UNIVERSITY OF OSLO
Department of Informatics

Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool

Master thesis

Ivar Refsdal

1st November 2011

Comparison of GMF and Graphiti based on experiences from the development of the PREDIQT tool

Ivar Refsdal

1st November 2011

Abstract

Creating graphical editors can be a difficult undertaking. A graphical editor specialized at a well defined domain will likely be of good value for the end user, as such a tool should support the domain in a more direct way than a general purpose editor would, enabling, among other things, the user to avoid simple mistakes.

This thesis evaluates two frameworks aimed at the creation of graphical editors, namely GMF and Graphiti. GMF is considered a mature technology, whereas Graphiti is currently in its incubation phase. Both technologies use GEF, Draw2d and Eclipse in general as underlying technologies.

While GMF is fairly well documented, little is written about GMF as compared to Graphiti. Furthermore neither has been evaluated with respect to tree-based methods using value propagation.

The main hypothesis is that GMF will outperform Graphiti with respect to the following criteria:

- Applicability for supporting tree-based methods using value propagation.
- Development time.
- Maintainability.
- Customizability.
- Various criteria set forth in Myers et al.

This is discussed based on the experiences made during the development of two editors, reviewing related work, as well as a survey where users try both editors.

While the two first criteria were found to be true, the three latter were found to be in favor of Graphiti. Both tools were equally favored by users. Related work was generally found to confirm the findings of this thesis.

More research should be done to further strengthen or weaken this thesis' findings. As Spray, the model driven approach targeting Graphiti, becomes more mature, it seems well fit and a natural comparison for GMF.

Acknowledgments

I would like to thank my supervisor Ketil Stølen and my co-supervisors Aida Omerovic and Fredrik Seehusen for their helpful comments and guidance throughout the writing of this thesis. This work would not have been the same without them.

I would also like to thank my family, girlfriend and friends for keeping me motivated.

Any remaining errors are of course my own.

– Ivar Refsdal, 1st November 2011.

Contents

1 Introduction **1**

2 Background **3**
- 2.1 Requirements gathering . . . 3
- 2.2 Software quality . . . 4
- 2.3 Selecting a technology . . . 5
  - 2.3.1 Evaluating Open Source Software . . . 5
- 2.4 Code maintainability . . . 6
  - 2.4.1 Modifiability . . . 7
    - 2.4.1.1 Design patterns . . . 7
  - 2.4.2 Understandability . . . 8
  - 2.4.3 Measures of complexity . . . 8
- 2.5 Conclusion . . . 9

3 Problem statement and research method **11**
- 3.1 Why Eclipse, GMF and Graphiti . . . 11
- 3.2 The problem domain . . . 12
- 3.3 Main hypotheses and goal . . . 12
- 3.4 Research method . . . 13

4 State-of-the-Art **15**
- 4.1 Underlying technology . . . 15
  - 4.1.1 Eclipse Modeling Framework . . . 16
    - 4.1.1.1 Java implementation of the model . . . 17
    - 4.1.1.2 Adapters . . . 17
  - 4.1.2 Draw2d . . . 17
  - 4.1.3 Zest . . . 19
  - 4.1.4 Graphical Editing Framework (GEF) . . . 19
- 4.2 Graphical Modeling Framework (GMF) . . . 20
- 4.3 EuGENia . . . 21
- 4.4 Graphiti . . . 22

5 Requirements for the tool **23**
- 5.1 Overview of PREDIQT . . . 23
- 5.2 Requirements for the PREDIQT tool . . . 24
  - 5.2.1 Overall requirements . . . 25
  - 5.2.2 Functional requirements . . . 25
  - 5.2.3 Data requirements . . . 26

5.2.4 Technical environment requirements 26
5.2.5 Non-functional requirements 26

6 Implementation of the tools 27
6.1 Shared model and code 27

7 Evaluation 31
7.1 Evaluation of applicability for supporting tree-based methods 31
7.1.1 Conclusion 38
7.2 Survey of tools 38
7.2.1 Background of students 38
7.2.2 Creation of DV 38
7.2.3 Time used 38
7.2.4 Favorite tool 40
7.2.5 Feedback during survey 42
7.2.6 Conclusion 43
7.3 Evaluation of development time 43
7.3.1 Graphiti 43
7.3.2 GMF 44
7.3.3 Actual development time spent on GMF and Graphiti . 44
7.3.4 Conclusion 45
7.4 Evaluation of maintainability 45
7.4.1 Code size 46
7.4.2 Dependencies 46
7.4.2.1 Discussion 48
7.4.3 Making a change to the editors 48
7.4.3.1 Disable linking of a node to itself 48
7.4.3.2 Enabling direct editing 49
7.4.4 Structuredness 50
7.4.5 Conclusion 52
7.5 Evaluation of customizability 52
7.5.1 GMF 52
7.5.2 Graphiti 53
7.5.3 Practical experiences 54
7.5.4 Conclusion 55
7.6 Evaluation of criteria set forth in Myers et al 55
7.6.1 Specificity 55
7.6.2 Threshold and ceiling 55
7.6.3 Predictability 55
7.6.4 Path of least resistance 56
7.6.5 Conclusion 56
7.7 Related work 56
7.7.1 Lack of complex editing operations 57
7.7.2 Migrating from XML/UML to Xtext/GMF 57
7.7.3 Difficult to use GMF 58
7.7.4 What does GMF say about itself? 59
7.7.5 Conclusion 60

8 Conclusion **61**
8.1 Further work and suggested improvements 62

Bibliography **63**

A Additional lessons learned **69**
A.1 Faster programming feedback cycle: Learn OSGi 69
A.1.1 Problem 69
A.1.2 Solution 69
A.2 Simple traceability: Git 71
A.2.1 Problem 71
A.2.2 Solution 71
A.3 Suggested prerequisite learning for GMF and Graphiti internals 71
A.3.1 Problem 71
A.3.2 Solution 71

B Survey **73**
B.1 Setup 73
B.2 Survey information as given to users 73

C Installation and source code of editors **77**
C.1 Shared environment 77
C.2 Installation of editors 77
C.3 Source code of editors 78

D PREDIQT case study **79**
D.1 Target modeling 80
D.1.1 Characterize the target and the objectives 80
D.1.2 Create quality models 81
D.1.3 Map design models 82
D.1.4 Create dependency views 83
D.2 Verification of prediction models 83
D.3 Application of prediction models 83
D.3.1 Specify a change 83
D.3.2 Apply the change on prediction models 84
D.3.3 Within the scope of models? 84
D.3.4 Quality prediction 84
D.4 A new change? If Yes, go to step 3 84
D.5 Other experiences 85
D.5.1 Conclusion 85
D.6 Quality models 86
D.6.1 Design models - Original system 86
D.6.2 Design models - New system 89
D.6.3 Dependency views 95

List of Figures

4.1 High level structure of Eclipse. 15
4.2 Structure of the Eclipse platform. 16
4.3 Structure of GEF. 18
4.4 Overview of the GMF development process. 21

6.1 Screenshot of the GMF-based tool. 29
6.2 Screenshot of the Graphiti-based tool. 30

7.1 How easy it was to create the DV the first time. 39
7.2 Time used to solve the exercises using GMF, then Graphiti. . . 40
7.3 Time used to solve the exercises using Graphiti, then GMF. . . 41
7.4 Favorite tool. 41
7.5 GEF architecture. 50
7.6 Graphiti architecture. 51

A.1 OSGi console 70

B.1 Dependency tree that should be created. 74

List of Tables

2.1 ISO/IEC 9126 software quality characteristics. 4

7.1 Overview of the evaluation of the two editors with respect to the requirements. 37
7.2 Actual development time. 45
7.3 Code size by package. 46
7.4 Code size by tool. 47
7.5 Imports used by the two tools. 47
7.6 Nested cyclomatic complexity. 52

Chapter 1

Introduction

Creating graphical editors can be a daunting task. As argued in [31], the choice of user interface framework will strongly influence the programmer's productivity. Once development has begun, it is generally hard to switch to a different underlying technology. The importance of the choice taken is further fortified by the fact the that 80-90% of a system's lifetime costs are spent on maintenance ([36, 6]). Thus an early bad decision with respect to choosing a graphical framework will likely haunt the project and be difficult to alleviate.

Currently the Eclipse Modeling Framework (EMF) [43] is actively used and well proved for specifying the domain model for widely differing areas. However, less work has been done on the use of EMF as the domain model for concrete visual syntax editors. Currently the main projects targeted towards this is the Graphical Modeling Framework (GMF) [18] and Graphiti.

Researchers working on a specialized topic will typically spend a large amount of time working with domain-specific data. However, the tool used may not be particularly well fit for their domain. This is the case for the PREDIQT method [33] and surely also many other methods. A domain-specific tool would be more appropriate and time saving for the researcher. This thesis will limit itself to deal with tree-based value propagation methods.

How applicable are GMF and Graphiti for creating editors that support these types of methods? What are the pros and cons of each technology? This is the topic of this thesis, and the findings of this study are largely based on the experiences gathered from implementing a prototype of a PREDIQT tool in both GMF and Graphiti.

In chapter 2 an introduction on software with respect to evaluation, quality and requirements gathering will be given. This is necessary knowledge for later chapters. Chapter 3 will explain the problem domain further and present the main hypotheses, i.e. what this thesis intends to examine, as well as how this examination will be done.

Chapter 4 will present GMF, Graphiti and their underlying technologies further. In chapter 5 a brief introduction to PREDIQT and the requirements for the tool will be given. This will serve as a backdrop for evaluating the

tools. Chapter 6 will briefly explain the implementation of the tools.

In chapter 7 the comparison of GMF and Graphiti will be done according to the plan presented in chapter 3. This is the major bulk of the thesis, and will thus build on most of the preceding sections.

Finally, in chapter 8, the conclusion of the thesis will be summarized and future work suggested.

Chapter 2

Background

In order to create and evaluate a tool, as is much of the basis of this thesis, one needs to have a reasonable understanding of several things:

- How to gather requirements for a tool.
- The method that is to be supported, namely PREDIQT. It will be briefly explained in chapter 5.
- What software quality is.
- Considerations to be made when selecting a technology.
- Strategies for ensuring maintainability, e.g. design patterns.

The three latter points are somewhat interrelated, but they are treated separately here. In the evaluation in chapter 7 , however, a mixed subset will be used.

2.1 Requirements gathering

While most people have an intuitive understanding about what a requirement is, it is appropriate with a definition:

"A requirement is a statement about an intended product that specifies what it should do or how it should perform." [41]. Furthermore requirements are typically divided into functional and non-functional requirements, whereas the former says something about "what the system should do" and the latter says "what constraints there are on the system and its development".

Requirements gathering is, in an ideal world, a sequential process consisting of three steps:

1. Gathering data.
2. Analysis and interpretation of the data.
3. Extraction of requirements.

Characteristic	Sub-characteristics
Functionality	Suitability, accuracy, interoperability, security, functionality compliance.
Reliability	Maturity, fault tolerance, recoverability, reliability compliance.
Usability	Understandability, learnability, operability, attractiveness, usability compliance.
Efficiency	Time behavior, resource utilization, efficiency compliance.
Maintainability	Analyzability, changeability, stability, testability, maintainability compliance.
Portability	Adaptability, installability, replaceability, coexistence, portability compliance.

Table 2.1: ISO/IEC 9126 software quality characteristics. From [22].

In practice though, it is not always so straight forward, i.e. the process is typically more interleaved. Analysis of data may for example lead to or require gathering of more data. Furthermore these steps seeks to achieve two central goals:

- Identifying the users' needs.
- Developing a stable set of requirements.

The former aim is sought through learning about the users, their work and the context of that work. The latter aim is sought by extracting requirements from the needs by analyzing and interpreting them.

As fixing problems after a product delivery is much more costly than fixing them early in the requirements or design process, see e.g. [29], requirements gathering is a crucial step to do as good as possible.

2.2 Software quality

Quality, according to ISO 8402, is "the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs." [22] decomposes quality into six quality characteristics which is further decomposed into sub-characteristics. These are given in table 2.1. The six main characteristics are described briefly below[1]:

- *Functionality* is the totality of essential functions of any product or service. These functions should be either present or not, i.e. they are not present to some degree.
- *Reliability* is the capability of the system to maintain its service under defined conditions for defined periods of time. For example one aspect may be how well the system is able to withstand a component failure. For this thesis it is more relevant to consider the ability to avoid or withstand invalid data.

[1] Paraphrased from http://www.sqa.net/iso9126.html, retrieved 8th February 2011.

- *Usability* only exists to a certain degree and refers to how easy it is to use a given function. Learnability, how easy it is to learn a system, is a major sub characteristic of usability.
- *Efficiency* is concerned with the system resources used when providing the required functionality, e.g. memory and disk space.
- *Maintainability* is the ability to identify and fix a fault within a software component. It is affected by code complexity and modularization. Maintainability will be treated further later in this chapter.
- *Portability* refers to how well the software can adopt to a change in its environment or with its requirements.

2.3 Selecting a technology

Deciding on a technology is important. One may distinguish between Open Source Software (OSS) and commercial software, typically represented by the Commercial Off-The-Shelf category of software. As this project has no spending budget and no expected revenue, it will only concern itself with the former.

OSS may have benefits such as lower purchasing costs, availability of high quality products, adherence to open standards and no vendor dependency [44]. While there is no universally accepted definition of OSS,[16] outlines three of the main criteria for determining whether a particular software is open source, as given by the Open Source Initiative[2]:

- It should be possible to distribute the software freely.
- The source code should be available.
- One should have the right to create derived works through modification.

These criteria will be required to be fulfilled before existing prospective software should be considered.

2.3.1 Evaluating Open Source Software

How can an OSS' quality be evaluated? Extensive literature exists on methods and challenges in assessing OSS, see e.g. [26, 38, 45] among others. Some of these methods are partially based on ISO standards such as [22] which was presented in section 2.2. As time is an important factor in this project, a method that is more automatic is preferable over one that requires more manual work.

However, these methods are general purpose, and as argued earlier, usability is particularly important for the new tool and should thus be given

[2] http://www.opensource.org/

special emphasis. To a considerable degree usability is concerned with user interfaces, and thus a more specific evaluation strategy with respect to user interfaces is favorable. This is in accordance with the findings of [19]. [31] lists five recurrent themes which are important for determining if a user interface framework is successful, four of which are interesting for this document and described below. Four more general themes are added from other sources.

1. The tool or framework should address a specific part of user interfaces.
2. Threshold and ceiling. By threshold it is meant how easy a tool is to learn. Ceiling means how much can be done within that tool, without resorting to modify underlying code, calling lower level APIs, etc. A given tool will typically score high or low in both threshold and ceiling, while what is optimal is low threshold and high ceiling.
3. Predictability: Tools that use automatic techniques, for example code generation, sometimes behave unpredictably. This has not been well received by programmers.
4. Path of least resistance: Tools shape what kinds of user interfaces can be created, and a successful tool should help the developer towards doing the right things, and away from doing the wrong things.
5. Structure: According to [10] deep hierarchies are more error prone than shallow ones. However, by the same argument, high-level languages should be more error prone to use than a low-level language, which generally isn't the case, so this argument may not always apply if the hierarchies are mature, well tested or completely hides the underlying layer.
6. Complexity: A number of metrics exists for measuring software complexity. Some of these are treated later in this chapter. There seems to be disagreement whether to take lines of code as a measure of complexity, see e.g. [50, 14] for differing views.
7. Lessons learned: Researchers often write down the lessons they learned after completing a project, and in the context of framework usage this surely should give an impression of the framework's applicability.
8. Maturity: If one seeks create a well working tool, it is generally an advantage if the framework or existing application one builds upon is mature. Choosing technology that is not mature is a common cause of software projects failures.

2.4 Code maintainability

While usability for the end user is of course important, it is also obviously important that a program is maintainable for current and future develop-

ers. As argued earlier and seems widely accepted in the literature, software maintenance stands for the major part of software project costs.

What exactly is meant by maintainability? [27] defines it as "the ease with which a software system can be corrected when errors or deficiencies occur, and can be expanded or contracted to satisfy new requirements." Three sub criteria should be met if a system should be called maintainable: The system should be testable, understandable and modifiable. This thesis will particularly focus on the two latter.

[27] gives the following definition of understandability: "Understandability is defined as the ease with which we can understand the function of a program and how it achieves this function by reading the program source code and its associated documentation." If a framework has a very high learning curve and thus an application using this framework may be hard to maintain, this should surely affect the decision of whether to use this particular framework.

Modifiability is defined as "the ease with which a program can be changed" [27]. One central sub criteria for modifiability is modularity and structuredness.

An introduction to these criteria follows.

2.4.1 Modifiability

In order to make programs modular and well structured, one should apply object-oriented techniques as well as design patterns. In order to achieve good object-oriented design, [17] suggests the following:

- Design to interfaces.
- Favor composition over inheritance.
- Find what varies and encapsulate it.

These suggestions are also applicable to, as well as found in, design patterns. The reader is assumed to be reasonably familiar with the object-oriented techniques and the subject will not be treated further. However, it will be noted when these principles are applied later. A few of the most common design patterns are presented below. The cited works should be consulted for additional patterns.

2.4.1.1 Design patterns

The idea of design patterns originates from [2], where a design pattern, as applied to architecture, is described as "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." An alternative, more succinct definition, from the same book, is that "patterns are solutions to problem in a context."

Design patterns was later applied to computer science in [17], which remains the classical reference. The idea is the same as in architecture: To provide a reasonably standard solution to a recurring problem of a certain type. A number of other books also describes design patterns, see e.g. [40].

In the text that follows a brief description of important patterns and strategies to make code maintainable is given. The reader is referred to the above works for a more extensive explanation.

- **Facade pattern.** "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." [17]
- **Adapter pattern.** "Convert the interface of a class into another interface that the client expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces." [17]
- **Observer pattern.** "Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." [17]
- **Factory pattern.** "Provide an interface for creating families of related or dependent objects without specifying their concrete classes." [17]

2.4.2 Understandability

[27] elaborates further on the criteria for what makes a program understandable:

- A program should be *concise*, meaning that every program instruction should be reachable.
- A program should be *consistent*, meaning that it is written in a consistent style and a consistent design approach.

Program complexity is also obviously related to understandability: A complex program will arguably be hard to understand. That said, there are many metrics for measuring program complexity, many of which are not merely subjective measures. An introduction to this follows.

2.4.3 Measures of complexity

[20] remains a classical reference to object-oriented complexity metrics. Many open source programs targeting measurement of complexity cites this book and the metrics presented. Particularly interesting for this thesis is overall complexity.

Lines of code is probably the most simple metric, yet says something about how large and thus likely complex a system will be.

Efferent coupling describes the number of types of external packages that are used inside the package being measured. Thus one can get an idea of how much one needs to know about external projects.

Cyclomatic complexity is a measure of how many distinct paths there are within a method. Thus it is computed by looking at conditional expressions. This can also be summarized for all methods.

Depth of inheritance hierarchy is a measure of how deep the inheritance hierarchy is. The depth of e.g. *java.lang.Object* is one. Any subclass will have a depth of its superclass plus one.

2.5 Conclusion

This chapter has briefly visited topics such as requirements gathering, software quality with respect to user interfaces and maintainability. While some of these topics are somewhat interrelated, an introduction have nevertheless been given.

Chapter 3

Problem statement and research method

This section will further explore the needs of which was briefly described in chapter 1 and which this thesis intends to alleviate.

There exists a fair number of tree-based prediction methods, of which PREDIQT is one. These usually share common properties such as value propagation through the tree and each node typically have a fixed number of attributes. In addition a leaf node may serve a slightly different purpose than an interior node. An analyst may also choose to manually override one or more of the values being propagated. All of these characteristics applies to PREDIQT, and likely also to many other prediction methods. Furthermore tree-based methods using value propagation will necessarily be a super-set of tree-based prediction methods. Thus eventual findings will likely apply to these methods also.

A fair amount of time will typically be spent using these methods and therefore good tooling support is important. However, there is no special tool support for methods such as PREDIQT. Furthermore, tool creation and creation of graphical user interface applications in particular, as shown in section 2.3.1, can be a time consuming task.

Which technology should be used for creating this particular type of tool? In the ideal world there would be enough time to evaluate all prospective technologies. Unfortunately this is not possible. While it is not certain how large the sum of all major graphical user interface frameworks would be, it is fair to say it would be close to impossible to do a thorough evaluation of each one. Instead this thesis will limit itself to focusing on GMF and Graphiti, which is part of the Eclipse world. The reasoning behind this follows.

3.1 Why Eclipse, GMF and Graphiti

Surely Eclipse, GMF and Graphiti all have their disadvantages; this will be the focus of later sections. In this section the focus is on why they were chosen as opposed to other technologies.

Why was the Eclipse platform chosen? The Eclipse platform enjoys reasonably widespread popularity in academia, e.g. a search for "Eclipse" and "Netbeans" yields 4000 and 350 results each in the ACM digital library. IntelliJ, the third largest Java IDE, gives even fewer results. This statement should also hold true with respect to modeling and EMF. Furthermore the Eclipse foundation has its own lawyers taking care of legal issues, and so all source code should have a clean intellectual history. In addition to this, the Eclipse Public License, under which Eclipse software is typically licensed, is liberal, allowing creation of for-money closed-source derivative works and so forth. These are the main reasons the Eclipse platform was chosen.

Within the Eclipse world EMF has reached widespread popularity. For this thesis' purposes EMF seems like the obvious choice for modeling the PREDIQT domain. It is mature, well documented and widely used. There does not appear to the author any other more well fit technologies. However, with respect to the creation of graphical editors, only two mainstream frameworks use EMF as the model, namely GMF and Graphiti. Thus, the particular focus on GMF and Graphiti arises naturally as there are not many alternatives when one restricts oneself to Eclipse and EMF. As will be shown later, both GMF and Graphiti uses the same underlying technology, making them well suited for comparison.

3.2 The problem domain

Now that the scope has been narrowed to GMF and Graphiti, what exactly constitutes the problem domain? Both are intended to be used for the creation of graphical editors. Though GMF has been actively developed since 2005, there are not too many evaluations available of GMF and its respective components. Not a single evaluation exists with respect to using GMF as a tooling technology for supporting tree-based methods in general or tree-based prediction methods in particular. Furthermore there is only a single comparison of GMF and Graphiti[1]. Given that they are the two main competing technologies within their particular domain, this is not a very good situation.

In order to make the choice easier for future programmers on deciding which frameworks to use when developing an editor for a tree-based method, more evaluations on this particular subject should be held. This thesis' goal is to reduce this lack of information. In the following sections how this will be attempted to be examined, is presented.

3.3 Main hypotheses and goal

As argued in the preceding text, there is a need for an evaluation of GMF with respect to supporting tree-based prediction methods, as well as comparing it to what is achievable in Graphiti. In order to confirm

[1]As of 30th October 2011.

or disprove GMF as the right technology, as opposed to Graphiti, for supporting these methods, the following hypotheses are presented:

Hypothesis 1. *GMF is a more appropriate technology for creating tools that support tree-based methods using value propagation than Graphiti is.*

Hypothesis 2. *Using GMF will shorten initial development time compared to using Graphiti.*

Hypothesis 3. *The resulting tool created by GMF is easier to maintain than a tool made with Graphiti.*

Hypothesis 4. *The resulting tool created by GMF is easier to customize than a tool made with Graphiti.*

Hypothesis 5. *GMF will perform better with respect to the criteria outlined in [31] as compared to Graphiti.*

3.4 Research method

Now that the main hypotheses have been presented, how does one go about for proving or disproving them? According to [28], one should seek research evidence that maximizes three properties:

- Generality: That the results are valid across populations.
- Precision: That the measurements are precise.
- Realism: That the evaluation is held in environments similar to that of reality.

As stated earlier, this thesis will use the implementation of two tools, one in GMF and one in Graphiti, as the primary underlying evidence for testing the hypotheses. The dual implementation is classified as a field experiment according to the research methods presented in [28] and the factor manipulated is the underlying framework. While field experiments scores high on realism, they score low on generalization and moderate with respect to precision. However, as the PREDIQT method necessarily shares many common traits with other methods, the results should yield some general knowledge as well.

