Comparison of different Model Driven Development approaches
A mobile Meal Ordering System for the healthcare sector

Master thesis

Gøran Klepp Olsen

27th October 2005
Comparison of different Model Driven Development approaches

A Mobile Meal Ordering System for the healthcare sector

By

Gøran Klepp Olsen

Master thesis in Informatics

University of Oslo
Preface
This thesis has been written as part of a master degree at the University of Oslo. The work started in September 2004 and lasted to the end of October 2005. The thesis was written at the Research group for object-orientation, modeling, and languages.

This thesis is about different approaches to model driven development. Much of the IT industries history can be summarized as a quest for faster and better methods for system development. This thesis has taken a closer look at some of the proposed methods for the future. The first approach presented is the use of the Unified Modeling Language version 2.0 (UML2.0). UML is a standardized language used by the majority of system developers at present time. An example system is modeled in UML2.0 and implemented in Java manually. Then an automatic code compiler is tested for the generation of code from the UML2.0 models into a running implementation in Java. For this the extension mechanism profile are used in addition.

The second approach makes use of a Domain Specific language (DSL). A DSL is a language that is made for the target domain only. It is composed out of domain concepts and from these it is possible to generate code.

The example system is a mobile ordering system for the healthcare sector. The idea is that patients in a hospital are offered a system where they can order and specify their daily food servings. This is both beneficial for the hospital and the patients. The mobility is handled by the Parlay API which enables communication over different heterogenous networks (IP, Mobile phone).

An example DSL is made that is capable of creating different system family members of the already introduced example system. Some different DSLs from the industry are also presented.

Then the different approaches are discussed. What are the benefits and drawbacks of the different approaches are important questions that I have tried to answered.

I would like to thank my supervisor Øystein Haugen for his contributions to this master thesis. This includes providing valuable and time consuming feedback, motivation and reading the text.

Oslo, 1.November 2005
Goran Klepp Olsen
# Table of Content

## PREFACE ........................................................................................................................ ......................3

## TABLE OF CONTENT .........................................................................................................................5

## CHAPTER 1 .....................................................................................................................................8

### THESIS BACKGROUND AND PROBLEM DEFINITION ........................................................................8

- **INTRODUCTION** .................................................................................................................................8
- **WHAT IS EXPLORED IN THIS THESIS** .................................................................................................9
- **WHAT ARE THE GOALS** .......................................................................................................................10
- **STRUCTURE OF THESIS** .....................................................................................................................10

## CHAPTER 2 .....................................................................................................................................12

### UNIFIED MODELING LANGUAGE (UML) 2.0 ..................................................................................12

- **INTRODUCTION** .................................................................................................................................12
- **IMPROVEMENTS IN THE NEW VERSION** .............................................................................................13
  - **Nested Classifiers** ..............................................................................................................................13
  - **Composite Structure Diagram** ........................................................................................................13
  - **Sequence Diagram** ............................................................................................................................15
  - **State Machine Diagram** ....................................................................................................................17
- **UML PROFILES** ..................................................................................................................................19
  - **Stereotype** .........................................................................................................................................19
  - **Tagged value** ......................................................................................................................................19
  - **Constraints** .......................................................................................................................................19
  - **Existing profiles** ...............................................................................................................................19
- **META OBJECT FACILITY MOF** ..........................................................................................................20
- **MDA MODEL DRIVEN ARCHITECTURE** .............................................................................................21
  - **Transformation languages** ................................................................................................................22
- **UML 2.0 TOOLS** ..................................................................................................................................22
  - **Rationale Software Architect (RSA)** .................................................................................................23
  - **Visio** ................................................................................................................................................23
  - **JavaFrameTransformation plug-in** ....................................................................................................23

## CHAPTER 3 .....................................................................................................................................24

### DOMAIN SPECIFIC LANGUAGES (DSL) .........................................................................................24

- **INTRODUCTION TO DOMAIN SPECIFIC LANGUAGES** ......................................................................24
- **TOOLS** ...............................................................................................................................................25
- **META CASE** .......................................................................................................................................25
  - **The MetaCase meta-language** ...........................................................................................................26
  - **Method Workbench** ............................................................................................................................27
  - **MetaEdit+** .........................................................................................................................................31
  - **The MetaCase Report Definition Language (RDL)** ............................................................................31
- **MICROSOFT VISUAL STUDIO** ............................................................................................................32
  - **The Software Factories initiative** .......................................................................................................32
- **OTHER** ...............................................................................................................................................33

## CHAPTER 4 .....................................................................................................................................34

### THE PATS-LAB AND ITS TECHNOLOGIES .......................................................................................34

### TECHNOLOGY ..................................................................................................................................34

- **THE PATS-LABORATORY** ....................................................................................................................34
Chapter 1

Thesis background and problem definition

*The history of software engineering is, in effect, the history of abstraction. As complexity rises, we respond by raising the level of abstraction in our programming languages and our methods.*

-Grady Booch

**Introduction**

While in the last 20 years the improvements in CPU and memory power has been enormous and has grown by a factor of 500 – 2000 according to [1], the ability to create applications easier and faster has not increased in the same pace.

