection creation. If there is more than one possible connection that can be created at drag-finished, a list of possible create connection features will pop-up prompting the user for a decision.

![Figure 4.20: Button Pad](image)

By default copy and delete buttons are always active. Update on the other hand is only active if the shape is marked as outdated, (more about this in chapter 4.3.6). Create containment and by-name connection are only available if such actions are allowed according to the model.

4.3.3 Property sheet

To enable a property view for the diagram when a pictogram is selected, it has to be converted to Eclipse’s standard property source. There are many ways to do this but we chose the simplest solution by making a contribution to Eclipse PDE runtime adapters plugin and providing a factory method for handling conversion of each pictogram element’s linked business object to the appropriate property source required by Eclipse.

When accessing the corresponding domain data for displaying in the property view, different Property Descriptors were used for different data types. Currently, String, Boolean and Numbers (Integer, float, double) are supported.

For String type, auto completion is available with content proposal data which is extracted from existing text found in the current XML resource. This allows completion based on already written values.

When a new value is inserted, SetValueFeature is called to modify the XML resource with the new value. Apart from making modifications to
the existing data in the XML resource, SetValueFeature also handles mutual propagation of changes on both sides of the by-name reference connections. When a by-name referenced object’s name is modified all other objects referring to it also get modified. When an existing referer object changes its value, the old by-name reference connection will be replaced with a new one if it refers to an existing object in the resource.

### 4.3.4 Remove and Delete

In Graphiti, *remove* means removing only the pictorial object from the diagram resource. To make it less confusing, the remove feature is hidden from the user since the creation of any visual representation without a linked business data object is not allowed. Instead, remove is made available only for use internally by other features for the removal of pictogram elements.

Delete on the other hand, is available both through the button-pad and the right click context menu. Invoking delete would result in deletion of the selected pictogram data. If the selected shape is a leaf node in the containment graph of the diagram, the corresponding business object will be deleted. This is basically the same as deleting the whole XML block and all of it contents surrounded by the element’s open and closing tag in the XML file. If the node have other child/children only the parent node will be deleted and their children will be remain intact. This was an implementation decision which allows better reuse of existing components. To avoid removal of children objects, they are moved temporarily to the diagram resource and will be moved to the XML resource as soon as they are linked to a new parent and all existing links are also swapped. The user can later use the create containment connection button to connect these shapes to other parent shapes for reuse.
4.3. EDITING

4.3.5 Copy and Paste

Similar to what was done during the import of XML resources, copy and paste were implemented in a similar manner. First, a copy of all linked business objects corresponding to the selected shapes is made and saved to the clipboard. When paste is called, all shapes in the clipboard are temporarily moved to the diagram resource before the Add feature gets invoked to create visual representation for these copies. It is important to have business objects persisted in a resource before calling add features or else linking will fail.

4.3.6 Update feature and Consistency Check

There are many scenarios where the XML resource and the complementary diagram resource might become out of sync with each other. No matter how this happens, any severe changes in the XML will make the diagram unusable. Since this can be fixed by simply re-importing the XML resource, we omit more advanced consistency checks for future implementation and only take care of simpler checks for attribute value mismatch.
as a demonstration.

Similar to how it was done in the layout feature, a layout tree is constructed prior to analysis. Since each row corresponds to one attribute or domain object, the check is a matter of comparing the existing string value in the pictogram to the corresponding value from the domain object. If any mismatch is found, the shape will be highlighted and update button pad will become available for manual update.

**4.4 Results**

The current version of the graphical editor met all the goals that were set at the beginning of this thesis. The current version of the implemented editor provides an excellent foundation upon which further development can be based upon.

The current implementation of the graphical editor offers all fundamental functionalities required for basic editing. It allows the user to create new diagnostic methods from a XSD schema model, import existing diagnostic methods and visualise them in a graphical diagram. The visualised data in the diagram can be edited, saved and exported as both a XML file and a printable image if required.

![Graphical Editor](image)
Chapter 5

Conclusion

5.1 Implementation

During the course of this thesis a large part of our time was spent investigating possible frameworks for the construction of the test implementation. A smaller part of it was spent building the test implementation of the editor. The investigation of the frameworks was done through trial and error with backwards compatibility as the only definite criteria. Thus, we discarded all solutions that require modification to the input data and found EMF to be the only candidate that could provide comprehensive support for Scania’s existing model resources.