After the initial implementations, the two resulting tools can be used to argue about the hypotheses. This will be done in several ways:

- Evaluate to what degree the frameworks differ with respect to the given requirements of the tool. This is well suited for answering hypothesis 1.
- Hypothesis 4 and 3 will be argued for based on the experiences made during the implementation. This will be a logical argument. Additional supporting evidence can come from reviewing the source code produced with respect to the information given in section 2, i.e. quality, complexity and so forth.

- The general experiences from the implementations will be used to answer hypothesis 2 and 5.
- Reviewing related work will be done as this can give an indication of whether valuable lessons has been made before that are applicable for this thesis. This will concern mostly GMF-related papers as there are yet to be a paper describing Graphiti[2].

[2] As of 30th October 2011.

Chapter 4

State-of-the-Art

This section will describe the state of the art, namely GMF and Graphiti. As argued in [39], graphical editors require customization to a considerable degree. The implication of this may be that one also needs to have an understanding of the underlying layers of GMF and Graphiti. These underlying layers include:

- EMF: The Eclipse Modeling Framework.
- Draw2d: A basic drawing system built on top of SWT.
- Zest: A simple framework for viewing graphs built on top of Draw2d.
- Graphical Editing Framework (GEF): A graphical editing framework primarily based on Draw2d.

An introduction to these components follows. Afterward GMF and Graphiti will be explained.

4.1 Underlying technology

The Eclipse platform provides an infrastructure for defining and using so-called extension points. Essentially this is to say that new components may extend existing components. This is part of the Open Services Gateway initiative (OSGi), of which Equinox, an Eclipse project, is the reference implementation. The goal of OSGi is to bring modularity to the Java platform.

An extension point need not know about the existence of those extending it. As an example, the tools developed in this thesis will plug into various Eclipse platform extension points. It would be very cumbersome if the platform itself would need modification for this.

Figure 4.1: High level structure of Eclipse.

UI	Workbench
	JFace
	SWT
Core	Workspace
	Runtime

Figure 4.2: Structure of the Eclipse platform.

The *workspace* manages one or more top-level projects. These projects maps to files and folders in the underlying file system.

The *runtime* manages handling of plug-ins such as discovery, loading and unloading. This is done by Equinox, an OSGi framework implementation. Given that the plug-in is configured correctly, Equinox should resolve dependencies correctly. Considerable burden is thus taken away from the developer with respect to dependency issues. See e.g. [49] for details.

SWT is an abbreviation for the Standard Widget Toolkit, which is designed to provide portable access to underlying operating system's user interface facilities.

JFace provides a set of convenience helper classes built on top of SWT which helps to solve common UI problems. It also implements the model-view-controller paradigm.

The *workbench* is sometimes called the Eclipse Platform UI. For the end user, it consists of views and editors. For the developer one has to create editors and views with respect to how the workbench operates.

For more details on Eclipse, see [8].

4.1.1 Eclipse Modeling Framework

EMF [43] brings modeling to and for Eclipse. The models are defined using the Ecore model. A number of features are provided by EMF working on or as a part of Ecore objects:

- A reflective API giving generic access to attributes of the model.
- Persistence.
- Notification and adapters.
- Comparing.
- Searching.
- Copying.
- Editor generation.

The Ecore model can be used to generate a generator model, which in turn will generate various parts of code such as the Java implementation of the model, EMF.Edit, EMF.Editor as well as a basic test setup.

4.1.1.1 Java implementation of the model

EMF will generate interfaces mirroring the model, as well as implementation classes implementing those interfaces. This is in accordance with the advice from [17]. Furthermore, this pattern allows Java to support multiple inheritance. All these generated interfaces will extend *EObject*, which in turn will extend *Notifier*. The latter interface provides the adapter design pattern. In EMF terminology the adapter pattern is roughly equivalent to a combination of both adapter and observer pattern in [17]. The *EObject* interface provides operations such as retrieving the contents of the object and various reflective methods providing generic access.

4.1.1.2 Adapters

Adapters are used for listening for changes to an object and also for extending the behavior of an object.

One example of behavior extension is item providers. Item providers, as the name suggests, provide functions on behalf of items (objects). For our case the objects will be instances of the EMF model. Item providers provide:

- Content and label functions. This can be used e.g. in a tree view in Eclipse.
- Property sources. This can be used in the property view of Eclipse.
- Command factory.
- Forwards change notifications from EMF objects. This is convenient so that a developer can merely use an extended behavior of an object and yet pretend it is the real object itself. Thus the developer only needs to deal with a single object rather than two.

A large part of this functionality is contained in general base classes, so the generated subclasses only implement a small portion of this and the rest is handled generically. As stated above, this functionality can be used by for example a TreeViewer or PropertySheet class.

The above mentioned item providers are generated by the EMF.Edit component. It creates UI-independent classes providing the above mentioned functions. The EMF.Editor component on the other hand generates UI-dependent code which essentially wraps the code generated by EMF.Edit. This becomes a kind of double delegation pattern which is a little complex. However, if one needs to supports additional back ends, for example a web-application, it should be possible to write a different EMF.Editor-like back end and the reuse the possibly customized code in EMF.Edit.

4.1.2 Draw2d

Draw2d is generally the lowest layer of the frameworks one needs to concern oneself with with respect to GEF. See figure 4.3 for an overview

<table>
<tr><th>Component</th><th>Important concepts</th></tr>
<tr><td rowspan="6">GEF</td><td>Requests</td></tr>
<tr><td>Edit policies</td></tr>
<tr><td>Commands</td></tr>
<tr><td>Tools</td></tr>
<tr><td>Edit domain</td></tr>
<tr><td>Edit parts</td></tr>
<tr><td rowspan="2">Zest</td><td>Content providers</td></tr>
<tr><td>Layout algorithms</td></tr>
<tr><td rowspan="5">Draw2d</td><td>Event dispatcher</td></tr>
<tr><td>Lightweight system</td></tr>
<tr><td>Update manager</td></tr>
<tr><td>Layout managers</td></tr>
<tr><td>Figures</td></tr>
<tr><td rowspan="3">SWT</td><td>Painting</td></tr>
<tr><td>Canvas</td></tr>
<tr><td>Shell</td></tr>
<tr><td colspan="2">Operating system</td></tr>
</table>

Figure 4.3: Structure of GEF.

of the layers. Much of the text that follows on the various layers is based on [37].

As the name suggests, the primary task of Draw2d is to handle two dimensional drawing. Draw2d is built upon SWT and is considered lightweight in the sense that Draw2d objects are not tied to an operating system resource, in contrast to SWT objects, and thus some less burden is put on the programmer. In addition to drawing, Draw2d also provides listening to events such as focus, keyboard and mouse events.

The top level element of a Draw2d system will typically be a so-called *LightWeightSystem*. It's job is to handle event dispatching, updating of figures as well as acting as a container of all figures. This is done through delegation to specialized classes, whereas the latter is a special root figure class. All figures contained by the root figure must implement the *IFigure* interface, which contains methods for translating between different coordinate system (relative and absolute), adding of child figures, painting, event handling, setting a layout manager, size of the figure, colors and so on. Aside from the obvious types of figures, such as a rectangle or an ellipse, a figure may also be a connection, a layer or a pane. It is helpful to have several layers, i.e. typically there will be one layer for connections and one for nodes. This is useful for example when one wants to route connections to have the shortest path without overlapping nodes.

When using GEF, one needs to understand Draw2d primarily with respect to concrete figures such as nodes. Event handling, layers and so on are taken care of by the higher-level frameworks.

4.1.3 Zest

Zest is a framework built on top of Draw2d. Some parts of Zest is also built on *JFace*.

It's main contributions are providing more advanced layout algorithms, as well as delegating graph creation to so-called content providers. The provided layout algorithms include a directed graph, grid, tree, spring, radial as well a composite layout algorithm.

The content providers provides a skeleton for creating a nodes and connections for the graph based on what type of data is stored in the domain model, i.e. if the model contains data that represents relationships, nodes and possibly nested content. Thus it should be relatively easy to create graphical views for differing domain models. Zest also provides various style providers for customizing the look and content of nodes and connections.

All in all Zest makes for a much quicker way to create Draw2d graphs. However, it comes at a cost: It is more difficult to customize the connections and figures, and one also gives up considerable control to the framework. In addition there is not any editing capabilities.

4.1.4 Graphical Editing Framework (GEF)

Formally the GEF *feature* is actually consisting of three *components*: GEF, Zest and Draw2d. This thesis will refer to the GEF *component* as simply GEF.

GEF is built upon Draw2d, SWT and JFace among others. It provides more flexibility and customization for showing nodes and interacting with and editing models. This is done with the help of the model-view-controller (MVC) paradigm, which is a well-proved technique.

MVC has three components: *Model*, *View* and *Controller*. The model is responsible for the actual business domain, i.e. what is actually persisted across sections. The view is only concerned with display of certain figures and labels. The controller binds the model and the view all together by listening to changes in the model and updating, creating or removing views correspondingly. Arranging things this way, the model should be completely separate from both the view and the controller. This allows for multiple editors for the same model, as well as enabling a clear separation of concerns.

However, GEF goes further than just plain MVC. According to [48] the architecture of GEF is closer to the Presentation-Abstraction-Control, see [7] for a description of this pattern. GEF introduces several other concepts such as an edit part factory, an edit domain, edit policies, requests, tools and commands. Additionally different terminology is used. MVC's view is a *figure*, while *controller* is called an edit part.

The edit part factory simply creates editparts, GEF's equivalent of controllers for MVC, based on the context, i.e. the owner editpart, as well as the model. This is done using the factory pattern as described in 2.4.1.1.

Requests are high-level operations that contain all the information that is needed to make an application change.

Commands represents a change in the application state that can possibly be undone. It is also possible to chain together several commands. Commands are put on the command stack, effectively giving the application a history.

Multiple edit policies can be associated with each edit part. An edit policy can contribute editing behavior to an edit part in several ways. Edit policies creates commands in response to requests. They also handle feedback and forwarding or delegation if needed.

The edit domain is the state of the GEF application. It has a command stack for recording the history of what actions the user has taken, so that these can be undone and redone. Furthermore it has one or more edit part viewers, e.g. the main view, and an active tool that determines what happens when the user interacts with the diagram.

As an example, when a user hovers a figure with the mouse, the LightWeightSystem will delegate the hovering to the viewer, which in turn will delegate it to the edit domain, then to the active tool. The active tool will then create a request and send it to the edit part, which will consult its edit policies, of which at least one should return a command. Visual feedback may also be performed. Note that while the command has been created, it has not been executed. Execution, e.g. triggered by a mouse click, follows roughly the same path, i.e. from LightWeightSystem to the edit domain, to the active tool which has saved the command. The tool then gets the current command stack, and uses this for executing the command.

While GEF adds some complexity, it should allow for code reuse, particularly for large projects, as well as being extensible.

4.2 Graphical Modeling Framework (GMF)

GMF [18] is a collection of three components:

- The GMF Tooling provides a model-driven approach to generating graphical editors.
- The GMF Runtime provides common features such as printing, export as well as actions. It also provides a bridge between GEF and EMF commands.
- The GMF Notation provides a standard notation for storing the diagram information separately from the semantic information.

An overview is presented in figure 4.4. Version 2.3.0 of GMF was used in this thesis.

As a tool developer, the GMF Tooling is of most interest to us. It consists of several models, namely the graphical definition model, the tooling definition model, the mapping model as well as the generator model.

The graphical definition model defines what the visual elements of the editor should look like. This model has a strong resemblance to

the classes available in GEF and Draw2d. The tooling definition model defines what tools should be available in the palette. Neither of these are directly concerned with the actual business domain. The mapping model is what connects the business domain model with the two former models. This is to say that the mapping model defines the mapping between the actual business domain and the graphical definition model and the tooling definition. From this mapping model the generator model can be generated. This model will reference the other models, as well as giving the user options to define certain generation settings such as output folder, whether to generate a RCP application or not, and so on.

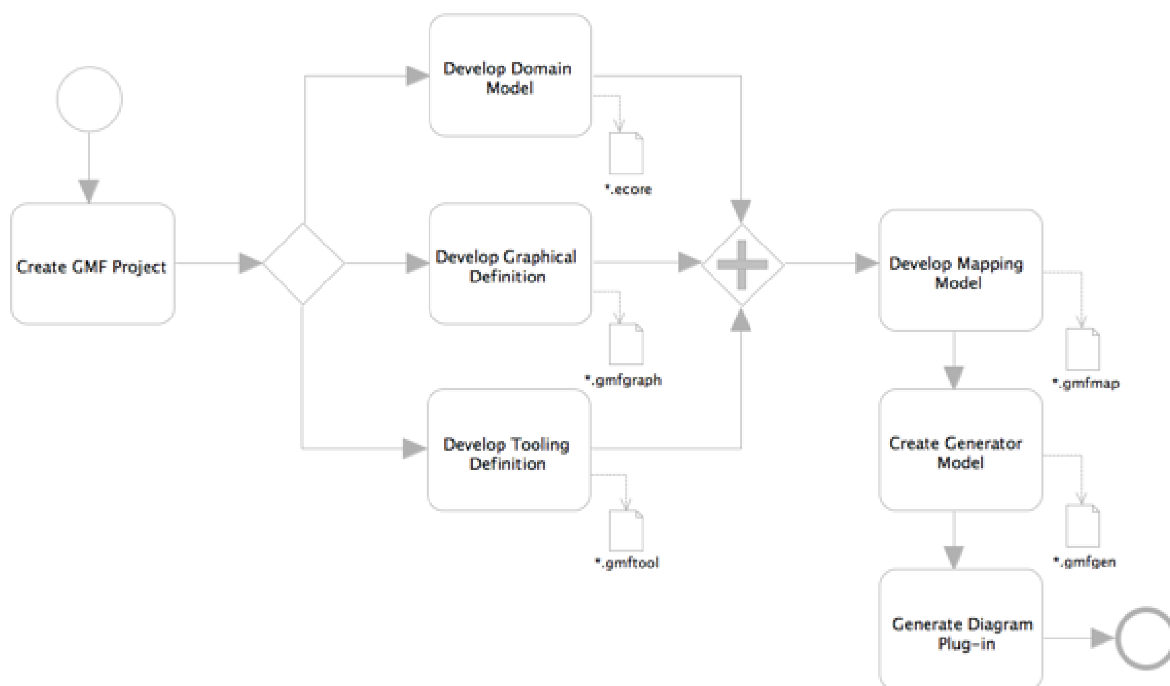


Figure 4.4: Overview of the GMF development process.

4.3 EuGENia

On top of GMF a tool called EuGENia[24] also exits. This produces a reasonable default of all the GMF Tooling models based on an annotated Ecore model. The generation of the defaults can be customized using the various Epsilon languages. This tool was used in this thesis to generate the GMF models. In short EuGENia can help the tool developer to jump start the development of a GMF-based editor.

4.4 Graphiti

Graphiti is a framework for creating graphical diagram editors. The framework itself does not do any code-generation and is written in plain Java. Currently this project is in incubation phase. During the writing of this thesis, a different project, Spray, using code generation and targeting the Graphiti framework, was released, but it remains in a fairly early phase.

Graphiti takes a different approach than both GMF and GEF. Instead of requiring the user to use the Model-View-Controller paradigm, it introduces so-called *features*. These encompass concepts that are all present in MVC:

- Creation, deletion and changing of business model elements.
- Creation, deletion and updating of visual elements.

As Graphiti internals builds on GEF, it also provides similar concepts to GEF, but again they are provided uniformly through *features*. In fact edit policies, requests and commands are invisible to the user of Graphiti. Upon entering features that are supposed to make changes to state, a transaction recorder will typically be added to the resource tree so that one does not need to use commands manually. The developer generally does not need to deal with the state of the editor. The tools may be overridden, but reasonable defaults are present.

There are two flavors of Graphiti, one pattern-based flavor that supposedly will alleviate some of the repetitive tasks. However this flavor is completely undocumented as of this writing[1]. Therefore the plain flavor has been used, more precisely Graphiti version 0.7.0.

In conclusion Graphiti provides an easy entrance to creating graphical editors through a simple and contained API.

[1] 13th September 2011

Chapter 5

Requirements for the tool

In the following sections the requirements for the new tool will be established. These will form a foundation for how the tool will be developed, as well enabling the specification of sub-goals that are to be reached after a given amount of time. That is to say it can make the tool development process more structured. The requirements were gathered using the principles described in section 2. For full details of how this was done, see the appendix, section D. A brief overview of PREDIQT, the method the tool should support, follows.

5.1 Overview of PREDIQT

PREDIQT [33] is a method for predicting the effect of architectural design changes on quality of a system. The notion of system quality can typically be based on [23]. The total quality is decomposed into several quality attributes, such as availability, scalability, and security. There is also typically an "other attributes", which is used to achieve model completeness.

The PREDIQT process involves the following steps and sub-steps:

1. Target modeling.
    1.1. Characterize the target and the objectives.
    1.2. Create quality models.
    1.3. Map design models.
    1.4. Create dependency views.
2. Verification of prediction models.
3. Application of prediction models.
4. A new change? If Yes, go to step 3.

In step 1, target modeling, it is assumed that the specifications and design models of the system are made available to the analysis team.

In step 1.1, a high level characterization of the target system is done, as well as defining the scope and objectives of the prediction analysis. The degree of expected design changes are also characterized.

In step 1.2 quality models are created. The total quality is decomposed into system specific quality attributes and their sub-characteristics. Again, this may be based on an ISO-standard such as ISO/IEC 12207.

In step 1.3 the design models found in step 1 will be customized. Only their relevant parts are selected for further use, and a mapping between the low- and high-level design models are also made.

In step 1.4 the dependency views are created. First a conceptual model with the following properties is created:

1. Classes represent elements from the underlying design and quality models.

2. Relations shows the ownership.

3. Class attributes represent the dependencies, interactions and properties.

The result will be a tree-formed class diagram, which will then be used to instantiate a generic dependency view (DV). Then, for each top level quality attribute defined in the quality models, a quality attribute specific DV is created, based on the generic DV. This attribute specific DV will have the form of a weighted dependency tree (WDT). The arcs in the tree will have an estimated impact (EI) attribute, which specifies how much impact the following child will have on the current node. The EIs of a node should always sum to 1. The leaf nodes in the WDT will specify the actual "degree of Quality attribute or characteristic Fulfillment" (QCF). The interior nodes' QCF will simply be a result of it's children's QCF and the EI and thus it will not be specified by any user. It is this DV the tool should primarily support.

The reader is referred to the section D in the appendix, as well as the quoted papers, for further information on PREDIQT.

5.2 Requirements for the PREDIQT tool

The following sections will give specific requirements for the new PREDIQT tool. The new tool, as noted elsewhere, is supposed to replace the Excel spreadsheet tool that is currently being used.

First, before going on to define objectives and requirements, a definition of who the stakeholders are is appropriate. For this tool, the stakeholders is defined to be *the analyst*, *the viewer* and *the software owner*. *The analyst* is the primary user of the tool, i.e. the person applying the PREDIQT method. *The viewer* is a person viewing the data that the analyst is presenting, typically in a group meeting setting. Given that the PREDIQT method may be performed on a software owned by a specific person or company, *the software owner* will also be a stakeholder with some particular interests. All of the following requirements are deduced from these stakeholders' point of view.

5.2.1 Overall requirements

Requirement 1. *The new tool should fully replace the functionality currently needed for DVs that is supported by the Excel spreadsheet tool.*

Requirement 2. *The new tool should fully support the propagation in DVs and calculation methods.*

Requirement 3. *The new tool should be easy to learn and use for an analyst.*

Requirement 4. *The new tool's presentation of data should be easy to understand for a viewer.*

With these overall requirements given, it should be reasonably easy whether to accept a requirement or not: One may ask the question if this requirement improves one or several of the overall requirements. If the answer is yes, then the requirement should be included.

5.2.2 Functional requirements

Requirement 5. *The new tool should support a main view and a small outline view for easy navigation and presentation of the DVs. Both views should support panning.*

Requirement 6. *The new tool should support manual and automatic layout of nodes.*

Requirement 7. *Creating the dependency view tree structure should be simple.*

Requirement 8. *Removal, insertion and direct editing of nodes should be easy for an inexperienced user.*

Requirement 9. *The editor should support copy, cut and paste.*

Requirement 10. *Hiding of nodes should be supported. More specifically all the children of a parent node should be possible to hide. Entire subtrees should also be possible to hide into separated canvases.*

Requirement 11. *Searching for a node by it's name should be supported.*

Requirement 12. *The text and data should always be easy to read.*

The latter requirement implies the following requirements:

Requirement 13. *Zooming of the main view should be supported.*

The latter requirement may enhance presentation of the data.

Requirement 14. *The tool should support resource change tracking.*

If a model is changed in a different tab, it should also be updated, possibly marked as changed or similar in the PREDIQT tool.

5.2.3 Data requirements

Requirement 15. *The data should be persisted on the analyst's computer.*

Requirement 16. *In a given dependency view, the new tool should verify that for all non-leaf nodes, the node's children EIs should sum to 1.0.*

Requirement 17. *For all leaf nodes, the QCF value should be verified to be* $0 \le x \le 1$ *where* x *is the QCF value.*

The two latter requirements both naturally follows from the PREDIQT method as described in its respective papers.

Requirement 18. *The QCF value of internal nodes should be possible to override.*

5.2.4 Technical environment requirements

Requirement 19. *The tool should support traceability with respect to imported models.*

5.2.5 Non-functional requirements

Requirement 20. *It should run on all major operating systems, i.e. Windows, Mac OS X and at least one mainstream Linux variant.*

Requirement 21. *The new tool should be free of cost.*

Chapter 6

Implementation of the tools

This chapter will detail some of the implementation aspects, first giving an overview of the shared code and then further explaining the differences.

6.1 Shared model and code

The business domain was modelled in the Emfatic language, which is a textual language for representing Ecore models.

The model is included below, showing the relative simplicity of the PREDIQT business domain.

```
@namespace(uri="prediqt", prefix="prediqt")
@gmf(f="b")
package prediqt;

@gmf.diagram(rcp="false")
class Project {
   val PNode[*] nodes;
   val PArc[*] arcs;
}

@gmf.node(label="name,qcf", border.width="1",
label.pattern="name={0} qcf={1}", label.icon="false")
class PNode {
   attr String name;
   attr EBigDecimal qcf;
   attr EBigDecimal qcfOverride;
   attr Boolean useQcfOverride;

   ref PArc[*]#target incoming;
   ref PArc[*]#source outgoing;
}

@gmf.link(label="impact", source="source", target="target",
target.decoration="arrow")
```

```
class PArc {
   attr EBigDecimal impact;
   ref PNode#outgoing source;
   ref PNode#incoming target;
}
```

While this model describes the general data requirements of PREDIQT, it does not impose any special limits on the data, such as avoiding cycles. It also does not describe the value propagation.

As explained in chapter 5, PREDIQT has certain requirements for value propagation. This was implemented in the generated model code, using plain Java. As long as the value propagation is done, exactly *how* it is done is not of great importance as long as it would be reasonably self contained, as was the case here. Thus the value propagation was shared among both the GMF-based tool as well as the Graphiti-based tool.

The Graphiti-based tool was handwritten by the author using plain Java. In order to reuse the generator EMF.Editor package for property sheets, the extension point for editors were overridden.

The GMF-based tool on the other hand was generated using the above Emfatic textual model. There were done some modifications to this generated code. These are with marked *@generated NOT* in the source code.

See appendix C.3 for instructions on where to obtain the source code of both tools.