There have been lots of initiatives to increase the quality and productivity over the last decade, some more successful than others. The object oriented paradigm, modelling and different architectures like thick and thin client can be mentioned as examples. But none of the improvements has radically enhanced either quality or productivity. Some actually claims that the most radical improvement in the past 40 years was the introduction of COBOL as a major innovation over the assembly-languages[1]. COBOL made it possible for developers to raise the abstraction, from the tight coupling to the CPU-instruction-set up to a more suited language for solving business problems. The new language accomplished to be kind of a standardized one and this was a crucial success factor.

4 generation languages and modelling tools made an attempt in the late 80’s and the 90’s to take the next step in developer productivity and software quality. Common to them was that they both used high-level languages that were better suited for business problems than their counterparts. They have been very successful but there were some problems. 4GL did not provide a standard and sufficient support for distributed environments. The modelling tools did not have a standard either and did not really decrease the distance between model and code.

Some of this has now changed. The Unified Modeling Language became an OMG standard in 1997. It was a combination of different well known modelling approaches. In the current version 2.0 many improvements have been made. Inspiration sources that have been important are among others the Specification and Description Language (SDL) and the Message Sequence Charts (MSC) both
originates from the telecom industry. SDL was a source of inspiration when the Composite Structure diagram was included in the UML2.0. MSC have influenced the new version in the way Sequence diagrams are made and structured.

Another important change is that the hardware now is powerful enough to automate much of the development process. The UML has as mentioned been thru a major revision that had the intentions of making it much more suitable for Model Driven Development. The superstructure is finished and three other parts (infrastructure, OCL and diagram Interchange) are following[2]. The discussion now is what approach that is the most suited for the task of Model Driven Development which is the overall goal.

At present the approaches can roughly be divided into two main groups. One of the groups wants to build on the existing standards from OMG the UML and / or MOF the second consortium are supporting different proprietary languages and metamodels and claims that the UML and MOF are badly suited for software development for tomorrow. The latter includes companies like for instance Microsoft and MetaCase; they are advocating Domain Specific Languages as the approach.

**What is explored in this thesis**

In this thesis I have tried out different approaches for developing a mobile ordering system for the healthcare sector. First I developed the example system with the use of UML2.0. It was implemented in Java and difficulties and positive experiences with this approach was collected first hand. Then I looked at how this could be done if the MetaCase approach was used. I wanted to explore the tool and create a Domain Specific Language for the task. This is demonstrated and special attention is paid to this solution because this is the less common and there has not been written as much about this approach as what is the case with UML. At the end of the project an UML2.0 to JavaFrame compiler became available, and this was also tested.

Developing a DSL is new to me and I will use my example system to get first hand experience on what is required for doing this. Literature in the field will be explored and example systems from the industry too. What are the success factors and what are the pitfalls? When will it be beneficial to take this approach, and when will it only introduce extra risks?

The whole community of software developers agree that raising the abstraction level is important. How this should be done is however a big issue. The MetaCase approach used in my example is only one way of many different approaches that can be used for making a DSL. Other approaches will also be discussed but not demonstrated as examples.

The two approaches will be explained and pros and contras will be highlighted and discussed in chapter 9. The other approaches within the groups will also be included in the discussion.
**What are the goals**

The goal of this work has been to get insight in the different approaches mentioned above. Then I will implement an example system in two of them and do a qualitative research connected to this experiment.

There are to some extent very sharp contrast in many of the claims found in the literature. Is UML badly suited for code generation? Is it crucial that we use building blocks that are standardized? Can a DSL improve the productivity by a factor of ten and does it raise the abstraction from modelling in UML? Are the different approaches better suited for some specific purposes? Will software be produced in a factory manner in the future? What is required for developing a DSL? These are some of the questions that I will try to answer in this thesis.

**Structure of thesis**

*Chapter 1 Thesis background and problem definition*
Gives an introduction to the thesis and what is going to be explored. It describes what is going to be done and the goals of the work are presented. A short overview of each chapter is given.