5.1.1 Interviews and Paper Prototyping

The user interviews and paper prototyping proved vital in cementing the design of the user interface of the editor. More iterations of paper prototyping with improved design changes would have been optimal but due to scheduling and time constraints with the method developers, it was not possible to conduct paper prototyping more than once.

5.1.2 Test implementation

While EMF provides a good effort in XML Schema to Ecore transformation, this mapping is not lossless. We noticed during import of the existing XML Schema model information such as complextype restrictions were lost however the structure of the model element remained intact. Neither
serialization nor the end result was affected but we were not able to obtain the restricted values through the imported model. As a result, the content proposal based on these values had to be attained by other means. If more time was available, further investigation could have been about this mapping to find out the real reason as well as a possible solution for this.

In our implementation, an in-memory Ecore model was used for the construction of the editor. While this approach provides a flexible and uniform way to work with the existing model, it is a notch slower and less customizable when compared to code-generation [4] due to the extra overhead access. If performance and customization is the priority, this might be worth looking into.

While the editor works well in importing most XML diagnostic files, it could not import large files with content around 6000 rows and above. The culprit to this problem lies in the way AddOuterShapeFeature in Graphiti was implemented. In the latest commits, we tried to add all nodes in the XML resource in one transaction to make it easier to implement copy and paste feature. To fix this problem, we suggest two possible solutions:

- Revert back to the old implementation of AddOuterShapeFeature and modify all related features.
- Create a newer and more efficient add procedure for import used in ResourceUtils and keep everything else as it is.

Due to time constraints, we were not able to test other features EMF provides as much as we wished, two of which are EMF’s validation framework for dynamic validation of the output XML files and EMF Query for providing quick and efficient access to XML objects which is similar to SQL queries. If Scania choose to adopt EMF for wider use, the above mentioned features will aid in the development of the editor.

5.2 Results

The final results met all the goals of this thesis and we are pleased with what we have accomplished. Even though most features we gathered during interview and paper prototyping had been realized, some of them are still missing and should be addressed in future implementation. In the current version of the editor, there have been some minor changes to the proposed design. The intention was to further improve the system’s usability but as of now we have not found the time needed to verify their intended behavior.
5.2.1 Design Changes

The final version of the graphical editor has been largely implemented with the design decisions made during the interviews and paper prototyping phase with one exception.

![Figure 5.1: Paper Prototype Delete](image)

From the paper prototyping phase, it was decided that deleting attributes belonging to a selected object will be performed through the properties view as shown in figure 5.1. However, during implementation it was discovered that with nested data delete would be confusing as it could refer to objects that are not currently selected. Thus deleting of attributes was done through the right click context menu.

![Figure 5.2: Implemented Delete](image)

5.3 Future Work

The development of the graphical editor is far from complete and in this section we will suggest areas for future development and improvement.
5.3.1 Diagram Types and Views

Currently, there is one standard diagram type upon which the diagnostic methods are drawn and all types of data are shown on this. Since different method developers prioritize different datatypes in the diagnostic methods, it is good to be able to customize what is shown in the diagram. To start with, there is a custom diagram types or views that can be developed.

- Decision flow view - Highlights all the steps and outcomes for each condition in the diagnostic method and hides containment and documentation references and nodes.
- Deployment view - Visualize each node with images and descriptions the same way SDP3 shows the workshop technicians.

5.3.2 Clean up of Eclipse plugin & Integration with PT

The graphical editor uses the standard version of the Eclipse plugin application and thus comes bundled with many default features and packages that are unused and unnecessary. These features and packages should be stripped from the plugin application until only the essential features and packages needed by the graphical editor are left.

Once the development phase of the editor is complete, integration with PT and the existing validation should be the next step. Since PT is an Eclipse RCP application, integrating the graphical editor with PT should not present any obstacles.

5.3.3 Validation of diagnostic methods

Validation of the diagnostic methods is an important feature and should be implemented as soon as possible because it is not possible to deploy any of the completed diagnostic methods without checking that they are valid. Since validation software already exists in PT it should not be a problem once the integration between the graphical editor and PT is accomplished.
5.3.4 Testing

Basic testing has been conducted on the system and the largest and most visible bugs have been removed from the graphical editor. Wider and more thorough testing of the system is required. Thus conducting unit and component tests is recommended.

Although the user interface and interaction of the system has been designed by using user interviews and paper prototype tests, the completed editor has not been user tested. Thus, user interface tests with users is recommended.
Bibliography


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Sudharshan Ravi and Quang Vu