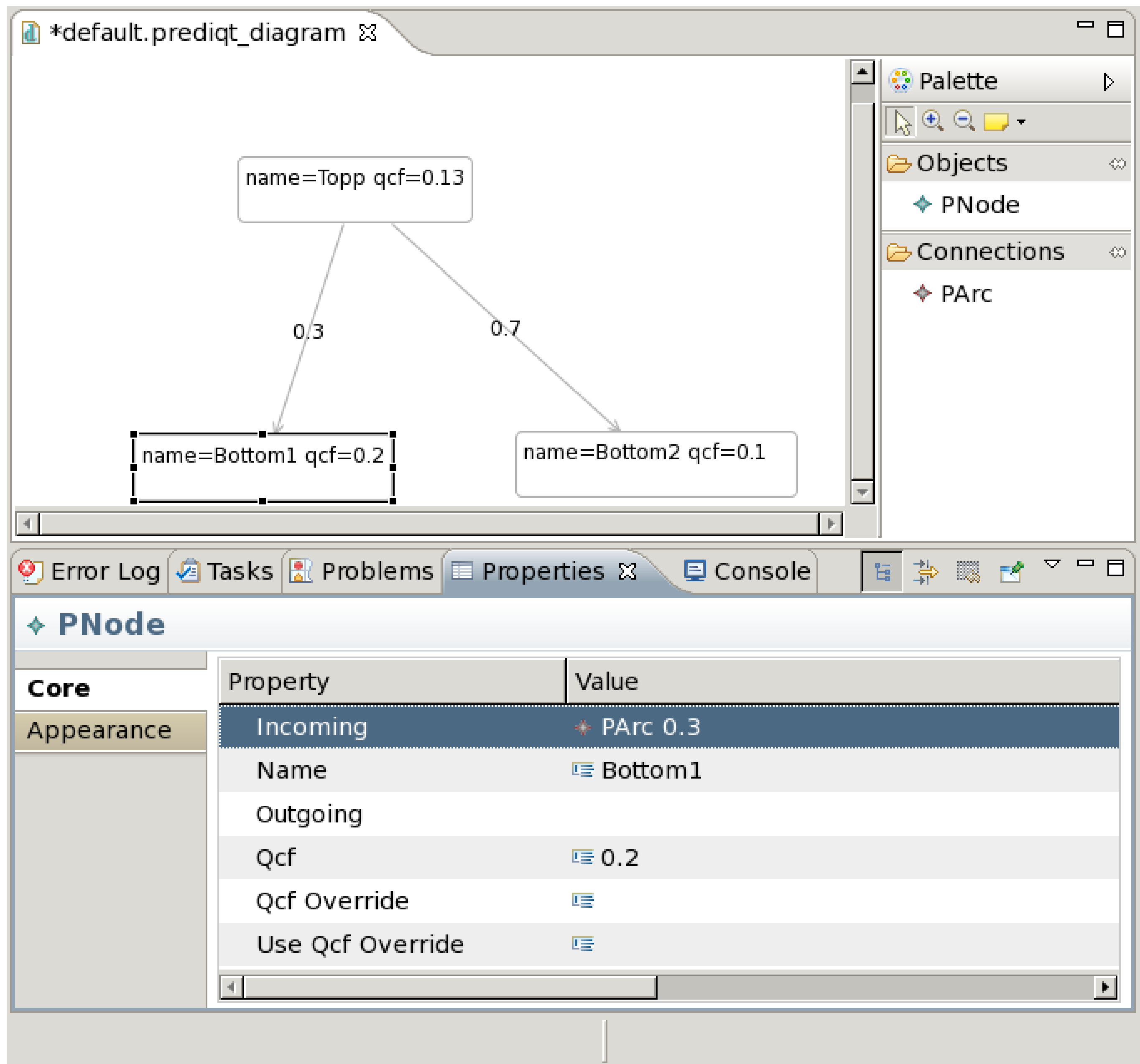


Figure 6.1: Screenshot of the GMF-based tool.

Figure 6.2: Screenshot of the Graphiti-based tool.

Chapter 7

Evaluation

This section will present the evaluation as outlined in section 3.4.

7.1 Evaluation of applicability for supporting tree-based methods

This section will describe to what degree the various requirements are accomplished. Each requirement is evaluated with respect to the two editors. The basis of the evaluation was gathered in the following ways:

- Manually running and testing the two editors separately.
- Reading the official documentation and source code of the two frameworks.
- Drawing from experiences from the development of the two editors.

Rather than manual testing, a better solution would have been to use the SWTBot-tool.

Some of the requirements are rather subjective and these will be treated in a survey.

Requirement 1. *The new tool should fully replace the functionality currently needed for DVs that is supported by the Excel spreadsheet tool.*

Both editors. Since this requirement is more or less a form of overall requirement, it is hard to argue for. However, as will be clear from the following text, there are several things that would likely be easier in Excel. This is valid for both tools. That said, it seems that the GMF-based editor overall supports more features than the Graphiti-based editor.

Requirement 2. *The new tool should fully support the propagation in DVs and calculation methods.*

Both editors. PREDIQT interval support is missing for both editors. The reason for this is two-fold: The author simply did not have enough time. The current business model does not support intervals. However, neither GMF nor Graphiti supports what would have been a natural concept

here: Switching between an interval mode and normal mode. The mode would then affect figures, property sheets and of course value propagation. While it is of course possible to hand code this, it is not easily supported in neither GMF nor Graphiti.

Having separate editors would also have been a possible choice. This would mean one had four editors, GMF-based and Graphiti-based with interval and normal mode. However there was also not enough time for investigating this approach.

While the author is the principal person to blame since not much of an attempt has been made with respect to interval support, both frameworks have large potential for improvement in this field.

Otherwise the support for propagation and calculation methods should be complete. As explained in section 6, both editors share code and this includes the propagation and calculation methods.

Requirement 3. *The new tool should be easy to learn and use for an analyst.* This is evaluated in chapter 7.2 and both tools were found to be equal.

Requirement 4. *The new tool's presentation of data should be easy to understand for a viewer.* This is evaluated in chapter 7.2 and both tools were found to be equal.

Requirement 5. *The new tool should support a main view and a small outline view for easy navigation and presentation of the DVs. Both views should support panning.*

Both editors. Both editors have an outline view. This is called a miniature view in Graphiti. For both editors there is also support for panning in several ways:

- Using the outline view.
- Holding down `space` while using the selection tool.
- Pressing `ctrl+shift` and one of the arrow keys.

It is common to have a "hand" or similar tool to achieve this purpose, but it is not present in either tool. Neither GMF nor Graphiti supports this out of the box.

In conclusion it seems fair to say that while panning is supported, but there is room for improvement for both editors.

Requirement 6. *The new tool should support manual and automatic layout of nodes.*

GMF-based editor. In the GMF-based editor manual and automatic layout of nodes is given in the generated code. However, the automatic layout was not entirely as good as desired, so it needed some small changes. Furthermore only one layout method is supported.

Graphiti-based editor. Automatic layout support is lacking in Graphiti and is not supported directly. However, recipes exists for this, and it was easy to add this support to the Graphiti-based editor.

Conclusion. Both editors supports manual layout out of the box. Adding automatic layout needed some more work for both editors. The resulting layout support is deemed to be equal for both editors.

Requirement 7. *Creating the dependency view tree structure should be simple.* It seems reasonable simple, but a survey is more appropriate for testing to what degree this is the case or not. This is evaluated in chapter 7.2 and both tools were found to be equal.

Requirement 8. *Removal, insertion and direct editing of nodes should be easy for an inexperienced user.*

Both editors. Removal and insertion are equally well supported in both editors.

GMF-based editor. Editing, however, is a different story. Due to the fact that the model used `BigDecimal` as the datatype QCF and estimated impact, the code generated by GMF simply failed. Conversely, the generated code in EMF.Editor worked, so it was still possible to set the values in the property sheet. The fact that EMF.Editor is able to generate proper code for this also makes for the case that this may be supported in GMF in future versions. It is of course possible to fix this in the generated code, but finding the right place was difficult and time consuming.

Graphiti-based editor. In the Graphiti-based editor one had to implement so-called direct editing features, once again in plain Java. Validation was also possible inside the direct editing feature. This was simple.

Conclusion. In conclusion Graphiti provided the easiest way to provide direct editing.

Requirement 9. *The editor should support copy, cut and paste.*

GMF-based editor. Supports duplication of nodes in the generated code. This allows one to essentially copy and paste within a single diagram in one operation. One cannot choose where to put the copied element, i.e. one must drag it after duplication if one wants a different position. A copy of the business object is done. Enabling copy and paste has attracted several recipes[1] as well as a number of newsgroup posts, so it seems like a hard problem in GMF.

Graphiti-based editor. Graphiti supports copy and paste through features. This gives the developer some flexibility with respect to copying the business object or not, while keeping the learning required low. As of September 25th of 2011, pasting relative to the mouse pointer is not supported. See bug number 339525[2]. Pasting relative to the copy

[1] For example http://esalagea.wordpress.com/2011/04/13/lets-solve-once-for-all-the-gmf-copy-paste-problem-and-then-forget-about-it/, retrieved September 25th 2011.

[2] https://bugs.eclipse.org/bugs/show_bug.cgi?id=339525

is supported though, so the functionality is similar to that of GMF as described earlier.

The Graphiti-based editor implemented copying business objects. This was not behaving entirely as expected initially. Furthermore copying between two diagrams behaved slightly unpredictably. See bug number 358818[3] for more information. These problems seems similar to those of GMF.

Conclusion. Both approaches seems somewhat flawed. GMF seems slightly more flawed as it does not support copying between editors. Graphiti provides the lowest learning curve.

Requirement 10. *Hiding of nodes should be supported. More specifically all the children of a parent node should be possible to hide. Entire subtrees should also be possible to hide into separated canvases.*

GMF-based editor. The two first parts of this requirements have been created, but there was a problem in deploying the functionality into the release version of the tool. Furthermore bug 351824[4] were also found. It took considerable amounts of time to find the right place to modify in order to support this feature. The last part is not supported in this editor, but with some effort it should be possible in a GMF-based editor.

Graphiti-based editor. The two first parts of this requirements are not supported. The official Graphiti documentation says that `setVisible(boolean)` method in `PictogramElement` is reserved for future use. Thus it would arguably be hard to support this.

Graphiti is supposed to offer a drill-down feature. This is the same as using separate canvases for the nodes. However, this functionality was not entirely behaving as expected. The editors were marked dirty upon a non-changing update. The color of the text became blue for unknown reasons.

Conclusion. The drill-down feature of Graphiti looks promising, but needs more polishing. The author spent considerable time on the GMF feature, and found the solution to work well. In conclusion GMF stands as offering the best solution.

Requirement 11. *Searching for a node by it's name should be supported.*

GMF-based editor. No support is generated for this to the best of the author's knowledge. That said, it would likely be possible to support this. However, as usual with GMF, finding the right place and learning the right way to do it would likely take a considerable amount of time.

Graphiti-based editor. Graphiti has no search feature. However, if implemented by the framework, it would likely be easy to use. The framework developer would do the hard work once, essentially providing a simple entry into the appropriate Eclipse mechanisms for the framework user.

Conclusion. Neither framework supports this out of the box. Graphiti would likely provide a simple solution to this if it was supported.

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=358818
[4] https://bugs.eclipse.org/bugs/show_bug.cgi?id=351824

Conversely, while it is surely possible to achieve this by modifying the generated code in GMF, it would likely be a difficult task. In conclusion the frameworks are deemed equal.

Requirement 12. *The text and data should always be easy to read.*

Both editors. Given that zooming is supported, this requirement should at least be considered partially supported.

GMF-based editor. The user of the tool can change typical properties such as font color and size, background color and so on. This is supported for the entire canvas as well as per node. Therefore it seems reasonable to consider this requirement achieved. This is supported by the GMF Runtime.

Graphiti-based editor. For the Graphiti-based editor changing of such properties is currently not directly supported. However, Graphiti offers the concept of styles. One style is associated with certain colors, font properties and so on. Several nodes will then share a single style. This allows for a more standardized look and feel.

Conclusion. Graphiti seems to offer a more structured solution to choosing different visual properties on the cost of requiring more work for the programmer. This appears to the author as the best solution in a more professional setting. GMF offers more individual customization as well as global customization while requiring no additional effort. A merge of these properties would be the best and thus there is room for improvements in both tools. As GMF offers considerably more in the generated code and GMF Runtime, it is deemed the best solution.

Requirement 13. *Zooming of the main view should be supported.*

Both editors. Both editors supports this.

Requirement 15. *The data should be persisted on the analyst's computer.*

GMF-based editor. The data is persisted in the XMI format. This is done out of the box by GMF.

Graphiti-based editor. The data is also persisted in the XMI format, but this is not offered out of the box by Graphiti, so it is likely to be more error prone.

Conclusion. The GMF-based editor is deemed to have the most robust solution.

Requirement 16. *In a given dependency view, the new tool should verify that for all non-leaf nodes, the node's children EIs should sum to 1.0.*

GMF-based editor. While this is supported, one needs to manually select "Edit → Validate" in the menu to trigger this functionality. One needs to change the generated code in order to trigger the validation upon save or similar. These errors also appear in the "Problem View" of Eclipse, so it is easy to see if there is an error in the diagram.

It also appears that many different kind of validations can be integrated into a GMF-based editor. Examples include OCL as well as what was used

in this particular editor, the Epsilon Validation Language. This should in other words scale well in a more enterprise setting. On the other hand, this increases the learning curve required.

Graphiti-based editor. A kind of validation in supported in Graphiti. This provided through something called decorators. The node will simply be marked visually if a validation fails. No error will appear in the "Problem View" of Eclipse, and there does not appear to be any attempt to integrate with other technologies such as OCL. This seems fine for small projects, but may be problematic for more enterprise cases. Validation is triggered without any special interaction from the user.

Conclusion. Graphiti offers a low learning curve, but with fewer features. The GMF-based editor offers the best functionality, but at the cost of requiring more learning for the programmer.

Requirement 17. *For all leaf nodes, the QCF value should be verified to be* $0 \leq x \leq 1$ *where x is the QCF value.*

Both editors. The same argument applies here as to the evaluation of the previous requirement, and thus the GMF-based editor is deemed the best.

Requirement 18. *The QCF value of internal nodes should be possible to override.*

Both editors. Both editors supports this through the EMF.Editor. One simply needs to choose "Use Qcf Override" in the properties view, and then setting a value for the qcf override property.

GMF-based editor. It is fairly hard to change the behavior of the generated editor. This also includes changing how properties are set and thus this is deemed a somewhat error-prone process.

Graphiti-based editor. As has been argued already, direct editing of properties in Graphiti is pleasantly simple. Thus, it would be quite simple to implement a direct editing feature supporting conditional writes depending on if the node had children or not.

Conclusion. Including more proper support for QCF override through direct editing would arguably be simpler in Graphiti.

Requirement 14. *The tool should support resource change tracking.*

Both editors. Both editors supports resource change tracking. GMF gives this in the generated code, while Graphiti provides this through update features. Upon a resource change the GMF approach is to simply update the canvas without notifying the user. Thus the user will have to discover the modification on his own. Conversely, in the Graphiti approach, the editor will be marked as dirty upon a update, so the user will know what has been changed. The change can also be undone. This is not possible in the GMF approach.

Conclusion. Graphiti provides the best solution for resource change tracking.

Requirement 19. *The tool should support traceability with respect to imported models.* There is currently no support for traceability or import of other models.

Requirement 20. *It should run on all major operating systems, i.e. Windows, Mac OS X and at least one mainstream Linux variant.*

Both editors. While initial tests are in favor of this requirement, no extensive testing has been done across platforms for neither editor.

For GMF though, it also supports RCP generation. Graphiti has no particular such support and currently relies on the Eclipse IDE.

Conclusion. As GMF offers RCP generation, albeit with fewer features than the Eclipse IDE-targeted generation, it is deemed as the best solution.

Requirement 21. *The new tool should be free of cost.* This is the case for both tools.

Requirement	**Best solution**
1 Replaces Excel-tool	-
2 Propagation of values	Equal
3 Learnability	Equal
4 Easy to understand	Equal
5 Outline	Equal
6 Layout	Equal
7 Simple creation	Equal
8 Editing	Graphiti
9 Copy and paste	Graphiti
10 Hiding of nodes	GMF
11 Search	None
12 Easy to read	GMF
13 Zooming	Equal
15 Persistence	GMF
16 Validation EI	GMF
17 Validation QCF	GMF
18 QCF override	Graphiti
14 Resource change tracking	Graphiti
19 Traceability	None
20 Major OSes support	GMF
21 Free of cost	Equal
Total	GMF: 6, Graphiti: 4

Table 7.1: Overview of the evaluation of the two editors with respect to the requirements.

7.1.1 Conclusion

As can be seen from table 7.1, the GMF-based editor provides the most amount of features. Is this representative? The simple answer is yes, this is in accordance with the author's experience. Thus it is also fair to say that hypothesis 1 holds, albeit not to a very large degree.

However, as will be argued in later sections, changing the generated GMF editor is quite difficult. This is generally not the case with the Graphiti-based editor. That said, from a user perspective the GMF-based editor simply has the most and best features.

7.2 Survey of tools

A survey was held letting primarily IT students test the two tools. The survey is presented in appendix B. For the various tasks, the student was asked to rate how difficult it was to accomplish on a scale from one to six, where one was "very easy" and six was "very difficult".

Half of the students tested the GMF-based tool first and then the Graphiti-based tool. The other half did this in the opposite order. See the appendix for more details. Some of the key findings is presented in the text that follows.

7.2.1 Background of students

The students were asked to rate their own skill in using Eclipse. Unfortunately, this was not a good indicator of their performance. Experienced master students with extensive work experience rated their own skill in Eclipse as completely unskilled. A better indicator would likely have been number of years studying computer science plus number of years working. Nevertheless, the combined rating of both groups were actually identical.

7.2.2 Creation of DV

The main task of the survey was to create the dependency view. In figure 7.1 the results from creating the DV with the first tool is given. One can see that the GMF-based tool was generally found more easy to use. Unfortunately though the perception of "easiness" is subjective. People who were more critical of one tool tended also to be more critical toward the other tool. Nevertheless though, one can see that for both tools few people actually found them difficult, that is to say giving the rating five, to use. The same conclusion holds with respect to the overall learnability. Therefore it seems fair to conclude that both tools are fairly easy to use.

7.2.3 Time used

The time used to solve the tasks was also measured. This is given in figure 7.2 and 7.3. All participants showed improvement with respect to the time used. Again the Graphiti-based tool has the longest time used for the

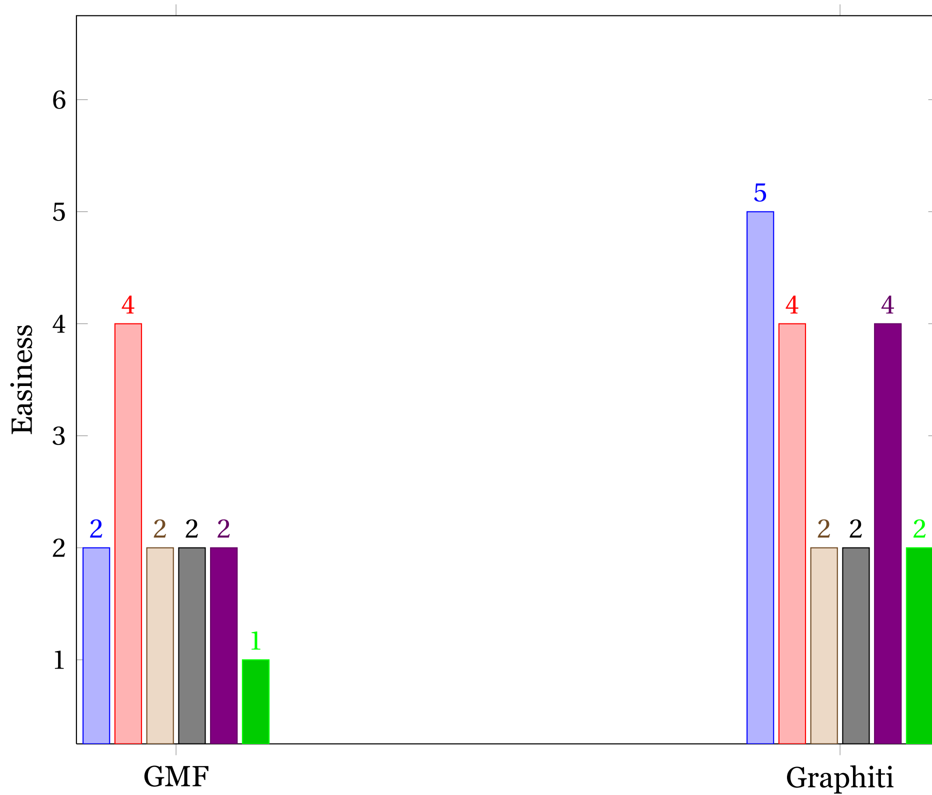


Figure 7.1: How easy it was to create the DV the first time. X-axis: One bar represents one person. Thus this diagram describes 12 different students. Y-axis: “1” means “very easy” and “6” means “very difficult”. One can see that those who used Graphiti as the first editor found it slightly more difficult than the group who used GMF first.

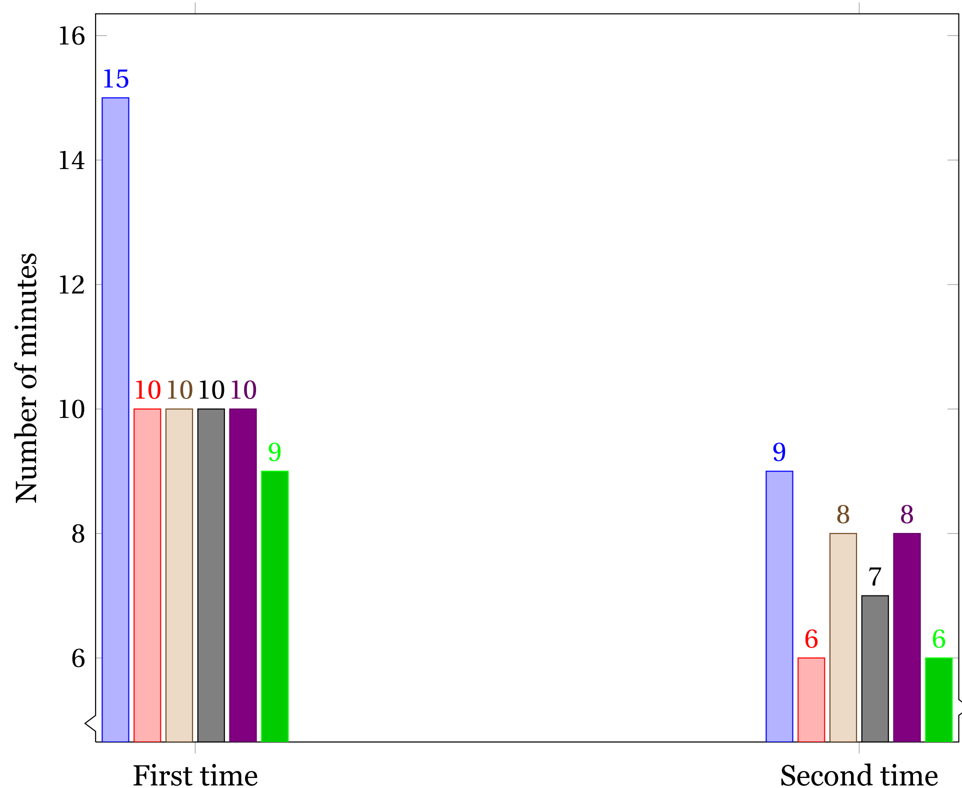


Figure 7.2: Time used to solve the exercises using GMF, then Graphiti. One bar with one specific color represents one person, thus six persons are described in this diagram. In the left area of the diagram the first time is described. In the right area of the diagram the second time is described. Each person has the same color in both the first and the second time. Thus the person who spent 15 minutes on the first editor, then spent 9 minutes on the second editor.

first time users. For the Graphiti-based tool as the second time used, it was the fastest. Thus it seems that this naturally depends heavily on the participants. For example the persons who used 20 and 17 minutes on the first tool were simply not the best computer users. It seems fair to conclude that the tools are reasonably equal with respect to time usage required and that there is a short learning curve. One walk-through seems enough to reduce the required amount of time with a few minutes.

7.2.4 Favorite tool

The last question asked in the survey was to ask the participant to name a favorite tool, that is which editor she/he would prefer in the future. The results of this is given in figure 7.4. It was also possible to not have any favorite, though no participant chose this. One can see that the Graphiti-based tool was marginally more popular.