*Chapter 2 Unified Modelling Language (UML 2.0)*
This is an introduction to the Unified Modelling Language; its history from it was included as an OMG standard in 1997 and to its current version UML 2.0. The different diagrams used to model my example system are described, and some examples of improvements from the previous versions are explained. The extension mechanism Profile is explained as a possibility to create Domain Specific Languages.

*Chapter 3 Domain Specific Languages (DSL)*
This chapter gives an introduction to domain specific languages and presents some tools that are suitable. The MetaEdit+4.0 from MetaCase is described in some detail.

*Chapter 4 The Pats-Lab and its technologies*
The example system developed makes use of different technologies. For accessing the telecom network an API called Parlay is used. This API is explained in connection with the ongoing challenge to make systems easier to develop and the time to marked (TTM) shorter. A short introduction to the JavaFrame framework is also provided.

*Chapter 5 The Example Meal Ordering System (MOS)*
The example system Meal Ordering System is introduced in this chapter. An example from how this is carried out in a hospital in Norway today is described. Some shortcomings are pinpointed and why MOS could be a solution is given.

*Chapter 6 The Example System in UML 2.0*
The Meal Ordering system is designed with the use of UML2.0. Some Composite Structure, Sequence and State Machine diagrams are shown. It then summarizes the process.
Chapter 7 The Example System in MOS DSL
The development of a domain specific language for the MOS domain is explained. The different domain concepts and their detailed specifications are shown. MOS DSL consists of a variety of different graph types, how these are made and how they function is demonstrated. Different examples of how Java can be generated from these models are also included.

Chapter 8 Examples of industrial DSLs
In this chapter five graphical domain specific languages from the industry are presented and explained. The languages are based on different foundations that are explained in earlier chapters; GOPPRR, MOF and the UML extension mechanism profile.

Chapter 9 Discussion and Conclusion
Several questions and claims raised and presented in earlier chapters are here discussed. Different requirements concerning language creation and use are presented. When it could be beneficial to use the different approaches is also suggested.
Chapter 2

Unified Modeling Language (UML) 2.0

Introduction

What is UML? The UML reference manual gives the following definition:

"The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artefacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."[3]

The language models static structure of objects and the relationship between them. Dynamic behavior shows the objects over time. This can be internal behavior or communication between them to accomplish a specific task. UML can be seen as a result of many different modeling languages melting together. In the beginning of the object oriented paradigm there was no standard way of modeling systems. It was mainly Booch, Jacobson and Rumbaugh that collected the experiences and produced the first proposal for a standard UML 1.X. The language quickly became very popular and it was adopted as an OMG standard in November 1997. Since then the language has evolved and new and better functionality has been added.

UML 2.0 is the newest version of the Unified Modeling Language. And is also one of the languages used in this paper. Its main focus was to better support MDD /A (Model driven development / architecture).[4] But new and better support for modelling real-time telecom applications was also provided. In the beginning UML was mainly used for modelling static structures, but with the idea of MDA the need for more powerful dynamic modelling techniques became visible. There is now possible to model executable UML (xUML). This chapter describes the improvements in the new UML version. It should be noted that not all of the revision is at this time finished. It consists of four different parts:

- **UML 2.0 superstructure.** The superstructure includes the different diagrams and the elements that comprise them are finished.
- **UML 2.0 Infrastructure:** The infrastructure defines base classes that form the foundation not only for the UML 2.0 superstructure, but also for MOF 2.0.
- **UML 2.0 Object Constraint Language (OCL):** This allows setting of pre- and post-conditions, invariants, and other conditions.
- **UML 2.0 Diagram Interchange:** This specification extends the UML metamodel with a supplementary package for graph-oriented information, allowing models to be exchanged or stored/retrieved and then displayed as they were originally. [2].
**Improvements in the new version**

UML 2.0 defines thirteen types of diagrams, divided into three categories:

- **Structure Diagrams** include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.

- **Behaviour Diagrams** include the Use Case Diagram, Activity Diagram, and State Machine Diagram.

- **Interaction Diagrams**, all derived from the more general Behaviour Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

There have been two primary thrusts of the RFP (Request for proposal) requirements: scalability and architecture. The changes in UML 2.0 for architecture are primarily in the structural (class) model while the changes for scalability are best seen in the improved sequence diagrams. But this is not all. The behavioural modelling capabilities have also been improved a lot. In UML 1.X, the different behavioural models were independent, but in UML 2.0, they all derive from a fundamental definition of a behaviour (except for the Use Case). UML 2.0 lets you designate that behaviour represented by for instance a State Machine Diagram is the behaviour of a class or a component.

**Nested Classifiers**

In UML, almost every model building block you work with (classes, objects, components, behaviours such as activities and state machines, and more) is a classifier.

In UML 2.0, you can nest a set of classes inside the component that manages them, or embed a behaviour (such as a state machine) inside the class or component that implements it. This capability also lets you build up complex behaviours from simpler ones.

**Composite Structure Diagram**

The notions of Component and Subsystem are clearly architectural in scope but how are they related to classes and how do they relate to each other? The UML 1.x specifications were vague on these issues. UML 2.0 has introduced the concept of a Structured Class—a class that is composed of parts with an explicit "nested" notation. The concepts that describe a class internal structure are **Parts, Port and Connector**.
**Parts**
A part is an internal property of a class. If the outermost class dies, the part follows. This is the same as for composition associations. An instance of a part can only exist in one class.

**Port**
A port describes the connection point between parts. They can provide both required and supplied interfaces.

**Connector**
A connector is an instance of an association between two or more parts. It is possible to connect via a port or directly to the part.

In UML 1.x composition were used for this purpose. This is still a possibility, but the focus is different. The Composite Structure Diagram clearly shows which parts the class is composed of. In the diagrams below it is easy to see the benefits of this.

![Figure 1 Association in UML](image1)

In the figure above it is possible for the button to be connected to both a stopwatch and a cellular at the same time. This is obviously not the intended meaning of the model.

![Figure 2 Class with internal structure Part, Port and Connector](image2)
The model should express that an instance of a button is a part of a cellular and another instance might be part of a stopwatch. In the Composite Structure Diagram in figure 2, this is stated the way it was intended. It does however not say much about i.e. deletion prorogation.

As with associations it is possible to show multiplicities on the format [n..m] where n is the initial and m is the maximum number of instances that can be created.

**Sequence Diagram**

Sequence Diagrams are used to describe interaction sequences between parts of a system. They are simple to both use and read. But this does not mean that they are not powerful. The diagrams have been used for a number of years. In 1992 they were standardized in Z.120 (Message Sequence Charts - MSC). The diagram was incorporated in UML in 1999, but the variant was a very restricted one. Only the most basic functionality was included.

They emphasize the interaction among objects in a run. They are seldom complete and normally they only describe a small portion of the total number of possible interactions. For instance only the main functionality with a small amount of negative traces could be shown. The point is not to show it all, rather just enough.

The new version of UML introduced a lot of new functionality. One such functionality is the Interaction Fragments. This has done the scalability of Sequence Diagrams much better. It is now possible to break up a Sequence Diagram into smaller pieces which can be represented in their own diagrams. But there are other operators too.

The operators include:

- sd – named sequence diagram
- ref – reference to "interaction fragment" defined elsewhere
- loop – repeat interaction fragment
- alt – selection between different paths
- par – parallel execution of traces
- seq – partial ordering
- strict – strict ordering
- assert – required, if something has happened this has to follow
- opt – optional, could or could not happened, but not required
- neg – "can’t happen" or a negative specification
In figure 3, we can see four examples of combined fragments with operators: “alt”, “neg” and “opt”. The inner “alt” fragment shows that the reply from the system could be either data (the position) or an error message (unable to locate). It also shows that if data is delivered back to the ServiceUser it would be a negative or impossible trace to send an error message after delivery of data. The “opt” combined fragment is the same as an “alt” fragment with an empty alternative. The ItemStore part is also worth noticing because it has an inner connectable part which is not very often supported in UML tools, nor is it in Rational Software Architect (RSA) [2, 5-8]
**State Machine Diagram**

A State Machine Diagram describes the behaviour of a single object. It describes all of the possible states that a particular object can get into and how the object’s state changes as a result of events that reach the object. A state is a stable situation where the process awaits stimuli and it also represents the history of the execution. Different kinds of triggers can cause a transition: signals, timeouts, operation calls and change in value.

There have been developed and used many different kinds of state machine diagrams the past decade. UML State Machines are based on David Harel’s state charts from 1987.

A State Machine Diagram in its simplest form is build up of an initial state, transitions, states and a final state. There may also be activities associated with the states.

![Figure 4 Simple State Machine Diagram](image)

The diagram above has one state, one final state, one initial state and three transitions. In this diagram it is very easy to see what happens inside the part. All this functionality was present in UML 1.x. Examples of new functionality in UML 2.0 are:

- Entry and exit points on Composite state. This makes the State Machine Diagrams much more scalable. UML 1.X. had no restrictions on how to enter or leave. Entry and exit points make it easier to reuse composite states in other state machines. And it encapsulates functionality in different parts of the model.
- State groups make it possible to share common behavior among different states. For instance a TV reacts the same way when it’s turned off no matter if it’s in regular mode or text mode.
- Generalization makes it possible to inherit and specialize behavior.
The State Machine Diagram in figure 5 shows the use of exit and entry points on a sub state. One transition leads into the state and there are three different transitions leading out.