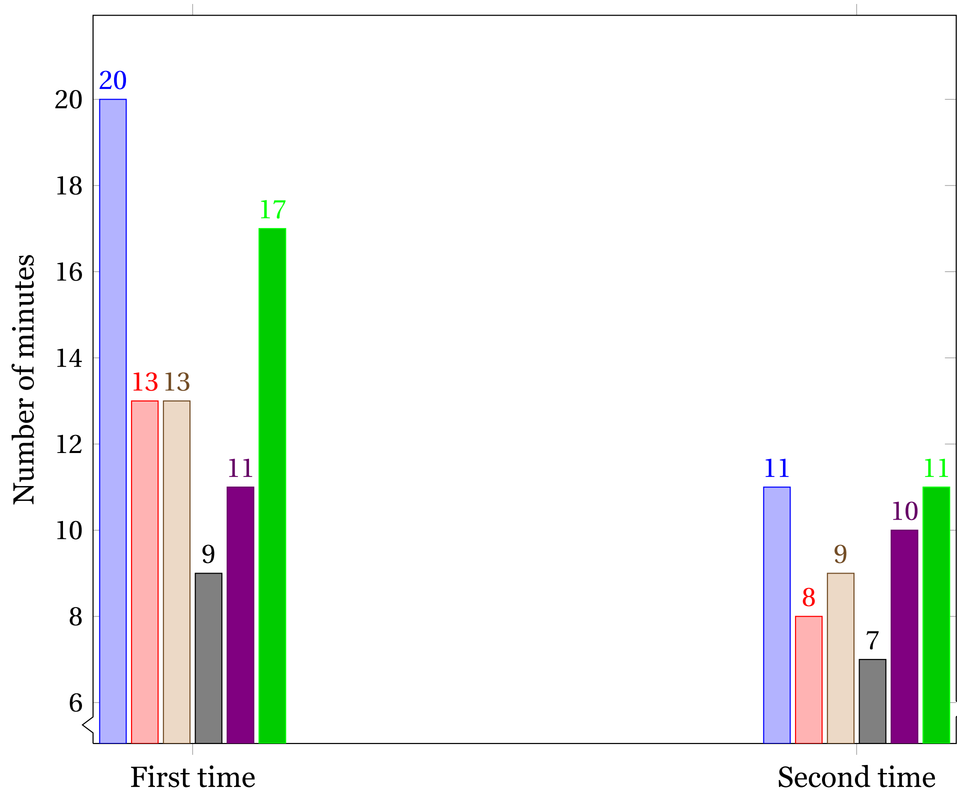


Figure 7.3: Time used to solve the exercises using Graphiti, then GMF. See figure 7.2 for an explanation.

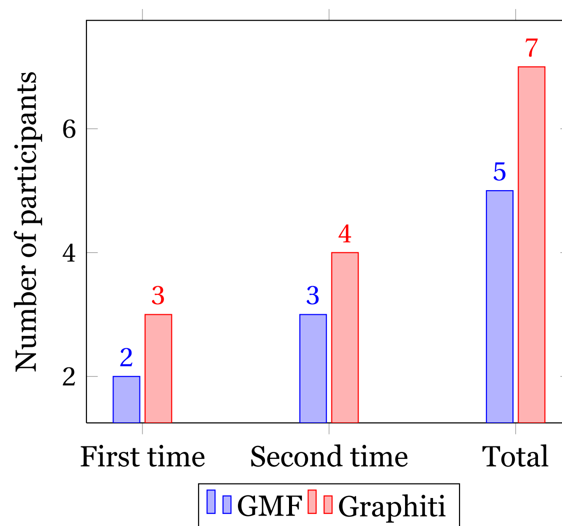


Figure 7.4: Favorite tool.

7.2.5 Feedback during survey

What was equally interesting as the survey itself, was the feedback from the participants during the testing of the tools. It can be summarized as the following:

1. All participants liked the layout of the nodes of the Graphiti-based editor better.
2. Most participants seemed to agree that good documentation was essential in order to perform these tasks.
3. Most participants appreciated the automatic validation of the Graphiti-based editor.
4. Many found the lack of direct editing capabilities in both editors annoying. It seemed very natural to use this for most participants.
5. Many participants made the mistake of linking a node to itself. This was due to lack of experience.
6. Many participants had problems spotting the warning icons of the Graphiti-based editor. Conversely, for GMF, this was not a problem.
7. Several participants found the validation of GMF to be a little confusing. Most of them tried to right-click the canvas.
8. Several participants were annoyed that the Node tool was deselected upon node creation in Graphiti.
9. Several participants had problems with activating the direct editing capabilities of the Graphiti-based editor that did exist.
10. Some participants liked the arcs of the GMF-based editor better due to the larger font.
11. Some participants had problems with undo in both GMF and Graphiti.
12. Some of the participants liked the grid and ruler helpers of the Graphiti-based tool.
13. Some participants preferred the colors of the GMF-based editor, whereas other preferred the colors of the Graphiti-based editor.
14. Some participants tried to use drag-and-drop in the GMF-based editor. It does not work.
15. Three participants found GMF to be slightly quicker than Graphiti.
16. Only one participant discovered and used the mouse-over create arc feature of the GMF-editor. None complained about this feature, finding it confusing or similar.

17. One participant asked if there were any key-bindings.[5]
18. One participant suggested column numbers for the property sheet in order to make the instructions more clear.

This was quite interesting feedback as most of it the author had not thought of. And most of this should likely be relevant for future work on the tools. From the points above 4 and 5 are fixed and described in chapter 7.4.3.

7.2.6 Conclusion

Both tools were roughly equally popular. The characteristics of the participant seems to determine how much time was needed to solve the various tasks. All participants showed improvement over subsequent usage of a similar tool. The layout and automatic validation of the Graphiti-based tool was appreciated. The choice of colors were more of an individual preference. In conclusion it seems fair to say that both tools performed acceptable and roughly equal with respect to requirement 3, 7 and 4. As outlined in 7.2.5 there is room for improvement in both tools.

7.3 Evaluation of development time

As argued in [6] and numerous other papers, the main share of a systems lifetime costs will be spent on doing maintenance. This aspect, i.e. maintenance, will be covered in section 7.4. In this section the focus will be on the time spent creating an *initial* tool, i.e. the initial development time. This is to say that the tool should be working reasonably well, though it is not expected to be perfect. GMF and Graphiti differs considerably with respect to the initial development time, though in different ways.

7.3.1 Graphiti

Graphiti has a very flat learning curve. The various aspects of an editor is laid out for the programmer in a very uniform way, through the feature provider as well the tool behavior provider. Internally Graphiti maps and dispatches between these uniform interfaces and the arguably more complex GEF mechanisms.

This seems all very nice at first, but as one implements a tool, a pattern of repetition occurs. This pattern of repetition is present in the official Graphiti documentation as well. Particularly manually searching for views one created in different features seems like an error-prone process. In the MVC paradigm one would simply store the reference of the particular view in the controller. This can partly be alleviated by giving the views properties, something which is available in the current release of Graphiti. However, this remains essentially a quick fix, not an elegant solution.

[5] This was an Emacs user.

In conclusion, for Graphiti, the larger a business domain gets, the more time one needs to spend on the initial development of the graphical editor. Some features are dependent on other features, making the introduction of changes an error-prone process.

7.3.2 GMF

GMF, as stated in [24], on the other hand, has a very steep learning curve, of which EuGENia is sought to alleviate. While EuGENia does a good job in producing reasonable defaults for the various GMF Tooling models, this generation also needs customization. For this task one is left with the various Epsilon languages. While these are not too hard to accomplish basic tasks with, these languages lack proper tool support including such things as auto completion, debugging facilities and so on, making them require some effort to use. While developing the tool, bugs specific to the EuGENia tooling were also discovered[6], rendering some of the needed customizations impossible. Essentially, if one wants to continue using EuGENia not only as a starting point, but also in further development, one needs to double the effort, i.e. understanding both the GMF Tooling models as well as the usage of the Epsilon languages to modify them.

Furthermore, there are many common options missing in the GMF Tooling models. This should be evident from the sheer number of newsgroup posts discussing modifications of the generated code. One simple example of this is setting the font color. It is currently not supported by the models. Thus, also in the initial development, one is usually left with the need to modify the generated code. If one is not highly familiar with the code generation facilities of GMF or the architecture of GEF, this will likely be a time-consuming task, as experienced both by the author and [39].

For producing a reasonable initial tool, GMF still do provide a fairly decent option. Coupled with EuGENia, one will have a fairly good graphical editor running in a short time. The additional effort one is required to make as the domain model grows is very small compared to Graphiti. Similarly, if one needs to support a new domain, the effort required will be smaller.

7.3.3 Actual development time spent on GMF and Graphiti

The creation of the GMF-based editor was not carefully logged. However, a rough estimate is taken to be that the development took around 3-4 weeks[7] for producing an initial editor. It should be noted that the author at this time had little experience with Eclipse, GEF, Draw2d and so on. Furthermore no attempt at a separate learning phase was done.

Roughly one man week was spent on learning the Graphiti framework from a user perspective. Additionally, roughly another week was spent on learning about JFace, EMF, EMF.Edit and EMF.Editor due to the lack of integration with the latter component in Graphiti. This work is considered

[6]Verified by Dimitrios S. Kolovos, co-author of EuGENia.

[7]A week is here considered 37.5 working hours.

the preparation for the development. At this point the author also had experience with GEF, Draw2d, parts of the Eclipse architecture and similar things, which, while helpful knowledge, is not included as Graphiti-specific knowledge. The development of the tool itself took roughly a half week.

Tooling framework	Learning phase	Development phase	Total time
GMF	-	3-4 weeks	3-4 weeks
Graphiti	2 weeks	0.5 week	2.5 weeks

Table 7.2: Actual development time.

As the background knowledge of the author was quite different when developing each tool, i.e. much stronger when creating the Graphiti-based tool, it seems fair to say that a more realistic estimate of the GMF development time, e.g. had one created the tools in the opposite orders, would be substantially less, possibly as much as two weeks.

7.3.4 Conclusion

As has been argued in the preceding sections, the initial development time will depend heavily on the size of the domain model. The smaller it is, the more sense it would make to use Graphiti. Conversely, the larger the domain model is, the more sense it would make to use GMF. Also, if one is expected to produce editors for many domains, using GMF or possibly a different model driven approach would likely be the best choice with respect to initial development time.

However, Graphiti is currently in incubation phase and this unfortunately still shows. As explained in section 7.1, Graphiti still lacks some of the features present in GMF. Thus, even for the small domain model of PREDIQT, developing a Graphiti-based editor still took considerable time, albeit a little less than the GMF-based editor. The latter fact has much to owe to the fact that the author had more experience when creating the Graphiti-based editor. Thus it seems fair to say that GMF still stands as the quickest way to create a reasonably capable graphical editor. Hypothesis 2 is thus found to be true.

7.4 Evaluation of maintainability

As has been argued before, and seems widely accepted in the literature, software maintenance stands for the bulk of software project costs. Therefore, although this project has not undergone much maintenance, an attempt at will be made at evaluating to what degree each editor is maintainable.

As explained in chapter 2.4, a maintainable system should be easy to correct upon the discovery of errors or deficiencies. Furthermore a system that is maintainable must also be understandable, which concerns issues

such as complexity, dependencies and structuredness. These are related concepts, but will be treated separately in the following text.

7.4.1 Code size

Complexity, though with some caveats, can be measured in terms of lines of code according to [34]. The code size of the various packages are shown in table 7.3. Given that GMF is about six to seven times larger than Graphiti, it should be fair to say that GMF is far more complex than Graphiti. Additionally, a user of GMF will typically need to know GEF and Draw2d as well, so the gap widens. The advanced user of GMF may also wish to look into the code generation (xpt files). Conversely, as a user of Graphiti, the author did not need to know either GEF, Draw2d or any code generation techniques.

Looking at the different tools and their respective code base sizes in figure 7.4, the same pattern holds. The generated GMF tool is far bigger. It should be noted that the tools are not identical, as shown in section 7.1. Despite this, in terms of less complexity, Graphiti seems to be the clear winner here.

Would a hand-written GEF-based editor be smaller than a generated GMF-based editor? Most likely. But would it be as small as a Graphiti-based one? This does not appear to be the case. The author implemented a small GEF-based editor, and the code base quickly grew bigger than that of the Graphiti-based editor even though it had very few features. That it takes a relatively large effort, and therefore also, to a certain degree, a large code base when creating a GEF-based editor, is also clear from the book Graphical *Editing* Framework[37] (author's emphasis): Actual editing of a model is left as a part of the last chapter of the book. With a framework that is focused on editing, this seems rather weak. Based on this it seems fairly safe to say that just about any GEF-based editor, generated or not, would be larger and more complex than a Graphiti-based equivalent.

Package prefix	Total number of classes	Total lines of code
org.eclipse.draw2d	288	52427
org.eclipse.zest	92	18180
org.eclipse.gef	327	58971
org.eclipse.graphiti	637	99306
org.eclipse.gmf	3176	655271
org.eclipse.gmf .xpt files	315	46531

Table 7.3: Code size by package. All lines are counted (code, blank lines, comments, etc.).

7.4.2 Dependencies

The fact that the learning curve is so steep for GMF has been repeated several times already. This is related to dependencies or efferent coupling,

Tool	Total number of classes	Total lines of code
GMF RCP tool	84	13711
GMF Eclipse-based tool	78	12949
Graphiti Eclipse-based tool	18	1381

Table 7.4: Code size by tool. All lines are counted (code, blank lines, comments, etc.).

as one should have an idea of the technologies that the project uses. One way to estimate the amount of dependencies is to look at the import statements used in the top of each Java file. If one groups the imports by xxx.xxx.xxx, one can get an idea of how many different packages one needs to be familiar with if one wants to maintain the application. One can also look at the number of unique imports to get an idea of how much of that package one needs to have knowledge of. This is given in figure 7.5. While this is very similar to efferent coupling, it gives a more detailed picture.

Package	Number of files		Percentage of files		Unique imports	
	Graphiti	GMF	Graphiti	GMF	Graphiti	GMF
org.eclipse.core	3	41	17.6 %	52.6 %	5	30
org.eclipse.draw2d	0	8	0.0 %	10.3 %	0	20
org.eclipse.emf	5	51	29.4 %	65.4 %	7	48
org.eclipse.gef	1	25	5.9 %	32.1 %	1	26
org.eclipse.gmf	0	61	0.0 %	78.2 %	0	153
org.eclipse.graphiti	16	0	94.1 %	0.0 %	66	0
org.eclipse.jface	1	31	5.9 %	39.7 %	5	41
org.eclipse.osgi	0	8	0.0 %	10.3 %	0	1
org.eclipse.swt	0	22	0.0 %	28.2 %	0	15
org.eclipse.ui	2	27	11.8 %	34.6 %	2	41
org.osgi.framework	1	1	5.9 %	1.3 %	2	1

Table 7.5: Imports used by the two tools. GMF and Graphiti should be read GMF-based editor and Graphiti-based editor.

From the figure one can see “Number of files”, meaning the number of files which import at least one class from the given package. The “Percentage of files” gives the relative amount of files which imports the given package. This gives an idea of how widespread the use of the given package is. The last column, “Unique imports”, gives how many unique imports there are from this specific package. This gives an idea of how much one must know or how dependent the editor is on that particular package. The last column is the most interesting measurement.

7.4.2.1 Discussion

For the GMF-based editor one can see that it would be a good idea to master a number of things:

- org.eclipse.core.[commands,expressions,resources,runtime]
- org.eclipse.draw2d
- org.eclipse.emf
- org.eclipse.gef
- org.eclipse.gmf.(runtime.[common,diagram,draw2d,emf,gef,notation])
- org.eclipse.jface
- org.eclipse.swt
- org.eclipse.ui

This requires, obviously, considerable effort.

For the Graphiti-based editor the picture is different. The number of unique imports if fairly negligible. The editor is by far primarily dependent on the Graphiti framework. The reason the Graphiti-based editor has dependencies to org.eclipse.jface and org.eclipse.ui is that the property sheet functionality needed to be changed. This is likely to be fixed in newer versions of Graphiti.

The measurements were also done on a slightly larger Graphiti-based editor, namely *org.eclipse.graphiti.examples.tutorial*, and the results were clear: There were fewer dependencies to outside projects, though they had grown slightly. Additionally a dependency analysis was done using CodePro AnalytiX[8] which gave roughly the same result. Measuring efferent coupling gave 25 for the Graphiti-based editor, and 101 for the GMF-based one.

In conclusion it is fair to say that the GMF-based editor has several magnitudes more dependencies than the Graphiti-based editor.

7.4.3 Making a change to the editors

In order to test the maintainability of the two editors, the author decided to try to fix some of the faults reported by the testers of the tools as described in chapter 7.2.5.

7.4.3.1 Disable linking of a node to itself

For both frameworks, the author had no directly relevant experience in doing such a modification.

In Graphiti, the author was familiar with the create feature that is responsible for creation of business domain objects. Thus this seemed like

[8]http://code.google.com/javadevtools/codepro/doc/index.html retrieved 18th September 2011.

a reasonable place to look. And indeed it was. It was simple to change the *canCreate* method to disallow linking of a node to itself.

For GMF-based editor, while the author had a reasonably good understanding of the GEF architecture, it was more uncertain exactly where to look for this feature. The author's first guess was that it would need to be done where visual feedback was performed. The problem was that it was not immediately clear exactly where this was. However, looking at the files, the author found the *PArcCreateCommand* class, and this had a *canExecute* method. Modifying this was just as simple as modifying the *canCreate* method.

Doing both changes took roughly the same amount of time. However, finding the right place to do the change in GMF was done more by chance than in Graphiti. In conclusion Graphiti is found to be somewhat easier.

7.4.3.2 Enabling direct editing

For both frameworks, the author had some relevant experience in doing such a modification. For Graphiti, the author had already implemented a direct edit feature. For GMF, the author had also fixed a broken direct edit feature earlier.

For Graphiti, again it was quite easy to add direct editing. It is well described in the official documentation and one can also provide feedback if the entered text is correct or not. Fixing the direct editing took about ten minutes.

For GMF it was a different situation. Despite the fact that direct editing was enabled in the generated code, it simply failed and gave no error messages. Neither of [37, 8] explain direct editing to any useful degree, but due to similar experiences with similar problems earlier, the author had some ideas about where to look. After some debugging, it was clear that the transaction for setting the attributes failed. This was because the creation of modification commands produced an invalid *SetCommand*. The quick and obvious fix was to change this code. However, that would rather merely patch the code for exactly this particular context. After some further exploration, the right place was found: *getValidNewValue* in the *AbstractParser* class. This method was then post-fixed with "Gen" and then a fixed method was coded supporting the *BigDecimal* class. There also appeared to be possible to provide an error message here, though the author did not get this working. It took about three hours to find the right place and do the necessary changes.

In the author's opinion this is fairly representative of Graphiti and GMF. While it was a rather easy to make change in Graphiti, it was also specifically tied to that specific attribute of the business domain. That said, it would obviously be possible to make something generic for Graphiti as well.

For GMF it was quite difficult to find the right place to apply the fix. However, at the same time it is not tied to any specific attribute. Thus for all new attributes using this particular data type, it will be supported.

Overall though, it seems fair to say that the simplicity as well as the feedback mechanisms of Graphiti trumps the genericity of GMF.

7.4.4 Structuredness

The structure of GEF is fairly complex, and GEF was named the most complex software framework in [3] of five graphical editor frameworks compared[9]. The architecture is depicted in figure 7.5. The user of GEF, and consequently GMF, needs to understand and use all of these concepts. For the author particularly requests, commands, edit policies and the command stack were difficult to understand.

As explained earlier Graphiti has a very simple API. Its architecture is depicted in figure 7.6. The developer needs to concern himself with the diagram type provider, the feature provider and the many features. Instead of using edit policies, requests, commands and explicitly managing a command stack, one simply uses features. This structure is much easier to understand.

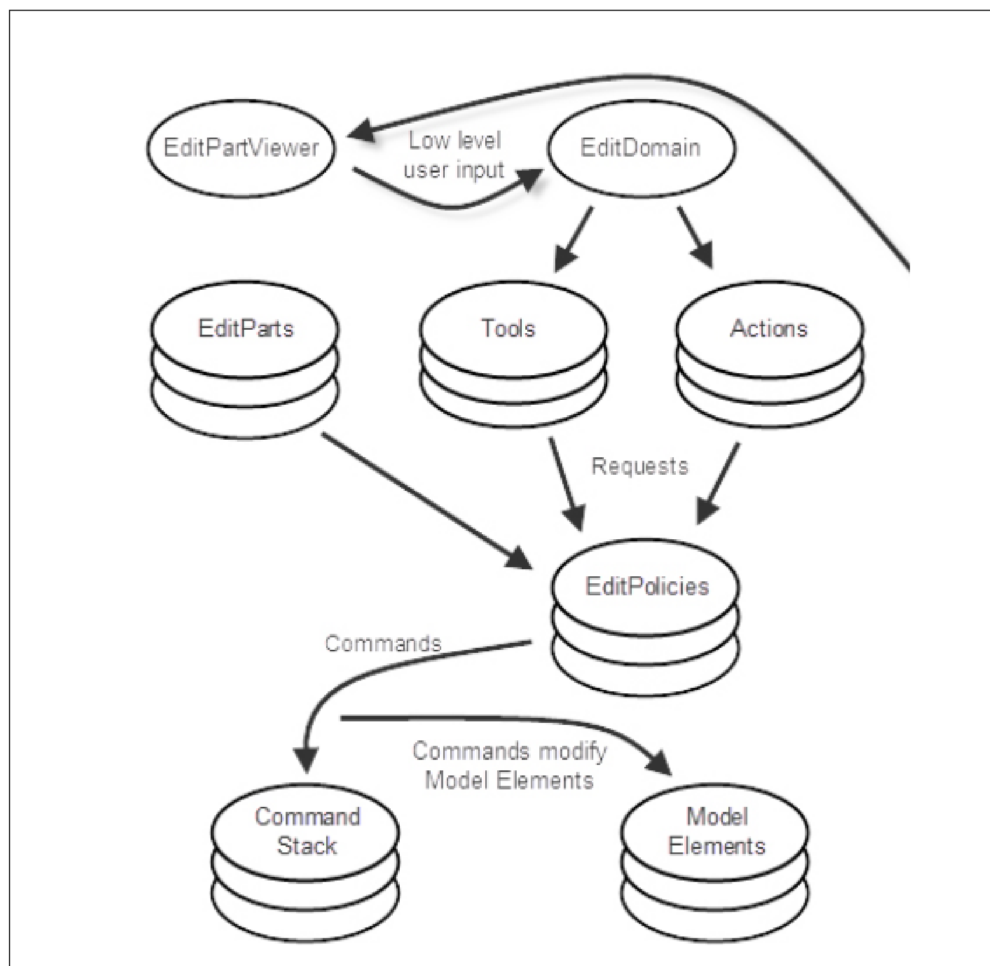


Figure 7.5: GEF architecture. From [37].

While the above argument should be reasonable, how can one measure this more precisely? According to [27] the most central aspect of structured

[9]GMF nor Graphiti included.

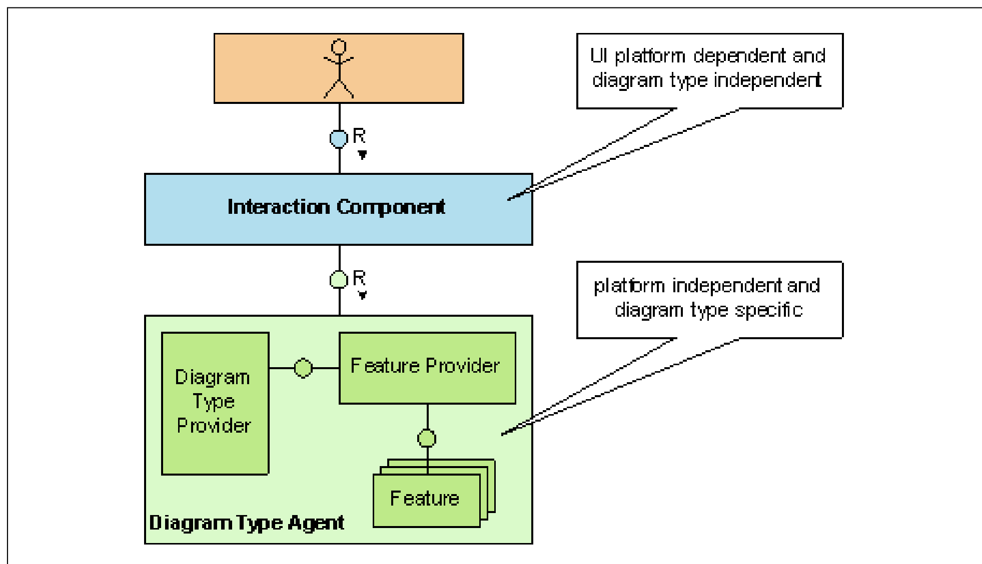


Figure 7.6: Graphiti architecture. From Eclipse help.

programming involves "reducing the number of program paths by imposing a simple program control structure."[10] This sounds similar to cyclomatic complexity as described in chapter 2.4.3. The Graphiti-based editor has an average cyclomatic complexity of 2.28, while the GMF-based editor has an average of 2.30. However, this was measured per method, and thus delegation from class to class or method to method were not taken into account. Thus the author decided to simply place a breakpoint at the former methods that were changed in chapter 7.4.3 and then manually look upwards the stack and note which methods were called before the relevant method was called. The cyclomatic complexity of those methods was then summarized. The following breakpoints were set for both editors:

1. *canExecute* and *canCreate*.
2. The actual create / execute methods.
3. The place where the parsing of the *BigDecimal* took place.