![State Machine Diagram with substate, exit and entry point](image)

**Figure 5 State Machine Diagram with substate, exit and entry point**

In figure 6 the internal structure is shown. This state machine would be possible to reuse in another state machine if it has some common functionality. The three different exit points “posNok”, “itemNok” and “ok” leads as we can see to different effects in the outermost diagram. They are named and placed on the frame of the state machine. [7],[9],[3],[10].

![Internal view of State processRequest](image)

**Figure 6 Internal view of State processRequest**
**UML Profiles**

Sometimes there may be necessary to extend the UML language. For this purpose UML offers the ability to create a profile. A profile in UML makes it possible to extend the language without making changes to the underlying metamodel. The extension mechanisms are constructed in such a way that tools can store and manipulate them without knowing their full meaning. All the extensions are organized into profiles. The profile would be a set of extensions that supports a particular domain or a special purpose. The different mechanisms are:

- Stereotype
- Tagged values
- Constraints

The mechanisms are not suitable for all purposes, and if it is possible to solve the problem in regular standard UML, this would be preferred in most cases. This is because of the disadvantages of using different dialects.

**Stereotype**

A stereotype allows you to create new model elements that are incrementally different. This is done by extending the semantic of an existing UML model element. The stereotype element is identified with a string enclosed within a pair of angle brackets («stereotype>>). The use of stereotypes is pretty common in UML for instance the «include>> and «extend>> relationships. Any model element can be stereotyped, classes, relationships, operations and so on. Stereotypes can have their own icons and this is normally implemented in the tool.

**Tagged value**

A tagged value can be applied to a stereotype to store additional properties. Tagged values can be used for storing the name of the author of a class or for instance in code generation where additional information is required.

**Constraints**

Constraints are defined by using OCL (Object constraint language) or as informal text. Constraints can for instance say that a stereotype [Class «Person>>] must have an attribute isAlive = true.

**Existing profiles**

There are some predefined profiles that are available for use. The UML J2EE profile is one that can be mentioned. This profile has stereotypes like for instance: «EJBEntityBean>> and «EJSBSessionBean>>. UML Testing profile is another. This profile has defined stereotypes that make it able to capture information needed for different test processes.
JavaFrame also has its own profile; this is used to help the generation of Java code from UML2.0 models. Stereotype examples here are "<<Composite>>" and "<<Mediator>>."[3, 6, 11]

![Figure 7 Stereotyped Class](image)

**Meta Object Facility MOF**
A metamodel is a description of concept language. Metamodels can be described conceptually with the use of class diagrams.

"*The detailed description of instance data; the format and characteristics of populated instance data; instances and values depend on the role of the metadata recipient.*"[12]

UML can be placed into a 4-layer architecture from OMG. The architecture defines the layers M0, M1, M2 and M3.

- M0 contains objects of the real world (concrete models).
- M1 or the model layer is the models that represent part of the reality. They are abstract and correspond to models on layer M0 and elements from M2 are used for construction. The Unified Modeling Language resides in this layer.
- M2 is the metamodel layer, and here we find the UML metamodel
- M3 is the meta-meta layer. This is used to construct the different metamodels.

The result of applying a meta-metamodel to a domain is a metamodel. This kind of architecture therefore defines a very strong relationship between the different levels. It also makes scaling possible within the modelling task, upper levels support every changes or extensions needed for new requirements not taken previously into account. [10, 13, 14]
**MDA Model Driven Architecture**

The Model Driven Architecture is a framework defined by the Object Management Group (OMG). The framework is for software development and it focuses on the importance of models in the development process. The base modelling language is naturally UML.

MDA consists of four core models:

- **Computational independent models (CIM):** This model is at the highest abstraction level and offers a view of the system from the customer’s viewpoint. It is pretty much like a PIM.
- **Platform Independent Model (PIM):** This model has a very high level of abstraction and is independent of any implementation technology. It is a model that describes a system from a viewpoint where it best supports the business. Whether the implementation is on .Net, EJB or mainframe with relational database.
- **Platform Specific Model (PSM):** This model is the next step and the vision is that this model should be the result of an automatic transformation from the PIM. It could be one or more PSM generated from one PIM (most systems today span several platforms). The PSM should target a specific platform and its implementations constructs. A PSM that targets JavaFrame would typically have Composites, StateMachines and so on. A PSM would typically only have meaning to a person familiar to the specific platform.
• Code: The final transformation is from PSM to code. This transformation is normally straightforward when it comes to architecture because of the pretty close fit between PSM and code.

As we can see MDA defines four main artefacts, the CIM the PIM the PSM and the code but also how these relate to each other. The ability to transform a high level PIM into a PSM makes it possible to work on a higher level of abstraction which is making the developers able to deal with much bigger and more complex systems with less effort.[15] If there in addition is introduced a layer CIM on top of the PIM on an even higher abstraction level this would be an enormous advantage. A CIM might be very similar to a domain specific language which will be described in chapter 3 and is actually a kind of PIM.

![Three Major steps in Transformation languages](image)

**Figure 9 Three Major steps in Transformation languages**

*Transformation languages*

The OMG MOF 2.0 QVT Query/View/Transformation RFP (Request for proposal) is an initiative to create a standard for model to model transformations (PIM to PSM). The OMG MOF Model to text Transformation Language is a model to text transformation that can be used from model to for instance code (PSM to Code). This is an important building block of the MDA initiative. Without standardized transformation the MDA vision will fail to fulfill its promises. [15-17]

**UML 2.0 Tools**

As we have seen in the previous sections UML is a complex language. It is therefore obviously that the tool support needs to be good. This is necessary for the full power of the language to be exploited. Hundreds of tools were available for UML 1.X. but with the introduction of UML 2.0 the number has decreased which is quite natural. But new versions from different vendors have been released the last year, many of them with different grade of supportability. Rationale Software Architect / Modeller from IBM, EclipseUml, open source tool from Omondo and Enterprise Architect
from Sparx Systems can be mentioned. RSA and Visio are to model the diagrams presented in this report.

Rationale Software Architect (RSA)
Rationale Software Architect is part of the Rationale Development Platform from IBM. It is an integrated design and development tool that leverages model-driven development with the UML for creating well-architected applications and services. It is built on the Eclipse platform and inherits all the capabilities like for instance plug-ins. But there are some problems with such tools. They are very often supposed to support all kinds of different projects and development. This can lead to badly suited functionality. It is easier to support a specific domain, rather than all possible targets.

Visio
Visio is a drawing tool from Microsoft. It is possible to download a UML2.0 template from the web [18] which makes it capable of drawing one hundred percent correct UML2.0 drawings (and much more). It is used because of the lacking support for gates in RSA. It does not support any code generation or any fixed process. It is however the basis for symbol creation in the new initiative from Microsoft called WhiteHorse. WhiteHorse is the codename for the tool Microsoft is planning to build their Software Factories initiative on and is an extension to Visual Studio 2005.

JavaFrameTransformation plug-in
As mentioned under RSA, the tool does not have any predefined state machine transformations. But plug-ins can be made to solve this lacking functionality. This has been done by the Master student Asbjørn Willersrud at UiO this semester.


The plug-ins let you use different stereotypes that the transformer recognizes. <<Composite>> is on such stereotype. Every <<Composite>> in the model has some common structure that the generator knows about. The specific data is entered into the model manually be modelling the composition of the different parts and what kind they are. Messages are generated from the transition signals. All this happens automatically. The effects of the received signals must though be written in the Java programming language. But this is nevertheless a huge improvement compared to the manual approach. All the static structure and messages that can be generated based on the models are generated.

The stereotypes end the compiler could be claimed to be a DSL for the JavaFrame Framework. If a model specific editor had been available this would have been very similar to the approach taken in DSL creation. But as we will see later, UML profiles are not quite the same as a DSL, there are some distinct differences.
Chapter 3

Domain specific languages (DSL)

Introduction to domain specific languages

Domain specific languages (DSL) are languages that have their main focus on the domains they are supposed to generate applications for. It is not trying to fit all domains but only the one in focus. This means that the language is build up of domain concepts and not for instance object-oriented ones. The domain concepts are directly mapped to domain specific code. This is done by writing domain specific code generators that produces the entire code or making calls to an already existing framework.

In software engineering the focus on increased productivity has been present for a long time, spanning from assembler to third generation languages building better languages and more efficient compilers. The DSL approach is also one such attempt. According to [19] UML have not increases productivity, this is because of the core models are on the same level of abstraction as the programming language supported.