The following restrictions were set for looking up the stack: Only code from the editors plug-ins should be considered. Code possible to override in the plug-in, i.e. if a subclass of an external package is used, should not be considered. Thus this will only concern itself with code that actually is defined in the editor's plug-in. This should likely be somewhat in favor of GMF as it depends more strongly on superclasses that are not generated. For this reason, the stack depth is also given. The top of the stack was considered reached once the top most stack frame from a class of the plug-in was reached. Thus this will also include the inherited code from external packages. The results are presented in table 7.6.

[10] Other aspects are mentioned, but seems not very relevant for the Java programming language.

One can see that the cyclomatic complexity of the GMF-based editor is higher. The results are based on rather few measurements, but are in accordance with the experiences of the author. However, more measurements would have been useful to verify this. The GMF-based editor will simply have much more nested code and a complex structure. Looking at the stack depth gives the same conclusion.

Method description	Graphiti	GMF
Can create arc?	4 (2)	20 (21)
Create arc	5 (2)	2 (3)
Parse BigDecimal	5 (1)	16 (6)

Table 7.6: Nested cyclomatic complexity. Stack depth given in parenthesis.

In conclusion the structure of Graphiti seems much easier to use and understand. It also fulfills the criteria set forth in [27] about structured programming to a higher degree than GMF does.

7.4.5 Conclusion

The Graphiti-based editor appears to do better than the GMF-based editor with respect to code size, fewer dependencies and also a simpler structure. It was also easier, quicker and relied less on mere chance to make changes in the Graphiti-based editor. Thus it should be fair to say that hypothesis 3 is not true.

7.5 Evaluation of customizability

In the following text customizability will be defined to be "the ease with which a program can be modified according to individual requirements."[11] GMF and Graphiti differs considerably with respect to customizability. For a graphical editor, the obvious focus of customizability is the visual aspect, and this will also be the focus in the following text. An overview is first given, then a report of the practical experiences of customization is given. Finally a conclusion is given.

7.5.1 GMF

GMF generates GEF code, which, as has been argued earlier, is of considerable complexity. The approach, compared to Graphiti, is a very different one. One is *expected* to customize the generated code in one way or another. This can be done in a number of ways:

1. Change the emfatic model.
2. Change the graphical definition and mapping model.

[11] Definition based on [27].

3. Override the xpt files.

4. Directly editing the generated code.

5. Using extension points.

6. Changing a factory implementation.

So one actually has six possible places to do the modification. All of these approaches have their disadvantages.

1: Changing the emfatic model is simple, yet it relies on merely annotations and thus could be error prone. In addition there is a fairly low ceiling of what can be customized.

2: In the author's experience, changing the graphical definition and mapping model was rather tedious work. The editors for these models were tree based and not well received by the author. It was somewhat hard to see what was available. Visually parsing examples of models were difficult. In the author's opinion a textual syntax would be preferable. If one does choose to change the graphical definition model and one wants to keep using the emfatic model, one additionally needs to use some Epsilon-based transformation to apply the change. These are loosely typed languages and the editor does not offer any guidance like auto completion or similar things available for the Java language in Eclipse.

3: While the author has only moderate experience with respect to this point, some thoughts are nevertheless presented. To the best of the author's knowledge there does not exist any way to trace back the generated code to the xpt templates. This makes it somewhat difficult to find the correct places to do the modifications. Furthermore one also needs to understand the models of the various GMF tooling elements. Partly this is difficult due to the fact that the editors for these models uses human readable names, and the xpt files does not. Modifying the xpt files requires somewhat more work than modifying the generated code, but it should also be more clear from a glance which code is modified.

4: The author edited the generated code several places. The problem with this method is that it can be somewhat difficult to see where the code was edited in retrospect. For a novice programmer wanting an overview of the code, it may not be intuitive what code was actually changed. This would have been easier had GMF used the generation gap pattern[12].

5 and 6: This merely offers a method for moving the modified code out of the generated package.

7.5.2 Graphiti

Graphiti strives towards completely separating the underlying internals of GEF and Draw2d from the interface one uses. Thus, according to the official documentation, one should only think of GEF and Draw2d as an underlying rendering engine, something that one should not concern oneself with,

[12]http://heikobehrens.net/2009/04/23/generation-gap-pattern/, retrieved 31th October 2011.

making it possible to be replaced with a different technology such as Flash[13]. Thus one is actively discouraged from changing the underlying internals of Graphiti. This is to say that in general it should not be or is not possible to override internal behavior through extension points or similar mechanisms.

As Graphiti is merely based on GEF and hides the underlying technology as much as possible, it is hard to argue that it, theoretically, can offer more customization than GMF and GEF does. This is also evident from the current state of the project: More features do exist in GMF and GEF, and moreover some features, such as layout algorithms, are currently deprecated in Graphiti because they rely solely on GEF.

That said however, unlike GMF, there is only one place to customize the Graphiti-based editor: In the Java files. While repetition may be an issue, doing customization is quite simple.

7.5.3 Practical experiences

For customizing the visual representation of nodes the author encountered no problems with using the Graphiti, whereas the generation models of GMF were not adequate:

- It was difficult to display multiple attributes using EuGENia.
- One could not seemingly show methods, only attributes.
- This required a special printf-like format string.
- The resulting nodes did not look very good. This impression was also confirmed in the survey of the tools.
- One would need fairly good knowledge of GEF layouts in order to fix some of the visual aspects.
- When displaying multiple attributes were accomplished, direct editing failed, see chapter 7.4.3.2 for more details.
- One needed to deal with multiple models, both the mapping and graphical definition model.
- The editors used to edit the models were basic tree-editor models. These were rather cumbersome to use. A textual approach would be preferable to the author as well as probably many programmers.

Consider how this was done in Graphiti: One simply used plain Java, creating the view models fairly straightforward and setting attributes of the views with plain Java code as well. This was simply much easier.

[13] In practice though, currently only one rendering engine exists[14].

7.5.4 Conclusion

In conclusion it seems fair to say that Graphiti will provide the most customizability, particularly for the novice developer, and thus hypothesis 4 is not found to be true.

7.6 Evaluation of criteria set forth in Myers et al

As outlined in 2.3.1, [31] lists several recurring themes that are central to determine if a user interface framework will be successful or not. These will be evaluated with respect to GMF and Graphiti in the following text.

7.6.1 Specificity

Graphiti is primarily concerned with what happens or is possible to do on the canvas, not so much with anything else. GMF on the other hand seems to have a more wider focus, both in terms of features, but also in terms of leaving the developer with more choice with respect to how the editor can be architected. This will likely benefit the expert developer.

In conclusion it is hard to name a winner. For a simple and small editor, Graphiti would likely be offering the most specific solution, whereas on a more complex editor, requiring a property sheet, navigator and so on, GMF would likely have the most specific solution.

7.6.2 Threshold and ceiling

As has been argued earlier in this chapter, there are big differences here. As a quick recap, Graphiti is the clear winner in terms of low threshold. However, this is at the cost of a medium high ceiling.

GMF has a high threshold and also a very high ceiling. The complexities of SWT, Draw2d, GEF and Eclipse RCP are not hidden away, given the developer a high ceiling, but also high threshold.

As with the previous theme, for smaller systems Graphiti would likely be the best solution, whereas for a very large editor, where high threshold would not be considered a problem, GMF would likely provide a better solution. Again it is hard to name a clear winner.

7.6.3 Predictability

What does Myers mean when he talks of predictability? To quote the article: "Tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers. (...) In fact, because heuristics are often involved [in automatic and model-based techniques], the connection between specification and final result can be quite difficult to understand and control."

As Graphiti does not use any generative techniques, it is hard to argue that it is anything but predictable with respect to the context given above.

GMF is relevant as it uses code generation. Does Myers criticism apply to GMF? In the experience of the author the simple answer is no. The code generation is done straight forward using an imperative style generation language. It simply does not behave unpredictably.

Thus, once again, there is no clear winner in terms of this particular kind of predictability.

7.6.4 Path of least resistance

[31] gives the following elaboration on this theme: "Tools influence the kinds of user interfaces that can be created. Successful tools use this to their advantage, leading implementers toward doing the right things, and away from doing the wrong things."

Currently GMF offers more features and thus leads the developer in a good direction more of the time. What about resistance? GMF does not try to hide away any complexities, so there is considerable resistance if one wants to modify what has been generated.

Graphiti on the other hand has less features, but essentially wraps several existing features of the underlying frameworks in an easily accessible manner. Seen from this perspective, Graphiti is more or less a big facade pattern.

Documentation is also important and the same pattern also emerges here. GMF has more documentation, but it is scattered over the official help, several wiki pages aiming at the same topic and one book. One is usually left searching for a specific recipe if one is not an expert user. In a sense, it can be overwhelming and one is not sure where to look.

This is not the case with Graphiti. More or less all documentation is in the official help, nicely organized. This is very helpful for novices.

Given the above reasoning, the author finds Graphiti the best with respect to the path of least resistance.

7.6.5 Conclusion

For predictability there is no clear winner. Both frameworks are deemed equal. For specificity and threshold and ceiling it depends on the context. A larger and more complex editor will likely be more fit for GMF, and conversely a smaller and less complex editor will be more fit for Graphiti. For path of least resistance Graphiti is deemed the best largely due to the fact that its features are simple and very well documented. Thus hypothesis 5 is not found to be true, albeit not to a very large degree.

7.7 Related work

To the best of the authors knowledge, there is only a single other comparison of Graphiti and GMF, namely [1]. The comparison was given at EclipseCon 2011 and its conclusions are in accordance with this thesis:

Which framework to use depends on the context. GMF takes more time to learn and use, but offers more features and flexibility than Graphiti.

Spray, the model driven approach to Graphiti, will have its first presentation at EclipseCon Europe November 3rd 2011[15], a few days after this thesis should have been delivered. The first version of Spray was released during the writing on this thesis, but currently is in a early stage and lacks much documentation and tutorials.

A number of other papers give some criticism towards GMF, but neither seems completely comparable for the context of this thesis. Nevertheless, they are summarized below.

7.7.1 Lack of complex editing operations

[11] states that "The disadvantage of the Eclipse approach to visual editor generation based on EMF/GEF and the GMF project, is that the underlying meta-model (i.e. the EMF model) mainly defines the visual language alphabet. Therefore it may be the case that an editor based on this model allows the editing of diagrams which are not valid in the VL [Visual Language]." The goal of this paper was offering more complex editor operations through the introduction of a new tooling environment called TIGER. This theme is also explored in [47]. For this thesis' scope though, the editor operations given by both GMF and Graphiti were largely sufficient. There was also not encountered any problems with invalid diagrams.

7.7.2 Migrating from XML/UML to Xtext/GMF

[13] presents a practitioner report of migrating a large modeling environment from XML/UML to Xtext/GMF. There is not any direct criticism of GMF here.

It is however noted that "An interesting aspect of this approach [migrating to a textual syntax] is that it eliminates the need to have the GMF editor cover the complete semantic model. The user can always switch to the textual syntax if the GMF editor does not (yet) support a certain syntax. *This is interesting since the effort of implementation per concept (meta-element) is much lower for Xtext than it is for GMF.*" (author's emphasis). It is not clear from the context if this meta-element is a part of the GMF-models or plain code. If it is the former, it seems reasonable it would require more effort in Graphiti. Spray is created using text and has a textual syntax, so it is quite possible that it would require less effort in Spray.

Among the authors and participants in [13] report is employees of itemis. This company is heavily involved in a number of Eclipse projects, employing several full-time Eclipse committers, so it should not be very surprising that the project were successful. Some mild praise of GMF is also given: "Compared to the UML tool the GMF editors are much less cluttered since they only provide the UI elements that are actually needed

[15] http://www.eclipsecon.org/europe2011/sessions/spraying-natural-way-create-graphiti, retrieved 5th October 2011.

by the domain-specific language." This would also hold for any Graphiti-based editor also.

Some of the main people behind Spray is also from itemis, so this at least should indicate that Graphiti can be used as at least a back end in large projects in the future. Curiously, this is what Karsten Thoms, head developer of Spray and employed at itemis, wrote about GMF in a newsgroup post: "But we want to provide easier [than the GMF tooling], [a] DSL based tooling. If you have used GMF tooling you know how difficult it is to use, and becomes almost unusable for larger editors. The GMF runtime is OK, but the tooling is bad and almost dead."[16] Additional evidence that something of a switch from GMF to Graphiti/Spray is happening is that two of the committers on gmftools, both of itemis, are now committers on the Spray project. The gmftools tools project had its last commit in October 2010, whereas Spray is actively developed. For gmftools this may simply indicate that the project is mature and does not need updates. However, it also reflects that at least itemis is putting considerable effort behind Spray.

7.7.3 Difficult to use GMF

By "use" it is here meant not only initial creation of the first editor, but also the customization of the generated editors. This will also be clear from the quoted papers.

That GMF is difficult to use is stated in [46]: "...our experiments revealed that GMF is quite complex, and not as user friendly as we had like it to be. (...) So refining the intermediate models can be painstaking and requires a good knowledge of the underlying meta-models of GMF. (...) When the user diverts from the GMF standard DSL specifications (non-default solution), the user has to hand-code what he really wants." This is accordance with this thesis' experiences as well.

[5] states that "GMF illustrates its ability to generate the code of powerful tools, but the resulting tools remain stereotyped and cannot be easily personalized." Exactly what is meant by personalized is not too clear, but this seems to be similar of what this thesis refers to as customizability.

While [39] does not explicitly state that GMF is hard to use, it does state that "It was not uncommon to spend days to search for the right place to edit, only to change a couple of lines here and there." So it seems fair to conclude that it deems GMF as difficult to use.

In [12] it is stated that "(...) we also faced challenges such as the high level of expertise required to develop a good enough language and tool, the shortcomings of the tools in providing support for modeling at different abstraction levels, and the difficulties in updating the modeling tool with changes in the metamodel." Again, GMF is deemed difficult to use.

[4] states that "(...) customization [in Eclipse GMF and Microsoft DSL Tools] is still a challenging task."

[16] http://groups.google.com/group/spray-dev/msg/a54db1650b8b9ecd, retrieved 7th October 2011.

While generally positive towards GMF and MDSD, [25] states that the development effort of creating a GMF-based editor is “medium to high (depending on the degree of customizations).” So it seems that customization requires considerable development effort.

In [35] two groups of students are set to create a graphical editor with the help of GMF and Microsoft DSL tools. While GMF is deemed the best framework for this task, only 12% of the students found GMF easy to use.

Yet more evidence of the difficulties of GMF is found in [30]: “(. . .) if we used e.g. GMF to generate code as far as possible, GEF apprentices without deeper knowledge of the mechanisms in GEF would surely struggle when laying hand on the generated code to extend it with complex features.”

Also telling of the complexities of GMF is found in [9], which builds additional functionality working with GMF, but nevertheless states that: “The existing GMF infrastructure is obviously rather complicated: it consists of a number of metamodels, libraries, generators, model transformations of industrial scale.” If expert GMF-users, who it should be fair to label the persons involved in creating extensions to GMF, also find GMF hard, it should hardly be surprising that novice programmers find GMF hard as well.

This section will conclude with the remarks of [24]: “It is widely accepted that implementing a visual editor using the built-in GMF facilities is a particularly complex and error-prone task and requires a steep learning curve.”

7.7.4 What does GMF say about itself?

Continuing on the trend of the latter section, it is interesting to see what GMF says about itself. This is to say what GMF committers or presenters of GMF have to say about it.

The author looked through the presentations given at the Eclipse conferences from 2008 to 2011 as given by the GMF wiki[17]. The selling points of GMF are typically high quality code generation, a good runtime and so forth. This is fine, but is it a selling point that GMF is easy to use? Not generally.

Of the various EclipseCon presentations on GMF, only [15] makes the explicit statement that “it was quick and easy” to generate the diagram editor. However, the fact that the code was customized was a separate point, so that it was “quick and easy” to generate the diagram editor, seems not to include the customization. Interestingly Markus Voelter was one of the authors of this presentation. Voelter has written books such as [42] and a number of other books on patterns and thus should obviously be regarded as an expert on MDE and similar topics. A year after the presentation was given, in 2009, he wrote that “GMF is still awful [sic]” in a blog post[18]. So, it seems that even the experts are having trouble with GMF. In this blog

[17]http://wiki.eclipse.org/Graphical_Modeling_Framework/Documentation#Presentations, retrieved 8th October 2011.

[18]http://voelterblog.blogspot.com/2009/06/gmf-is-still-awful.html, retrieved 8th October 2011.

post he also suggests the creation of a textual language for the creation of graphical editors. This is obviously exactly the aim of Spray.

7.7.5 Conclusion

Several papers have criticized GMF for not offering some particular feature that would make a part of the developers' life easier. These typically propose some new abstraction on top of or as an addition to GMF, or a new tooling environment as a replacement of GMF. For this thesis' tool scope, the features offered by GMF were generally sufficient, and thus only a few papers of this type are cited.

That GMF shortens initial development time compared to manual implementation seems also to be widely accepted.

What has been a recurring theme of this thesis though is the complexity of GMF and GEF with respect to customization. This also seems in accordance with the papers quoted in section 7.7.3. As argued earlier, that GMF is easy to use is not a selling point and even some experts find it hard.

What relevance does this have to Graphiti? Would the author expect to find numerous papers noting the complexities of Graphiti in the coming years? Certainly not. While the author is not familiar with the pattern flavor of Graphiti, the plain flavor certainly seems easy to understand.

In conclusion, for GMF, it seems to be widely accepted that initial development time will be short, the generated code will be of high quality, further development and customization is generally difficult yet required and thus maintainability will also be somewhat difficult.

Chapter 8

Conclusion

This thesis has evaluated the GMF and Graphiti frameworks primarily based on the experiences from the development of two editors supporting the PREDIQT method.

Which framework is better, GMF or Graphiti? As argued throughout this thesis, it depends, both on the expertise of the programmer and the size of the domain model. For the end user, GMF currently offers more features than Graphiti does. Thus hypothesis 1 was found to be true, though not to a very large degree. If development with Graphiti continues, this is expected even out in the future.

For the programmer, GMF offers a quick initial editor with many features. The larger the domain model is, the more time can be saved using GMF as compared to Graphiti. Thus GMF will be the best choice with respect to initial development time and thus hypothesis 2 was found to be true.

With respect to maintainability, Graphiti is the clear winner. Its architecture is very much simpler than the editor generated by GMF. Thus hypotheses 3 was found not to be true.

For customization, Graphiti is also found to be the best solution. This, however, is a less clear finding. As this tool is primarily a diagramming tool, the focus of customization is on visual aspects of the diagram. This was, indeed, quite easy in Graphiti compared to GMF. Thus hypotheses 4 was found not to be true. For what is external to the diagram, however, GMF is expected to offer the most choice, though requiring the developer to learn the various parts of Eclipse RCP.

For the criteria set forth in [31], the frameworks are equal on most criteria. However, in the author's opinion, Graphiti offers less resistance with its simple architecture and well written and clear documentation. Thus hypothesis 5 was found not to be true.

In section 7.7 related work was reviewed. While currently very little is written about Graphiti, many studies have been conducted with respect to GMF. The author's findings in this thesis were, in general, confirmed. The statements above about the various aspects of GMF were similar to that of the reviewed papers.

In conclusion, as has been argued in the preceding text, Graphiti

appears to have somewhat more benefits, particularly if one does not mind the extra initial development time. Thus, for the average programmer and a domain model that is not very large, the author recommends Graphiti over GMF if one is interested in attaining maintainability, customizability and a reasonably low learning curve.

8.1 Further work and suggested improvements

More evaluations of GMF and Graphiti should be done to further strengthen or weaken this thesis' findings. As Spray becomes more mature, it seems like a natural candidate for comparison with GMF.

For supporting value propagation and tree calculation methods in general, it would be interesting to see if Xcore, the textual syntax for Ecore models incorporating the Xbase language, could make such things more concise. A separate DSL could also be an option.

For the author, the feeling of being "lost" inside the code generated by GMF and simply searching for recipes to a specific problem was a somewhat frustrating experience. Based on the talks the author has had with various students taking the "INF5120 - Model Based System Development"[1] course, this feeling is by no means unique for the author. Incremental development using Graphiti one felt more productive. It would be interesting to see more research in this field.

The complexity metrics used in this thesis were somewhat lacking. These were usually calculated statically and only considered a method or class at a time. Thus a project that is split up into many files and methods would perform well on many metrics. For measuring the overall complexity of a project, it would be interesting to see metrics taking into account the runtime structure of the project.

For GMF, it would be good to have an overview of what prerequisite knowledge one should have before starting to create a GMF-based editor. This should include which topics one should learn, how long it would approximately take to learn this as well as which resources one should use. An attempt at this is given in appendix A. This would enable developers to make a more realistic judgment if using GMF is a good idea or not.

Similarly, for Graphiti users, hitting the ceiling of what is possible inside Graphiti might be a problem. Documentation on the internals of Graphiti would then be very helpful. In a similar spirit to the suggestions above, a guide of what one should learn would be helpful.

[1] http://www.uio.no/studier/emner/matnat/ifi/INF5120/index-eng.xml, retrieved 24th October 2011.

Bibliography

[1] Koen Aers. Graphiti and gmf compared: Revisiting the graph editor. In *EclipseCon 2011, Santa Clara, California*, march 2011.

[2] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction (Cess Center for Environmental).* Oxford University Press, 1977.

[3] Daniel Amyot, Hanna Farah, and Jean-François Roy. Evaluation of development tools for domain-specific modeling languages. In *System Analysis and Modeling: Language Profiles*, volume 4320, pages 183–197. Springer Berlin / Heidelberg, 2006.

[4] Turhan Özgür. Comparison of microsoft dsl tools and eclipse modeling frameworks for domain-specific modeling in the context of the model-driven development. Master's thesis, Belkinge Institute of Technology, Ronneby, 2007.

[5] Olivier Beaudoux, Arnaud Blouin, and Jean-Marc Jézéquel. Using model driven engineering technologies for building authoring applications. In *Proceedings of the 10th ACM symposium on Document engineering*, DocEng '10, pages 279–282. ACM, 2010.

[6] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition).* Addison-Wesley Professional, 1995.

[7] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley, 1996.

[8] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins (3rd Edition).* Addison-Wesley Professional, 2008.

[9] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated co-evolution of gmf editor models. In *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin / Heidelberg, 2011.

[10] Tore Dybå, Barbara A. Kitchenham, and Magne Jorgensen. Evidence-based software engineering for practitioners. *IEEE Softw.*, 22:58–65, 2005.

[11] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 134–143. ACM, 2005.

[12] Andy Evans, Miguel Fernández, and Parastoo Mohagheghi. Experiences of developing a network modeling tool using the eclipse environment. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 301–312. Springer Berlin / Heidelberg, 2009.

[13] Moritz Eysholdt and Johannes Rupprecht. Migrating a large modeling environment from xml/uml to xtext/gmf. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 97–104. ACM, 2010.

[14] N.E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675 – 689, sep/oct 1999.

[15] Tatiana Fesenko, Radomil Dvorak, Bernd Kolb, and Markus Voelter. Using gmf and m2m for model-driven development. In *EclipseCon 2008*, 2008.

[16] C. Gacek and B. Arief. The many meanings of open source. *Software, IEEE*, 21(1):34 – 40, 2004.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009.