DSL are according to DSL tool vendor MetaCase 10 times faster than current development methods:

“Current development methods are based on the code world, leading to a poor mapping to organisations own domains and duplication of effort in problem-solving, design and coding. Developing an in-house, domain-specific method allows faster development, based on models of the product rather than on models of the code. Industrial applications of this approach show remarkable improvements in productivity and training time: up to ten times faster in both cases.” [20]
The idea is to skip some of the intermediate steps of the process. Figure 20 on the previous page shows an overview of the process from domain idea to finished product. The figure shows four different approaches for reaching the finished product. All cases start with the domain idea. The two first cases show that the domain idea was mapped into the world of code, and implemented on that specific platform. In case three the domain idea is mapped into UML models which then further some code could be generated. The supporters of the DSL approach claims that UML changed surprisingly little from the steps above. Too little code is possible to generate from UML models.

The quality of the code is also said to be better in the DSL approach. This is because it is the experts that create the language. An expert is someone with experience in the domain and has detailed knowledge of system development. He should also have developed some variants of the targeted system family before. In this way he knows how the solutions should be. These solutions are then connected to the domain specific language such that people with less expertise is able to create systems of the same quality grade as the expert. He has already defined what kind of code and how it can be connected.

The idea is also that the product gets shorter time to market because the systems can be developed more quickly. This productivity gain comes from the raise of abstraction. The traditional modelling languages do not constitute such a raise. This is because the models feature the same concepts that we also find in the code, like for instance class and return value. It becomes wrong to talk about code generation from such a model according to [21].

**Tools**

It has recently appeared a number of different tools that supports DSL creation. MetaEdit+ from MetaCase is the one used in the experiment presented in chapter 6 and is therefore explained in detail.

**MetaCase**

MetaEdit+ is a multi-user, multi-platform environment that supports both system and method development simultaneously. It supports great flexibility concerning the method used.[22]

The ideal is that the developers only have to solve the problem once. Not first in domain design, UML design and then code. To achieve this goal it is crucial that the tool supports three things: a modelling language specific to the domain, tool support for building models in the domain specific language and automatic code generation form the domain models to appropriate implementation code on the given platform. [23]
The MetaCase meta-language
The MetaCase tools are based on the GOPPRR meta-language. GOPPRR is an acronym from the different concepts in the data model and is the basis for all other metamodels created in a MetaCase tool.

Object:
An object exists on its own. This could be a patient, an order, a UML class or for instance a button. Objects can have properties like name and number. All instances of the type object supports reuse in different kinds of graphs.

Relationship:
A relationship is used to connect groups of objects to each other. A communication link between a Controller and the SmsHandler is one such relationship. Each side has a distinct role in the connection. Looking at the asynchrony message in a sequence diagram it has the roles Message to and Message From.

Role:
As mentioned above, a role helps specifying how an object participates in a relationship. Another example could be the inheritance relationship between two classes in UML where we have the roles ancestors and descendants.

Port:
A port is a connection point between an object and a role. It is part of the object an all instances share the same ports. A computer object would typically have different ports for external devices, one port for the display, the keyboard the mouse and so forth.

Graph:
The graph is a collection of objects, relationships, roles and the bindings between them. A UML class diagram would be one kind of graph, a state diagram another example. In addition the graphs hold information about the different decompositions and explosions allowed on the different types of objects and relationships in the graph.

Property:
A property can be associated with the other types. It is a describing or qualifying characteristic. For example the object name. The property is shared among for instance the objects. Properties have some characteristics: data type and its value. The data types could be a String, Number, Boolean, Text, Collection or a Non-property. The Non-property could be for instance a user defined object. So in a UML setting a class could have a property containing a non-property class. [24]
The MetaEdit tool consists of two different parts:

**Method Workbench**
Method Workbench is a tool for designing your method: its concepts, rules, notations and generators. The method definition is stored as a meta-model in the MetaEdit+ repository. The Method Workbench is for the experts with the domain knowledge. They use it to create the methods for the developers to use.
The Method Workbench provides twelve different tools divided into three groups: Method development, ancillary and managing tools. The method development tools are:

- Object Tool
- Relationship Tool
- Role Tool
- Port Tool
- Graph Tool
- Property Tool

The first four tools have a similar look and feel which is good for the learning process. It has a simple user interface that is self-explanatory.

Figure 14 shows the object tool. It has the possibility to create a new or open an existing object. In the figure the MmsHandler is opened and we see that it inherits from the super ComHandler. The project it belongs to is DSL_MealOrderingFamily. Further it is a property box where we see that the MmsHandler has an Id of type String. The Symbol button is used to apply your own drawing as symbol and it can be either drawn or imported from a bitmap drawing.
The graph tool is essential when different diagrams are made. The graph tool composes for instance a class diagram of objects, relationships, role and ports which is already defined in the other tools.