[19] Oyvind Hauge, Thomas Osterlie, Carl-Fredrik Sorensen, and Marinela Gerea. An empirical study on selection of open source software - preliminary results. In *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '09, pages 42–47, Washington, DC, USA, 2009. IEEE Computer Society.

[20] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[21] ISO/IEC. *ISO/IEC 17799. Information technology – Code of practice for information security management*. ISO/IEC, 2000.

[22] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[23] ISO/IEC. *ISO/IEC 12207. Information technology – Software life cycle processes*. ISO/IEC, 2002.

[24] D.S. Kolovos, L.M. Rose, R.F. Paige, and F.A.C. Polack. Raising the level of abstraction in the development of gmf-based graphical model editors. In *Modeling in Software Engineering, 2009. MISE '09. ICSE Workshop on*, pages 13 –19, may 2009.

[25] Klaus Krogmann and Steffen Becker. A case study on model-driven and conventional software development: The palladio editor. In *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 169–176. GI, 2007.

[26] Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. Predicting oss trustworthiness on the basis of elementary code assessment. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 36:1–36:4, New York, NY, USA, 2010. ACM.

[27] James Martin. *Software Maintenance: The Problem and Its Solution*. Prentice Hall, 1983.

[28] Joseph Edward McGrath. *Groups: Interaction and Performance*. Prentice-Hall, Inc., 1984.

[29] Nancy R. Mead and Ted Stehney. Security quality requirements engineering (square) methodology. *SIGSOFT Softw. Eng. Notes*, 30:1–7, 2005.

[30] Tony Modica, Enrico Biermann, and Claudia Ermel. An eclipse framework for rapid development of rich-featured gef editors based on emf models. In *GI Jahrestagung*, pages 2972–2985, 2009.

[31] Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7:3–28, 2000.

[32] Aida Omerovic, Anette Andresen, Håvard Grindheim, Per Myrseth, Atle Refsdal, Ketil Stølen, and Jon Ølnes. A feasibility study in model based prediction of impact of changes on system quality. Technical Report A13339, SINTEF ICT Norway, 2010.

[33] Aida Omerovic, Anette Andresen, Håvard Grindheim, Per Myrseth, Atle Refsdal, Ketil Stølen, and Jon Ølnes. Idea: A feasibility study in model based prediction of impact of changes on system quality. In *Engineering Secure Software and Systems*, volume 5965 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin / Heidelberg, 2010.

[34] Andy Oram and Greg Wilson. *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, 2010.

[35] Vicente Pelechano, Manoli Albert, Javier Muñoz, and Carlos Cetina. Building tools for model driven development. comparing microsoft dsl tools and eclipse modeling plug-ins. In *DSDM'06*, pages –1–1, 2006.

[36] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering: Theory and Practice (4th Edition)*. Prentice Hall, 2009.

[37] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF) (Eclipse Series)*. Addison-Wesley Professional, 2011.

[38] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The sqo-oss quality model: Measurement based open source software evaluation. In *Open Source Development, Communities and Quality*, volume 275 of *IFIP International Federation for Information Processing*, pages 237–248. Springer Boston, 2008.

[39] Fredrik Seehusen and Ketil Stølen. An evaluation of the graphical modeling framework (gmf) based on the development of the coras tool. In *Theory and Practice of Model Transformations*, volume 6707, pages 152–166. Springer Berlin / Heidelberg, 2011.

[40] Alan Shalloway and James Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison-Wesley Professional, 2001.

[41] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2007.

[42] Thomas Stahl and Markus Voelter. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley, 2006.

[43] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework (2nd Edition)*. Addison-Wesley Professional, 2008.

[44] Klaas-Jan Stol and Muhammad Ali Babar. Challenges in using open source software in product development: a review of the literature. In *Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, FLOSS '10, pages 17–22, New York, NY, USA, 2010. ACM.

[45] Davide Taibi, Luigi Lavazza, and Sandro Morasca. Openbqr: a framework for the assessment of oss. In *Open Source Development, Adoption and Innovation*, volume 234 of *IFIP International Federation for Information Processing*, pages 173–186. Springer Boston, 2007.

[46] S. Temate, L. Broto, A. Tchana, and D. Hagimont. A high level approach for generating model's graphical editors. In *Information Technology: New Generations (ITNG), 2011 Eighth International Conference on*, pages 743 –749, april 2011.

[47] Rayner Ron Vintervoll. Modeling editing behavior for editors of graphical languages. Master's thesis, The University of Oslo, Oslo, 2010.

[48] Jens von Pilgrim and Kristian Duske. Gef3d: a framework for two-, two-and-a-half-, and three-dimensional graphical editors. In *Proceedings of the 4th ACM symposium on Software visualization*, SoftVis '08, pages 95–104. ACM, 2008.

[49] Craig Walls. *Modular Java: Creating Flexible Applications with Osgi and Spring (Pragmatic Programmers).* Pragmatic Bookshelf, 2009.

[50] M.V. Zelkowitz and D.R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23 –31, 1998.

Appendix A

Additional lessons learned

In the spirit of [50], this chapter will describe some of the lessons learned while writing this thesis that are not directly relevant to the topic of this thesis. Thus it is put in the appendix. Nevertheless it should be useful as a guide for students wanting to explore the same topics as this thesis. In other words the mistakes of the author need not be repeated. Simply put this chapter will outline how one should go about learning skills useful for GMF and Graphiti. It will be presented in chronological order: What one should learn first will be described first. The student is assumed to be reasonably familiar with Java.

A.1 Faster programming feedback cycle: Learn OSGi

A.1.1 Problem

One of the things that made development hard, was the long feedback time. Particularly before one has a certain knowledge of a framework it is beneficial with a short feedback time between editing a file and seeing the results. The author typically left-clicked the project, selected "Run As" and then "Eclipse Application". The newly launched Eclipse instance will be called the *testing instance*, whereas the one which was used to launch will be called the *development instance*. The issue here is not the use of the mouse, but the long start up time of the testing instance. While the development instance will do some hot code swapping, this only goes so far. With a larger change of the code, one needs to shut down the testing instance one ran and restart it again. For small changes, which is typical in the learning phase of a framework, this makes development quite slow.

A.1.2 Solution

Eclipse is based on OSGi and Equinox. As described earlier, OSGi should bring modularity to the Java platform. So it should be possible to simply refresh the project one is working on. And indeed it is possible, however

the author has yet to see this described explicitly in the context of Eclipse in any tutorial or book.

The following should be done in the testing instance of Eclipse:

1. Select “Window” → “Show View” → “Other…”.
2. Select “Console” and press “OK”.
3. In the right corner of the console view, click the icon which has “Open Console” as tool tip text.
4. Select “Host OSGi Console”. The console should now display “`osgi>`”.
5. Type “`ss yourbundlename`” to find your bundle.
6. Type “`refresh bundleid`” where bundleid is the numeric id from the results of the previous command.
7. The bundle should now be reloaded.

See figure A.1 for a screenshot.

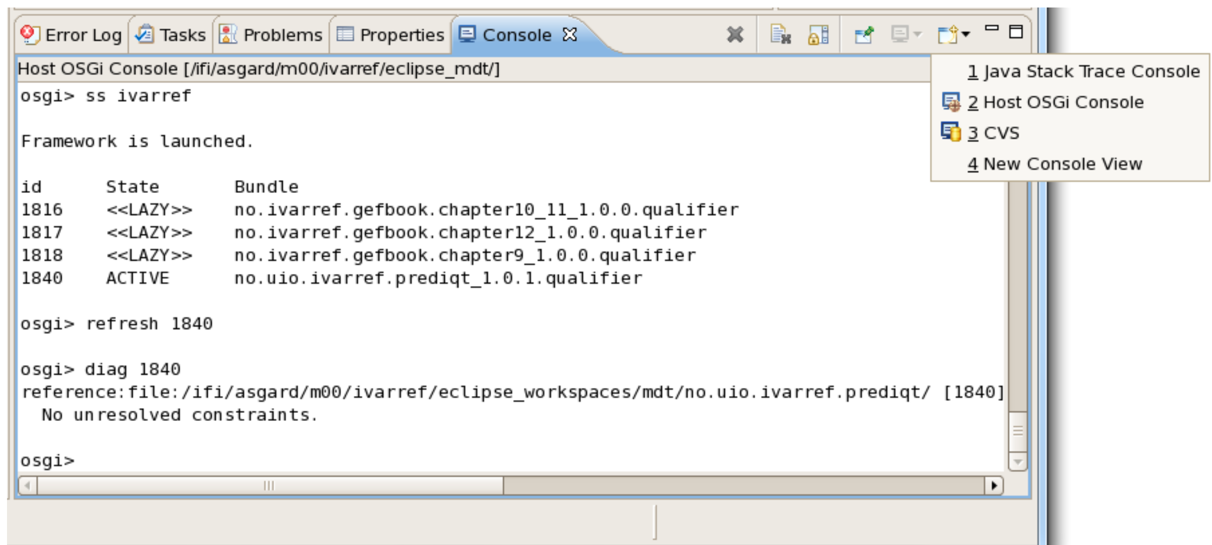


Figure A.1: OSGi console

Exactly which files are used for this refresh? Does the development instance package JARs on the fly? No. Doing “`diag bundleid`” in the testing instance will show that the bundle is located in exactly the same folder as in the development instance. Thus no JARs are packaged.

While this may seem like a trivial thing, it does speed up development speed considerably. The reader is referred to [49] for an introduction to OSGi.

A.2 Simple traceability: Git

A.2.1 Problem

For model driven software engineering, that is GMF in this case, another obstacle to productivity is not knowing where the models and their attributes ends up being in the generated code. Looking through the `xpt` files is rather tedious work, particularly because the model attributes have human readable names in the generated editors. Thus one needs to map the human readable names back to the actual attribute names if one seeks to understand the `xpt` files.

A.2.2 Solution

While also a rather trivial solution, one possibility is to use git. Initially one should first do the following in the folder of the generated code:

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

This commits the initial code to a local repository. Then the style of development cycle looks like the following:

1. Change some attribute in the models.
2. Re-generate the code.
3. Use "`git diff`" to see what was actually changed.

The author found this method quite useful when exploring the Spray samples and could easily discover that some of the attributes of the models were not (yet) used in the code generation. This method were also useful when experimenting with the Emfatic model. This technique has the advantage of working with any generation techniques, but obviously does not provide a reverse mapping from generated code to models.

A.3 Suggested prerequisite learning for GMF and Graphiti internals

A.3.1 Problem

Going straight at GMF development was not very simple. One typically ends up just browsing code and searching forums looking for a specific recipe. The same would likely apply to dealing with the Graphiti internals.

A.3.2 Solution

- If one is not familiar with the classic software patterns, one should read [17] or a similar book.

- Learn basic OSGi usage as stated above. Recommended book: [49].
- Walk through at least up to chapter 9 in [8].
- Walk through all of [37]. For the parts on GEF architecture, consult [7] as this provides a much simpler explanation of some of patterns present in GEF.
- Walk through some of [43]. Particularly chapter 3, 11 and 16 are important. Again one should consult [7] for a more clear explanation of some patterns.

After this one should have a good chance of understanding GMF and Graphiti internals.

Appendix B

Survey

B.1 Setup

The partcipant should be a reasonable competent person with respect to computer usage. The tasks should be done on the author's computer where the two tools are already installed, thus the participant should not need to do any downloading or installing. Each participant should use both editors to perform the tasks given. However, half of the participants should use the GMF-based editor first, and then the Graphiti-based editor. For the other half the exact opposite applies. By letting each participant use both tools, the background of the each participant will not favor any particular tool. If a participant was stuck at a certain task, the author intervened and aided the participant.

The questionnaire that the users should answer is presented in the following text.

B.2 Survey information as given to users

All of the rating should answered in a scale 1 to 6, where the numbers represents the following:

1. Very easy.
2. Easy.
3. Somewhat easy.
4. Somewhat difficult.
5. Difficult.
6. Very difficult.

1. **Rate your own skill in using Eclipse where 1 is highly skilled and 6 is completely unskilled.**
2. **Create a new diagram file.**

2.1. **Instructions for editor number one:**

2.1.1. Click “File” → “New” → “Other”.

2.1.2. Type “Prediqt” in the search field.

2.1.3. Choose “Prediqt Diagram”.

2.1.4. Click “Next” then “Finish”.

2.2. **Instructions for editor number two:**

2.2.1. Click “File” → “New” → “Other”.

2.2.2. Type “Graphiti” in the search field.

2.2.3. Choose “Graphiti Diagram”. Click “Next”.

2.2.4. Select “prediqt” as the diagram type.

2.2.5. Click “Next” then “Finish”.

2.2.6. Close the opened file.

2.2.7. Right-click the newly created file and select “Open with” → “Prediqt editor”.

2.3. How would you rate this task?

3. **Create a dependency tree in the diagram identical to the one in figure B.1.**

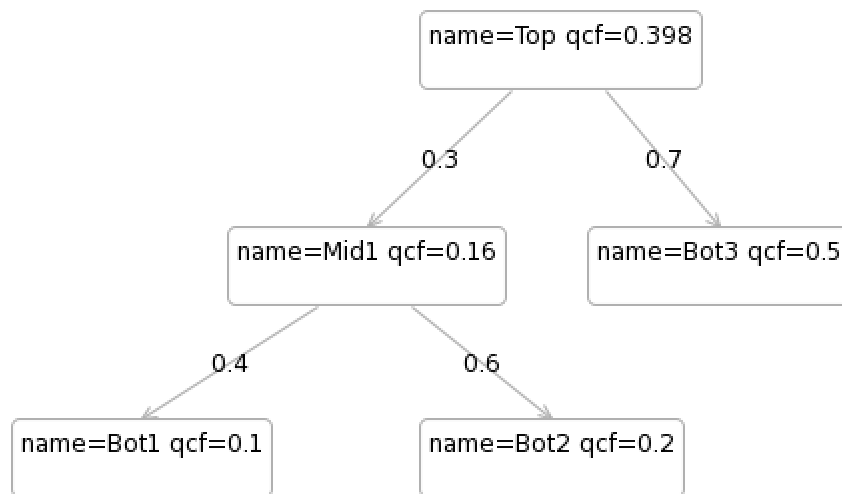


Figure B.1: Dependency tree that should be created.

3.1. **Instructions for both editors:**

3.1.1. Create the node and the arcs using the tools located to the right of the canvas.

3.1.2. Edit the name and qcf attribute through the property sheet located below of the canvas.

3.2. How would you rate this task?

4. **Validate the tree.**

   4.1. **Instructions for editor one:**

      4.1.1. Click the canvas, then select "Edit" → "Validate".

   4.2. **Instructions for editor two:**

      4.2.1. Validation should happend automatically.

   4.3. Assert that there were no errors detected. If this is not the case, please note the errors.

   4.4. Introduce an error in the tree, such as changing the impact of an arc.

   4.5. Re-validate and assert that the error is detected.

   4.6. Fix the error manually.

   4.7. How would you rate this task?

5. **Select the "Mid1" node, choose to use qcf override with the value of 0.5.**

   5.1. **Instructions for both editors:**

      5.1.1. Set the "`use qcf override`" to "`true`" in the property sheet.

      5.1.2. Set "`qcf override`" to "`0.5`".

   5.2. How would you rate this task?

6. **Overall.**

   6.1. How would you rate the degree to which the tool is easy to learn and use?

   6.2. Which editor would you prefer to use in the future?

   6.3. If you found something to be particularly good or bad, please describe it here.

Appendix C

Installation and source code of editors

C.1 Shared environment

Eclipse Indigo MDT is available for download at
http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr1

The editors are available for download at
http://code.google.com/a/eclipselabs.org/p/prediqt-editors/downloads/list

C.2 Installation of editors

1. Go to http://code.google.com/a/eclipselabs.org/p/prediqt-editors/downloads/listand download the *prediqt_editors.zip* file.

2. Open your installation of Eclipse Indigo MDT.

3. Go to the workbench.

4. Select "Help", then "Install New Software" from the menu.

5. Select "Add...", then "Archive" and locate *prediqt_editors.zip*. Press "OK".

6. Select the editor(s) you want to install from the Prediqt category, e.g. "PREDIQT GMF-based editor" and then click "Next".

7. Follow the instructions until the software is installed.

8. Choose to restart Eclipse when asked.

For instructions on how to use the editors, the reader is referred to the instructions in appendix B.2.

C.3 Source code of editors

The source code of the editors are available for download at
http://code.google.com/a/eclipselabs.org/p/prediqt-editors/downloads/list
Download *prediqt_sources.zip*. This archive can be imported into Eclipse using the following steps:

1. Right click the package explorer, select "Import...".
2. Select "Existing Projects into Workspace". Click "Next".
3. Locate *prediqt_sources.zip*. Select the desired projects.
4. Click "Finish".

Appendix D

PREDIQT case study

To understand the PREDIQT-method from a more practical point of view, a case study was held. ICUSystem, a system which is developed in the course INF5150 at University of Oslo every autumn, was used as the target system. This system was reasonably small, reasonably well understood by the various parties involved in the case study and thus deemed fit as a target system. The year the author took the course, the system was specified at only a design level, i.e. there was no actual running code.

Originally the ICUSystem was used so that a user may retrieve a nearest "hot position" such as a coffee shop or bus stop. This was done without a central server.

ICUSystem was then, at a design level, changed to support namely that, a central server. With these changes, users could also ask for permission to retrieve each other's hot position, e.g. for a meeting point. The original and new system were specified using UML models such as sequence diagrams, composite structures and use cases.

The following sections describe the PREDIQT method and process as applied to the ICUSystem, with changes made as outlined above. First though, the overall steps of PREDIQT are repeated as an easy reference.

1. Target modeling.
    - 1.1. Characterize the target and the objectives.
    - 1.2. Create quality models.
    - 1.3. Map design models.
    - 1.4. Create dependency views.
2. Verification of prediction models.
    - 2.1. Evaluation of models.
    - 2.2. Fitting of prediction models.
    - 2.3. Approval of the final prediction models.
3. Application of prediction models.
    - 3.1. Specify a change.

3.2. Apply the change on prediction models.

3.3. Within the scope of models?

3.4. Quality prediction.

4. A new change? If Yes, go to step 3.

D.1 Target modeling

The first overall step in the PREDIQT method, is target modeling, where one characterizes the system. As in the industrial PREDIQT-case [32], while the ICUSystem is obviously much simpler, there was a lack of precise design models for the system. These were created in collaboration with fellow student M. Køller, and are included in the appendix, section D.6.1.

D.1.1 Characterize the target and the objectives

The system boundaries for the ICUSystem was defined as follows:

- The stakeholder was defined to be the end user.
- The functionality covered was set to be the whole system, with exception of the basic infrastructure.
- The communication between buddies is realized through SMS messages.
  - Should be safe for us to assume this will not fail, i.e. we assume the infrastructure is reliable.
  - The analysis will therefore only focus on the client side.
- External functionality/applications on the smartphone such as GPS and Maps are only briefly described.
- The system runs on Android smartphones.
- The maximum number of users is thought to be 50.

Furthermore the system context, that is who is using the system as well as the operational environment, was defined as the following:

- The users of the system is thought to primarily be used by the average IT-person, in other words a typical IT-student.
- The system is thought to be running on the Android platform, which is based on Java.
- The system is thought to be used on a weekly or daily basis.
- The system is used by a user by opening the ICU-application on his/her smartphone. The application may also be opened by the system on the reception of an SMS.

- It is used when a friend wants to see or meet another friend, e.g. for taking a coffee break, establish a common meeting point or similar activities.

The system life time was set to be five years. The extent, i.e. nature and rate, of design changes expected, was set to the following:

- Larger changes are expected.
- Possible changes include:
  - Moving the "Archive" component to a central location.
  - Letting users add and share new Archive entries.
  - Supporting buddy groups.
  - Integrating with other services.
- The changes are expected to occur on a six month basis.

While these expectations may not be the most realistic for this particular system, in a sense they are needed to mandate the usage of PREDIQT. If the changes were expected to be small and frequent, it would not make sense to apply the PREDIQT method. Regardless though, the goal here is to learn the PREDIQT method, and thus these descriptions, realistic or not, should suffice for learning purposes.

D.1.2 Create quality models

The next step in the PREDIQT process is to create the quality models. This was done primarily based on [22], which details software quality, and [21], which details information security. The following definitions, i.e. where quotes are used, are taken from or based on these standards.

During a meeting with supervisor K. Stølen and co-supervisor A. Omerovic, it was decided that the focus should be on security, particularly with respect to confidentiality and integrity, both of which are explained in the the aforementioned standards.

For this case study, security will be the quality attribute or quality characteristic. These words are used interchangeably in the referred papers. The quality characteristics will always be at the root node in the dependency views. In a larger case study, one would normally have other quality characteristics as well.

As given in [21], security is characterized as the preservation of confidentiality, integrity and availability. As described earlier, it was decided to focus on the two former. These sub-characteristics are defined as:

- Confidentiality: "Ensuring that information is accessible only to those authorized to have access."

  The interpretation of this with respect to the ICUSystem is straight forward and will not be explained further.

- Integrity: "Safeguarding the accuracy and completeness of information and processing methods."

  For the ICUSystem, *accuracy* is the degree to which the GPS positions, the hot position's calculation methods and data stored by the system are accurate at the sufficient detail level.

  Completeness, for the ICUSystem, is the degree to which the data received and stored by the system, as well as the methods operating on data, is complete.

PREDIQT requires the quality characteristics to be measurable, otherwise none of the quality characteristic fulfillment (QCF) would be meaningful. The following measurement methods are given:

- Confidentiality: Can be calculated as 1 - (number of unauthorized accesses / total number of accesses).
- Integrity
  - Accuracy: Can be measured as 1 - (number of inaccurate items found in review / total number of items requiring accuracy as specified in the system requirements).
  - Completeness: Can be measured as 1 - (the number of incomplete items found in review / total number of items reviewed which require, implicitly or explicitly, completeness).

    In both cases "items" is defined to be data and methods.

This all leads up to the security rating, i.e. what precisely is meant by "security":

- Security rating: $0.75 * x + 0.25 * (0.5 * y + 0.5 * z)$

Where:

- x = Confidentiality rating.
- y = Integrity's accuracy rating.
- z = Integrity's completeness rating.

The rationale for this is that confidentiality is more important than integrity (roughly three times) for the system. With respect to integrity, accuracy and completeness are equally important. This security rating allows us to interpret the QCF value of the top level node in the dependency view.

D.1.3 Map design models

This step, where a subset of the design models may be chosen as well as establishing mappings between different levels of design models, is more applicable in a larger setting, and thus this step was not taken further in the ICUSystem case.

D.1.4 Create dependency views

In this step, the dependency views are created. One of the requirements for a dependency view, is to have a quality characteristic as the top level node.

The dependency view is created by deducing the sub-characteristics and leaf nodes. This is done partially based on the design and quality models. However, it is also a creative process where domain experts are involved[1]. In other words, the dependency views are not a direct mapping or modification of quality models, design models, indicators or sub-characteristics in any "same input, same output"-algorithmic sense. The overall goal of this process is to cover the quality characteristic completely. For that purpose, in order to ensure model completeness, an "other" node is also typically added.

Furthermore, the indicators serves as helpers or guidelines for estimating the QCF of leaf nodes. As the leaf nodes' QCF will be propagated to the top level quality characteristic, they will ultimately determine, together with the estimated impact values, the QCF of the top level quality characteristic. This top level value will be interpreted based on the rating, and therefore also the leaf nodes' estimation must be based on the rating as well, otherwise the numbers would not make particularly much sense.

In the end the dependency view were created based on the available dependency views from the PREDIQT industrial case, the design and quality models developed earlier, as well as the author acting as a domain expert.

D.2 Verification of prediction models

This step, and it's sub-steps, involves a verification of the prediction models where statistical methods are applied to verify whether the prediction models are suitable for further use or not. As this case study is reasonably small and for learning purposes only, it was not deemed necessary. This however does not seriously weaken the overall understanding as it does not heavily involve new models, methods or similar that are of particular relevance to the PREDIQT method itself.

D.3 Application of prediction models

Application of prediction models is the last major step of the PREDIQT method where the actual changes are applied to the prediction models.

D.3.1 Specify a change

The following changes were specified:

- Moving the database to a central location.

[1] Personal communication with co-supervisor A. Omerovic.

- The addition of the register functionality.
- The getHotpos message now takes a username as a parameter.

D.3.2 Apply the change on prediction models

The changes had the following effects on the design models:

- The composite structure changed. The database component was added.
- Sequence diagrams were added specifying new services.

For more details, see the appendix, section D.6.1.

The affected DV parameters were thought to be:

1. Redundancy.
2. External backup.
3. Data encryption.
4. Illegal/incorrect events.
5. User events.
6. Firewall.

D.3.3 Within the scope of models?

The applied changes were thought to be within the scope of the models, i.e. the dependency view was able to represent the quality of the systems without further changes.

D.3.4 Quality prediction

The quality prediction itself was done within Excel as this is what the current PREDIQT method uses. The net result was a worsening of the security characteristic after applying the changes. More specifically the degree of data encryption and logging of illegal/incorrect events was thought to be worsened. The quality of the redundancy and external backup nodes was thought to be improved. The full details are given in the appendix, section D.6.1.

D.4 A new change? If Yes, go to step 3

As this was a reasonably small system, all the changes were simulated at once, thus there was no need for this step.

D.5 Other experiences

The Papyrus tool, an Eclipse based UML editor, was used for many of the design models. While mainly achieving what is was supposed to do, it was not the most pleasant user experience. It was hard to create some UML models properly, and the tool seemed at times to work in counter intuitive ways. This may of course change in later versions, or perhaps the setup used was incorrect. TopCased, another Eclipse-based tool supporting UML models, was better received.

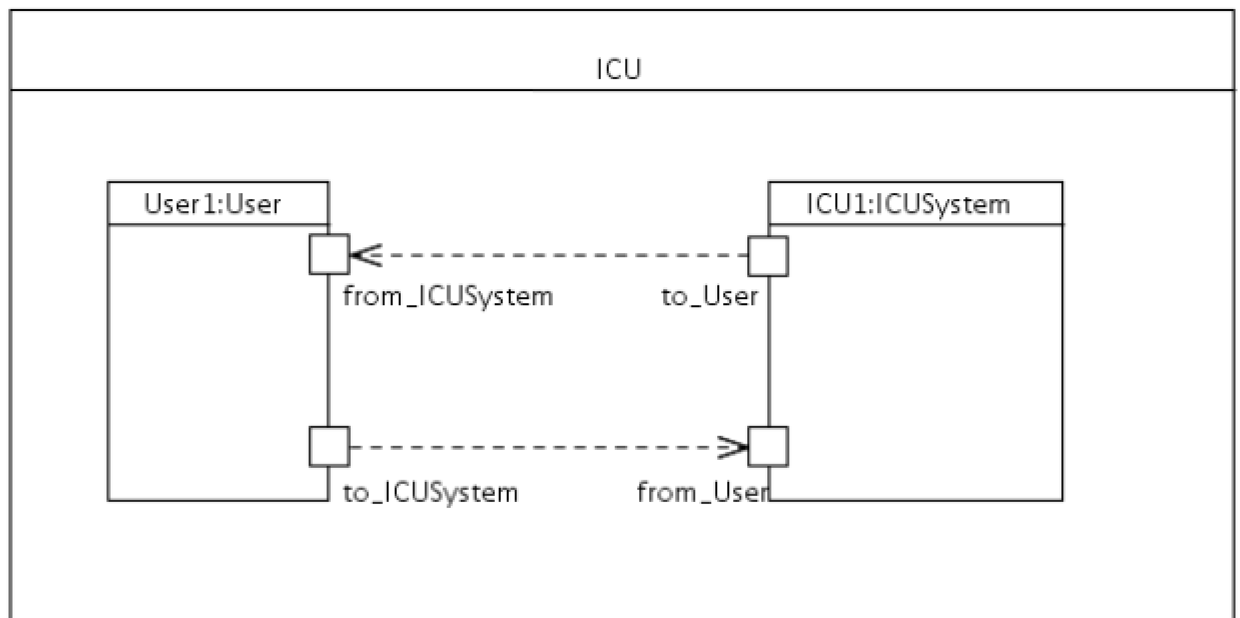
Either way though, it could have been helpful to have a textual syntax for precise specification of e.g. sequence diagrams. Depending on the syntax, it may or may not impose some restrictions on what is specifiable. If the end user does not find the graphical editor to be sufficient or lacking in some ways, a textual syntax may be an alternative if it is reasonably intuitive. A textual syntax may also be useful for specifying exactly what was done or used in a precise, yet human readable way. This may be practical for textual exchanges over email or for using as a listing in technical reports. This should be considered for PREDIQT as well and will be explored further in later sections.

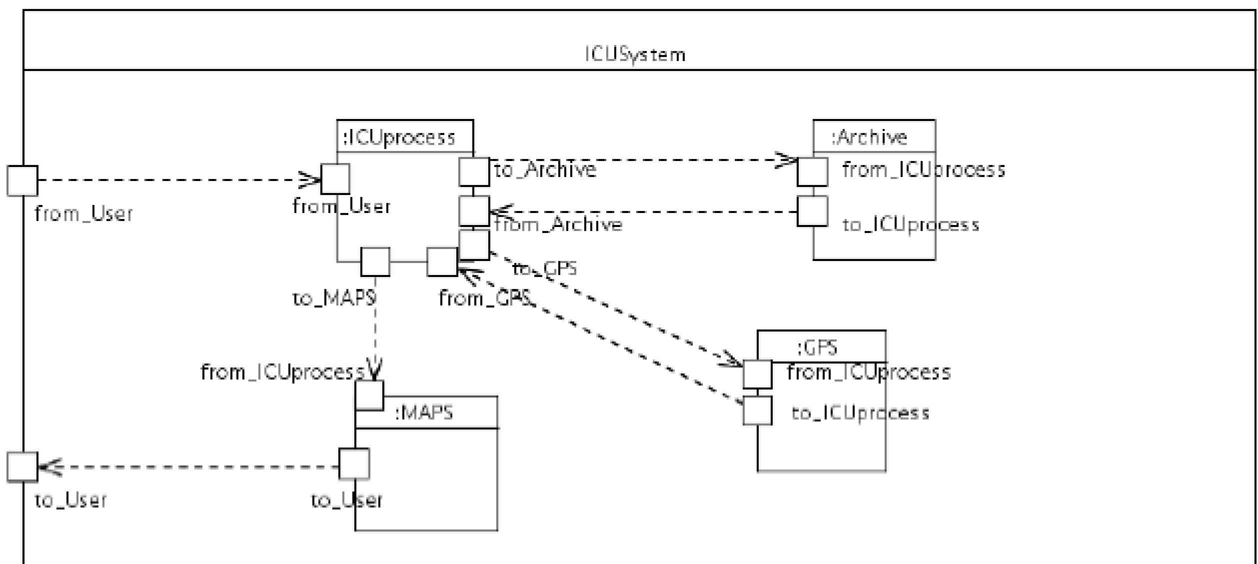
D.5.1 Conclusion

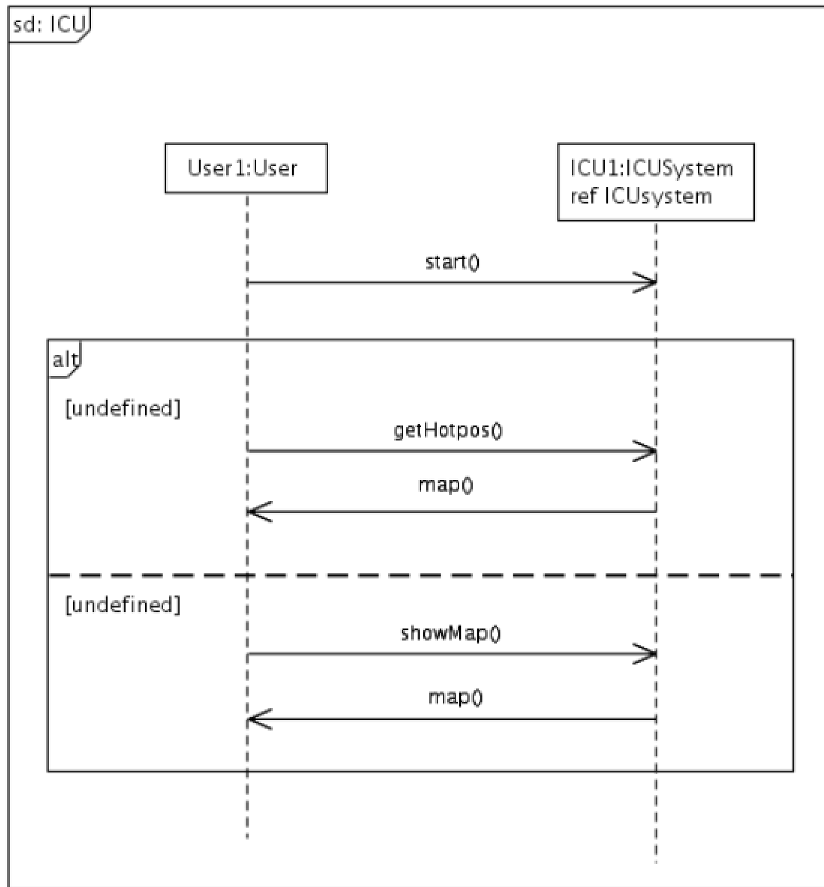
PREDIQT proved to have a reasonable amount of subtleties that the author had not understood well from only reading the technical reports. The overall process of the PREDIQT method is also more well understood after having performed this case study. As a background for tool creation, this case study has been useful and should particularly serve well with respect to defining requirements for the new tool.

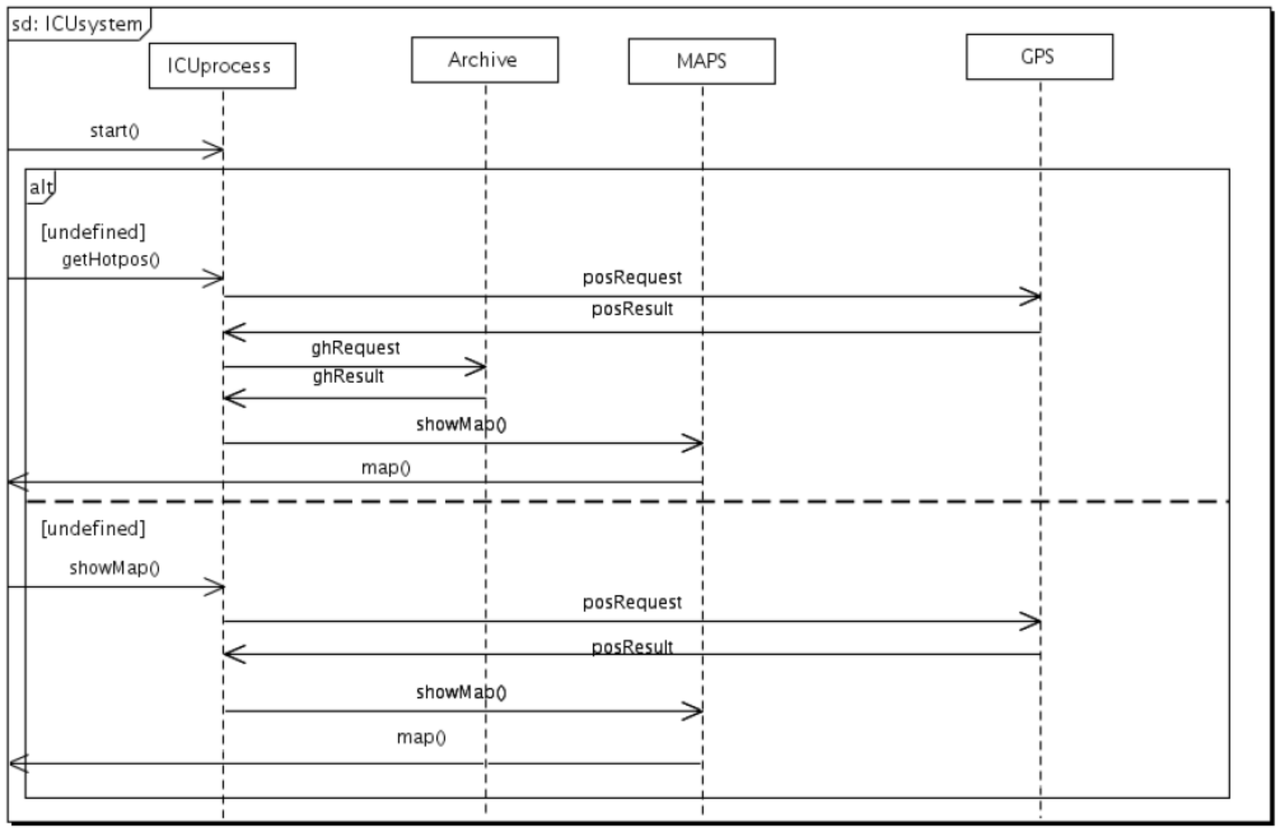
D.6 Quality models

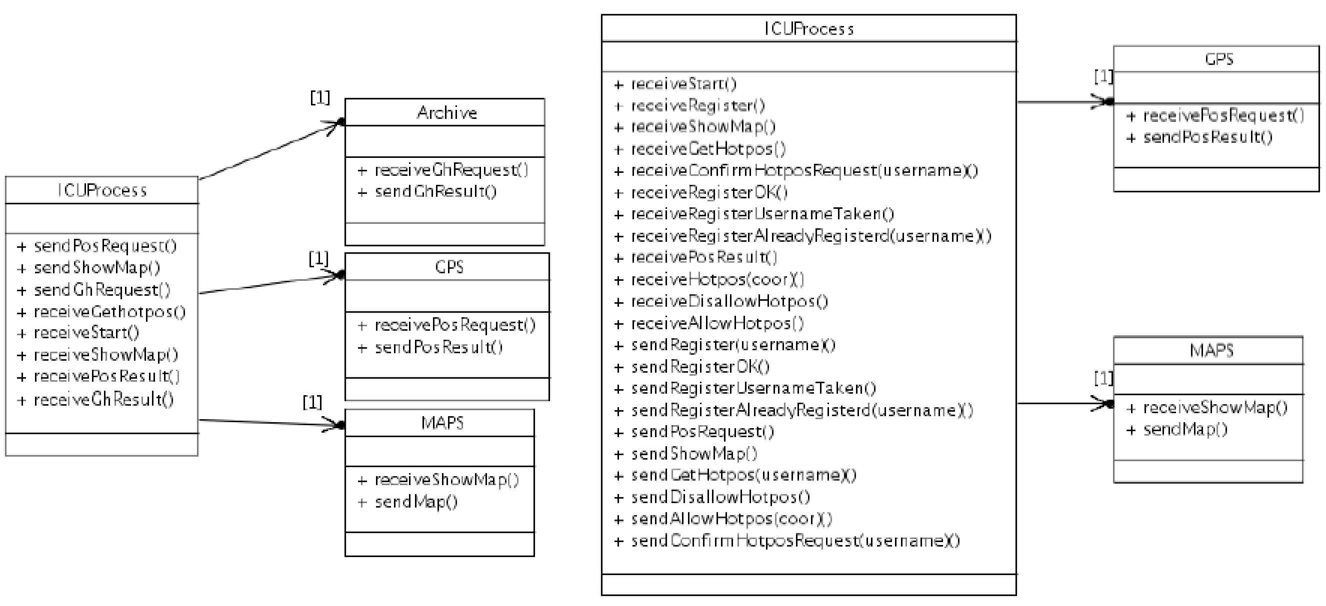
D.6.1 Design models - Original system



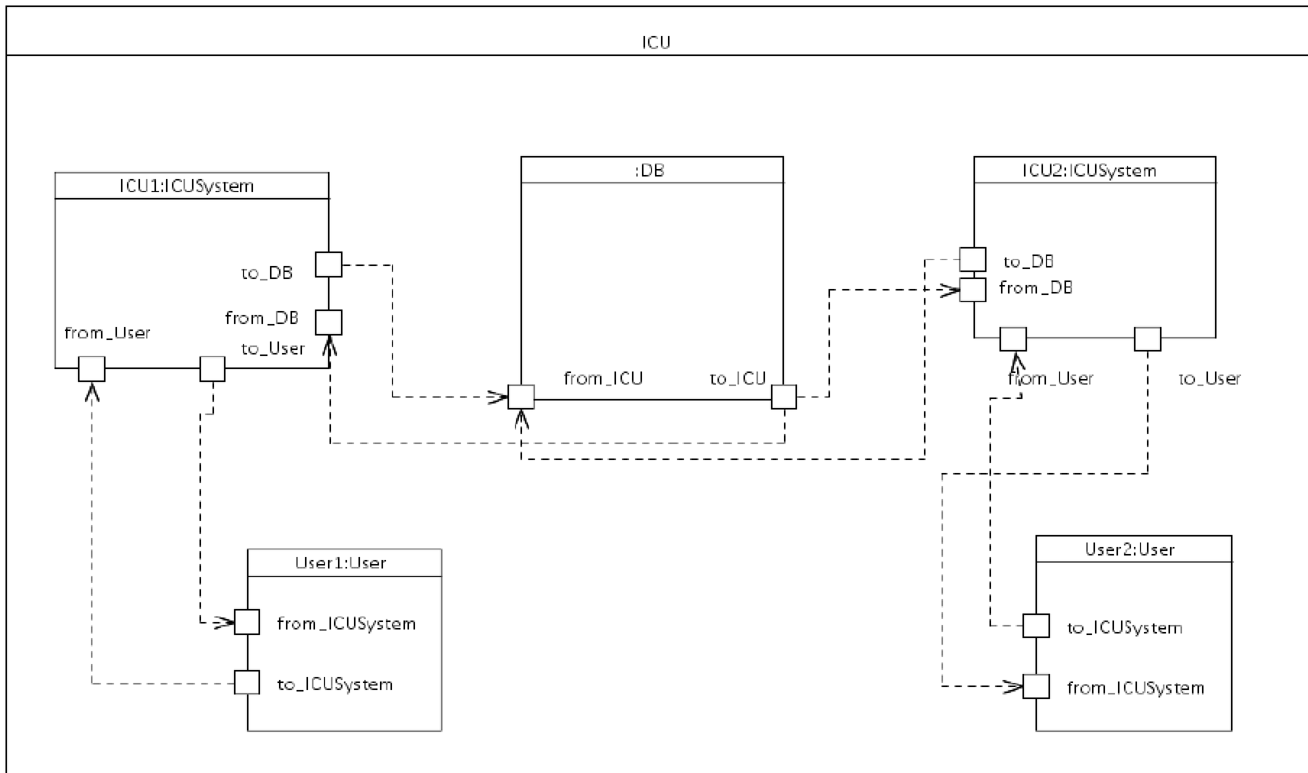


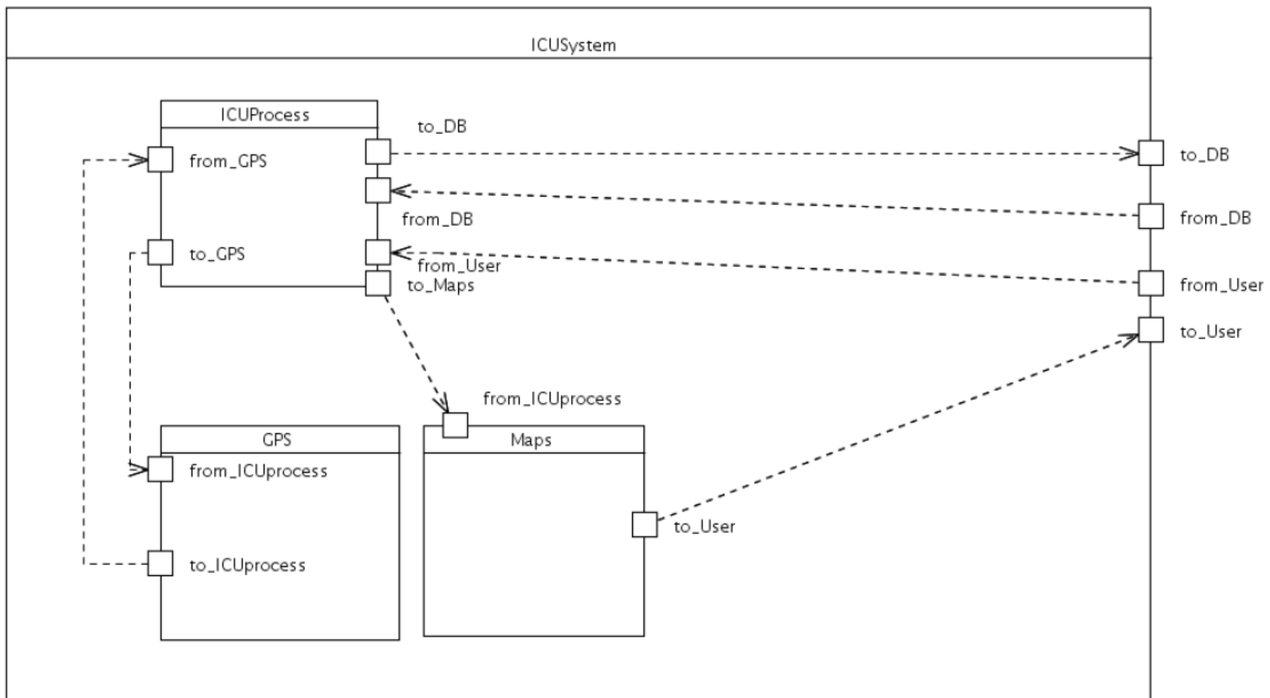

sd: ICU
User1:User
ICU1:ICUSystem
ref ICUsystem
start()
alt
[undefined]
getHotpos()
map()
[undefined]
showMap()
map()

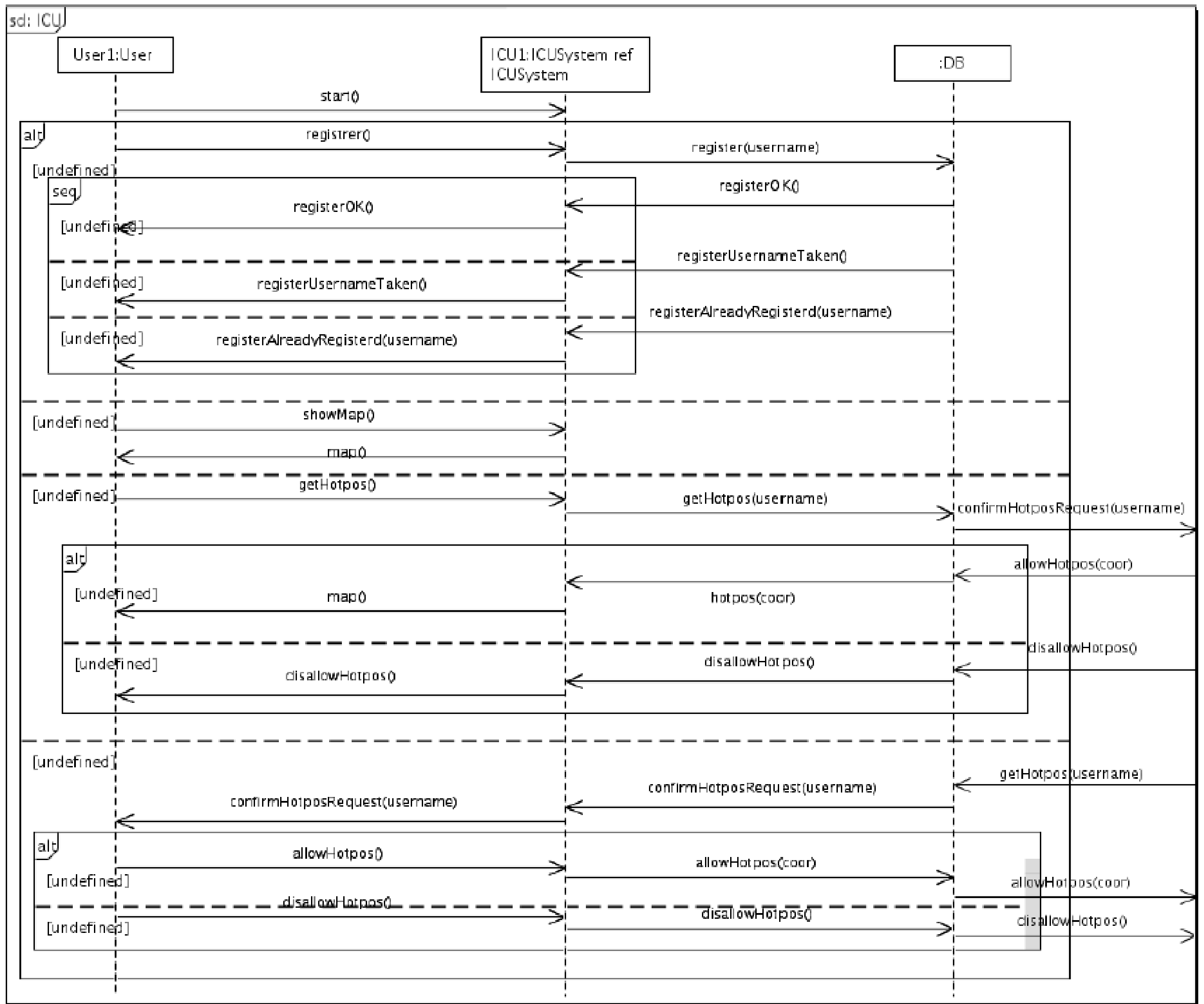

sd: ICUsystem
ICUprocess
Archive
MAPS
GPS
start()
alt
[undefined]
getHotpos()
posRequest
posResult
ghRequest
ghResult
showMap()
map()
[undefined]
showMap()
posRequest
posResult
showMap()
map()

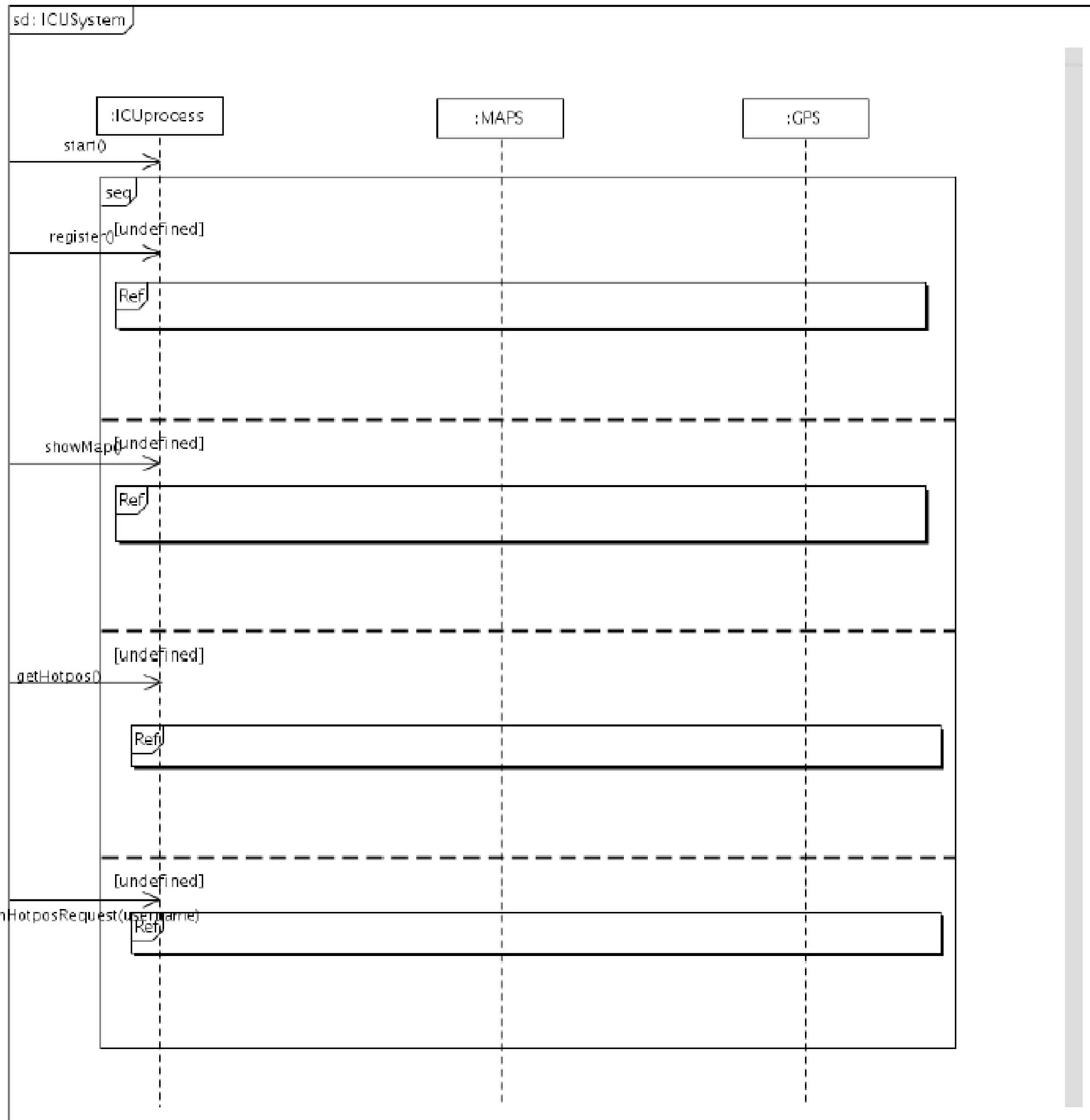

ICUProcess
+ sendPosRequest()
+ sendShowMap()
+ sendGhRequest()
+ receiveGethotpos()
+ receiveStart()
+ receiveShowMap()
+ receivePosResult()
+ receiveGhResult()
[1]
Archive
+ receiveGhRequest()
+ sendGhResult()
[1]
GPS
+ receivePosRequest()
+ sendPosResult()
[1]
MAPS
+ receiveShowMap()
+ sendMap()
ICUProcess
+ receiveStart()
+ receiveRegister()
+ receiveShowMap()
+ receiveGetHotpos()
+ receiveConfirmHotposRequest(username)()
+ receiveRegisterOK()
+ receiveRegisterUsernameTaken()
+ receiveRegisterAlreadyRegisterd(username)()
+ receivePosResult()
+ receiveHotpos(coor)()
+ receiveDisallowHotpos()
+ receiveAllowHotpos()
+ sendRegister(username)()
+ sendRegisterOK()
+ sendRegisterUsernameTaken()
+ sendRegisterAlreadyRegisterd(username)()
+ sendPosRequest()
+ sendShowMap()
+ sendGetHotpos(username)()
+ sendDisallowHotpos()
+ sendAllowHotpos(coor)()
+ sendConfirmHotposRequest(username)()
[1]
GPS
+ receivePosRequest()
+ sendPosResult()
[1]
MAPS
+ receiveShowMap()
+ sendMap()

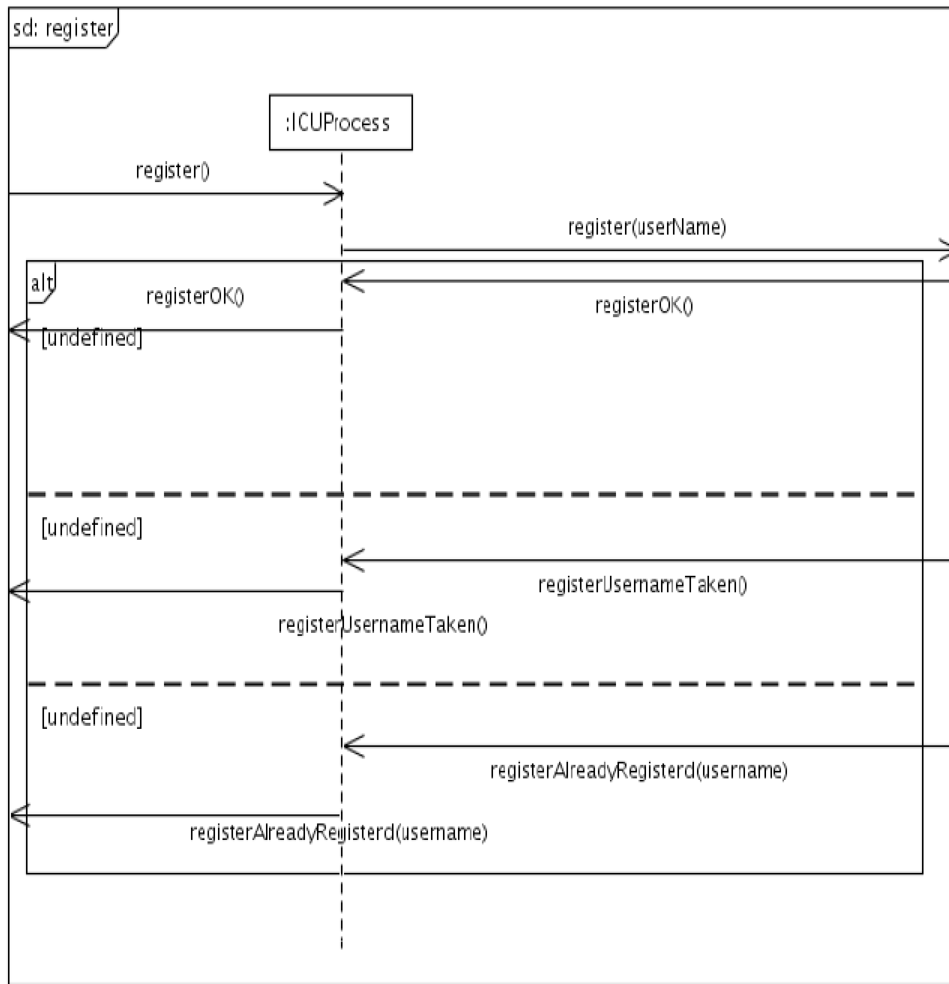
D.6.2 Design models - New system

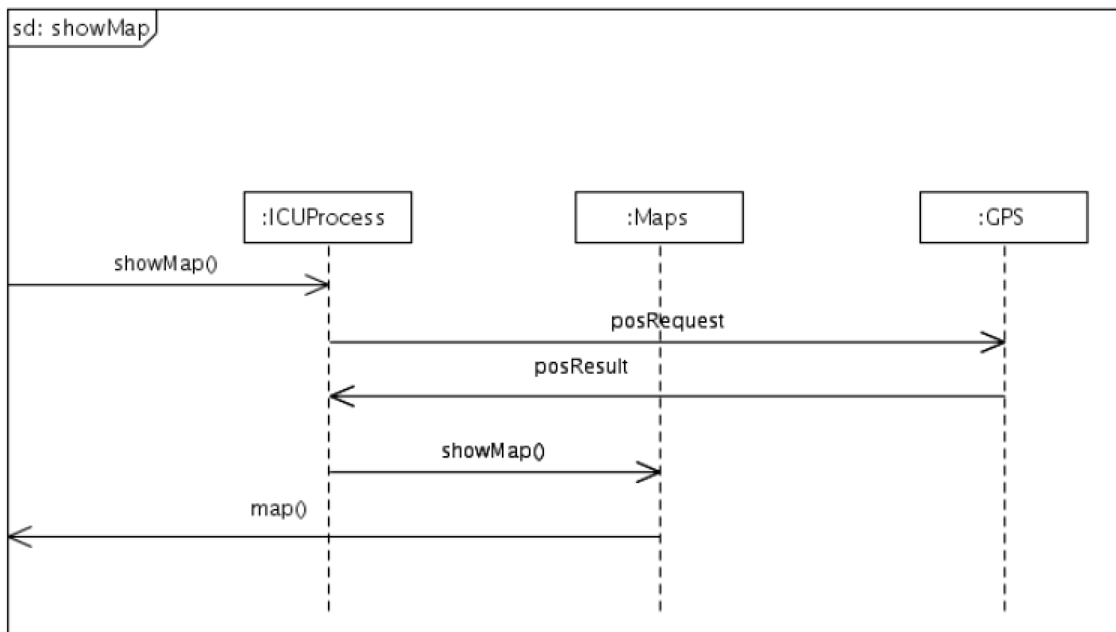


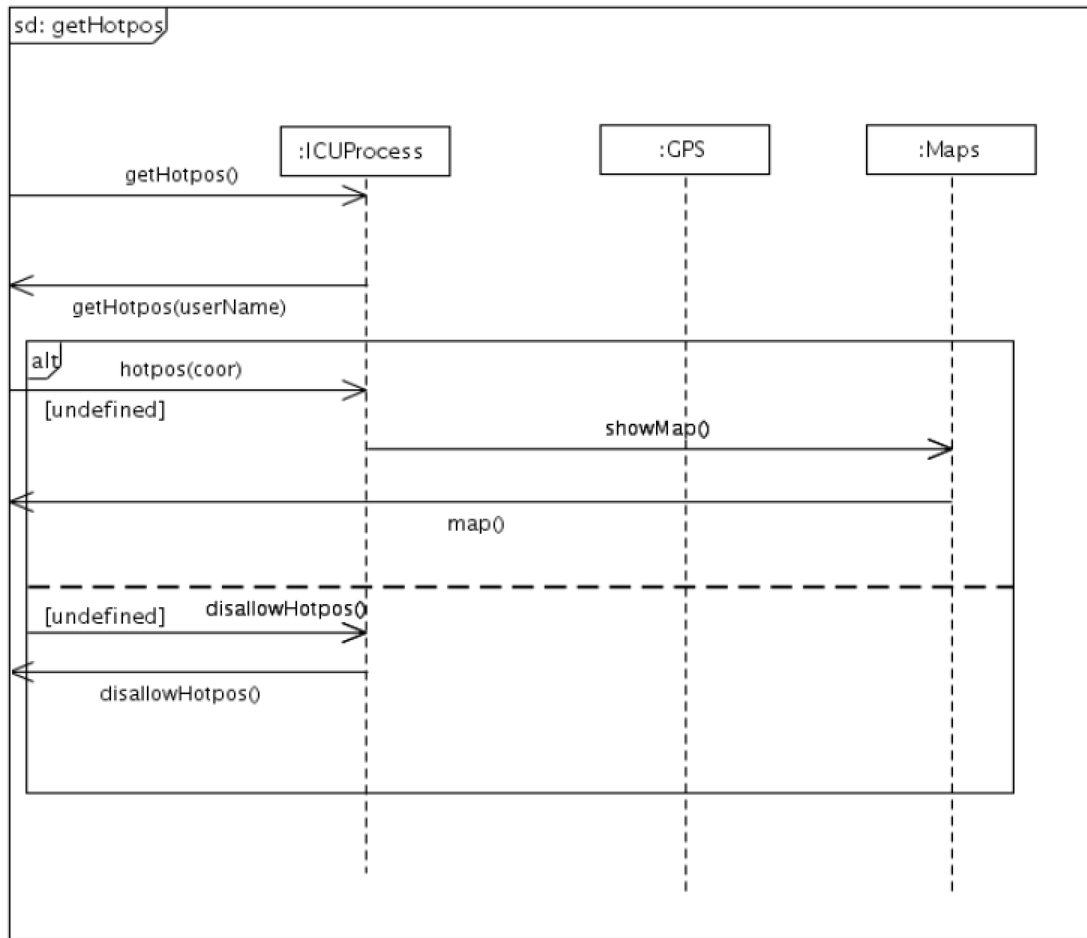


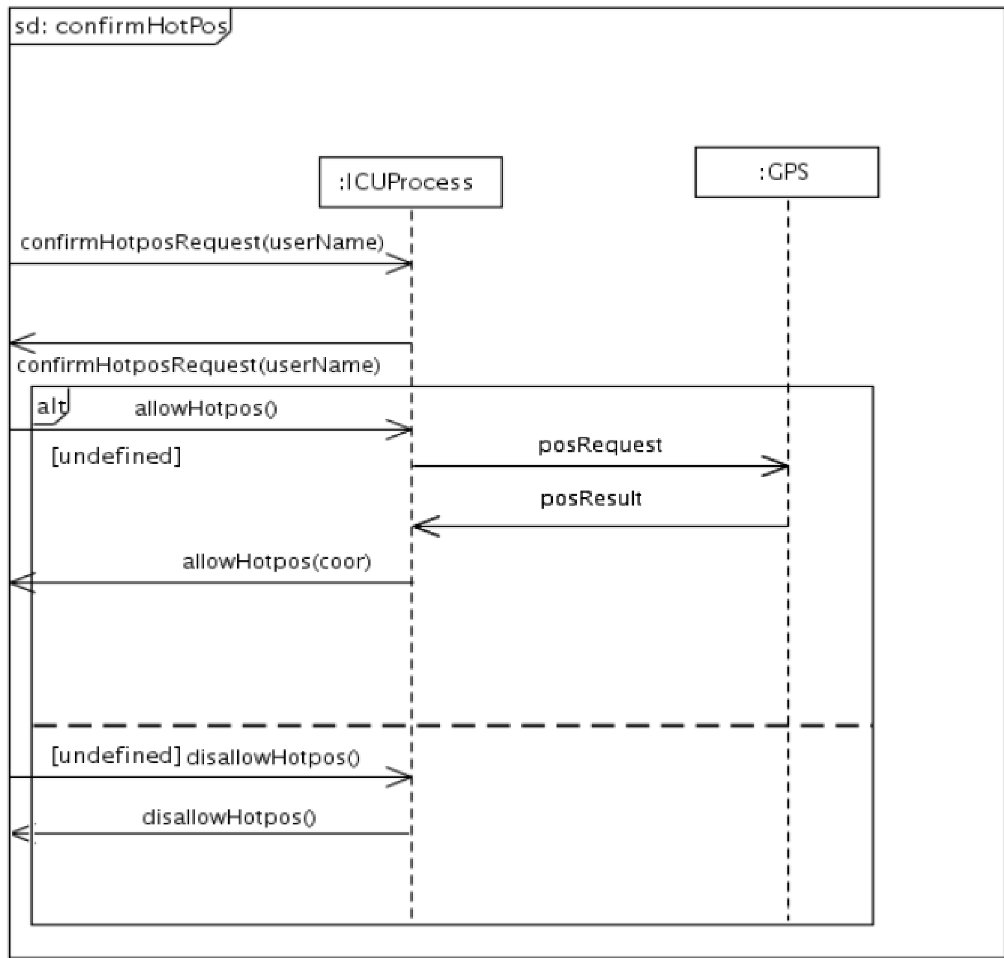

sd: ICU
User1:User
ICU1:ICUSystem ref ICUSystem
:DB
start()
alt
registrer()
register(username)
[undefined]
seq
registerOK()
registerOK()
[undefined]
registerUsernameTaken()
[undefined]
registerUsernameTaken()
registerAlreadyRegisterd(username)
[undefined]
registerAlreadyRegisterd(username)
[undefined]
showMap()
map()
[undefined]
getHotpos()
getHotpos(username)
confirmHotposRequest(username)
alt
allowHotpos(coor)
[undefined]
map()
hotpos(coor)
disallowHotpos()
[undefined]
disallowHotpos()
disallowHotpos()
[undefined]
getHotpos(username)
confirmHotposRequest(username)
confirmHotposRequest(username)
alt
allowHotpos()
allowHotpos(coor)
[undefined]
allowHotpos(coor)
disallowHotpos()
disallowHotpos()
[undefined]
disallowHotpos()

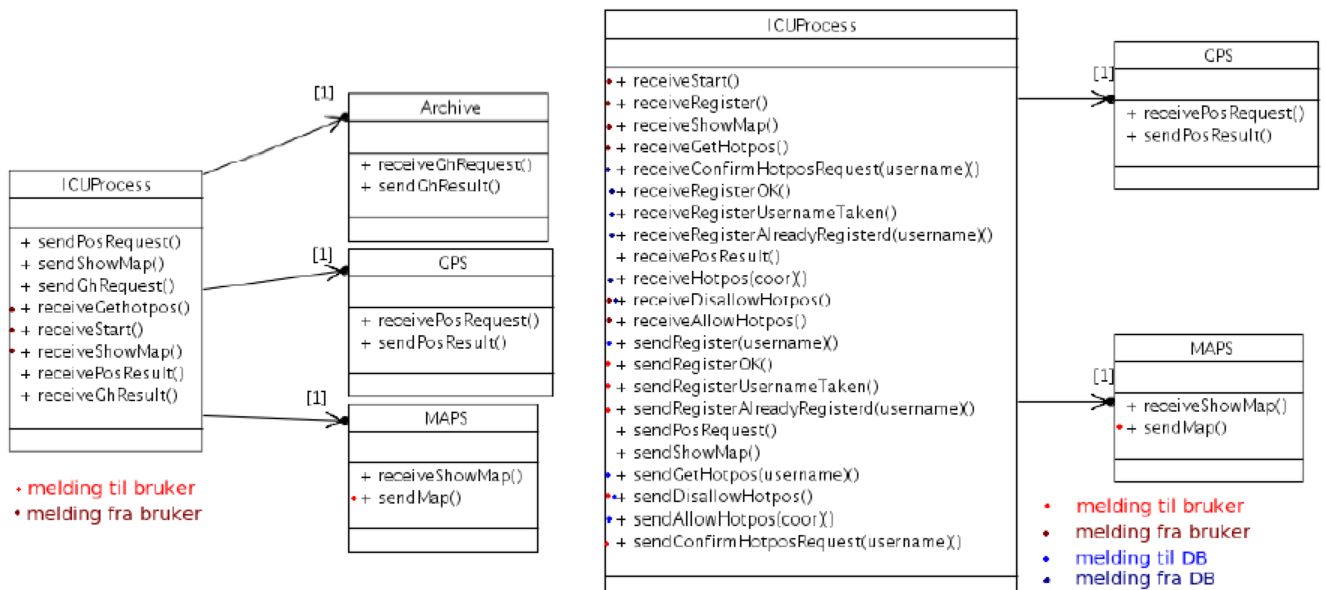

sd : ICUSystem
:ICUprocess
: MAPS
:GPS
start()
seq
[undefined]
register()
Ref
[undefined]
showMap()
Ref
[undefined]
getHotpos()
Ref
[undefined]
nHotposRequest(username)
Ref


sd: register
:ICUProcess
register()
register(userName)
alt
registerOK()
registerOK()
[undefined]
[undefined]
registerUsernameTaken()
registerUsernameTaken()
[undefined]
registerAlreadyRegisterd(username)
registerAlreadyRegisterd(username)


sd: showMap
:ICUProcess
:Maps
:GPS
showMap()
posRequest
posResult
showMap()
map()


sd: getHotpos
:ICUProcess
:GPS
:Maps
getHotpos()
getHotpos(userName)
alt
hotpos(coor)
[undefined]
showMap()
map()
[undefined]
disallowHotpos()
disallowHotpos()


sd: confirmHotPos
:ICUProcess
:GPS
confirmHotposRequest(userName)
confirmHotposRequest(userName)
alt
allowHotpos()
[undefined]
posRequest
posResult
allowHotpos(coor)
[undefined] disallowHotpos()
disallowHotpos()


ICUProcess
+ sendPosRequest()
+ sendShowMap()
+ sendGhRequest()
+ receiveGethotpos()
+ receiveStart()
+ receiveShowMap()
+ receivePosResult()
+ receiveGhResult()
[1]
Archive
+ receiveGhRequest()
+ sendGhResult()
[1]
GPS
+ receivePosRequest()
+ sendPosResult()
[1]
MAPS
+ receiveShowMap()
+ sendMap()
melding til bruker
melding fra bruker
ICUProcess
+ receiveStart()
+ receiveRegister()
+ receiveShowMap()
+ receiveGetHotpos()
+ receiveConfirmHotposRequest(username)()
+ receiveRegisterOK()
+ receiveRegisterUsernameTaken()
+ receiveRegisterAlreadyRegisterd(username)()
+ receivePosResult()
+ receiveHotpos(coor)()
+ receiveDisallowHotpos()
+ receiveAllowHotpos()
+ sendRegister(username)()
+ sendRegisterOK()
+ sendRegisterUsernameTaken()
+ sendRegisterAlreadyRegisterd(username)()
+ sendPosRequest()
+ sendShowMap()
+ sendGetHotpos(username)()
+ sendDisallowHotpos()
+ sendAllowHotpos(coor)()
+ sendConfirmHotposRequest(username)()
[1]
GPS
+ receivePosRequest()
+ sendPosResult()
[1]
MAPS
+ receiveShowMap()
+ sendMap()
melding til bruker
melding fra bruker
melding til DB
melding fra DB

D.6.3 Dependency views