Figure 15 shows the MOS_StaticStructure_Graph. The tool has the same functionality as the object tool described earlier but in addition: types, explosions,
bindings, decompositions, constraints and reports choices. This functionality will be explained in chapter 7.

The Property tool is for creating property types for other components. As we saw both objects and graphs have optional properties. The properties can be defined earlier in already existing projects or be created for this special case.

![Figure 16 MetaEdit+ Property tool](image)

Figure 16 shows the Property tool and its functionality. The extra options in this tool are data type, widget and default value. The widget can be of different type like for instance input field or fixed list. The default value can make the finished modeling language easier and faster to use.

The ancillary tools are used to change the looks of the models available in a method. There are three tools:

- Symbol Editor: Used edit or create the representation of elements in a graph.
- Dialog Editor: Editing dialogs for property for instance an input field.
- Report Browser: Used to create code generators, reports and consistency checks.

The three remaining are tools for managing methods and their components.

- Metamodel Browser: Displays the meta data and their relations.
- Type Manager: Exporting or removing specifications from the repository.
- Info Tool: Type hierarchy of a given type.
MetaEdit+

MetaEdit+ follows the given method definition and automatically provides the developers with full CASE tool functionality: diagramming editors, browsers, generators, multi-user support, etc. In this way the development is raised to a higher abstraction layer. The developers do not need knowledge of implementation details. Rules and constraints guides the development, illegal operations are handled by the CASE tool. [25].

![MetaEdit+ Watch Example](image)

Figure 17 MetaEdit+ Watch Example [26]

The figure shows the MetaEdit+ environment from the WatchApplication example that comes with the MetaCase tool. As we can see from the menu only the domain specific symbols that is possible to use in the graph are shown. If you try to do something that is not intended or supported in the language, you will be presented an error message. [26]

The MetaCase Report Definition Language (RDL)

One of the most important aspects of both DSL and MDA is the ability to automatically generate code from models. Models should raise the abstraction level and for this to be valid some or all the code must be generated. There are several ways of doing this. In MetaEdit+ these transformations are written in so called reports. The reports are written in a language called Report Definition Language (RDL). The language is powerful and let you specify the mapping. The reports access the data stored in the repository and let you create various text output. For instance code in a java or C# file. The MetaEdit+ tool comes with a number of predefined reports that can be used on every project created:

- Object List: Describes all objects in a given graph
- Property List: Describes all the properties of the objects
- Export graph: Export the graph to different files, HTML, XML and Word document.
These are some examples of method independent reports available in the tool.

But these reports will not help you in generating code from the graphs. If this is the purpose you will have to create your own method dependent reports. This will be explained further in chapter 7 The Example System in a DSL.

**Microsoft Visual Studio**

Visual studio is Microsoft’s development platform and this platform comes with support for a suit of tools called “Tools for Domain-Specific Languages”. The tools are for creating, editing, visualizing and using domain-specific data for automating the enterprise software development process. They are supported by a code framework that makes it easier to define domain models and construct a custom graphical designer hosted in Visual Studio.

The initiative is often referred to as Software Factories and it is Microsoft’s version of OMG’s Model Driven Architecture.

**The Software Factories initiative**

Answering what it is in detail is a huge task and not done in this paper. But a short definition is given in the following sections:

“A software factory is a software product line that configures extensible tools, processes, and content using a software factory template based on a software factory schema to automate the development and maintenance of variants of an archetypical product by adapting, assembling, and configuring framework based components.” [27] p 163.

The elements involved are the schema and a template. The schema defines the artefacts and the assets used to build them. It is actually a directed graph whose nodes are viewpoints and the edges are computable relationships called mappings. It also describes the artefacts that must be developed to produce a software product. It is like a recipe that defines the ingredients, tools and preparation process for a family of software products.

All that is described in the software factory schema is great, but it is not any thing else than a description. We also need the assets to build a member of the family. Therefore an implementation of the schema is needed. This includes defining the DSLs, the patterns, the framework and the tools. Packaging them and make them available for developers must also be done. This is called a software factory template. While configuring a schema customizes the description of the software factory, configuring a template customizes the tools and other pieces of the development environment that is required. [27-29]
The idea is very much the same as with MDA, but they are not supporting the idea of building it on OMG standards. It also seems that Microsoft tends to use DSLs in a more narrow scope than what is the case with other modelling initiatives. There is a clear focus on technology dependent domains.

**Other**

**Rhapsody**
I-Logix have introduced Rhapsody 6.1 with Domain Specific Language (DSL) Support. It is based on OMG standards and uses the profile mechanism of UML for domain specific purposes. An example is shown in chapter 8.

**The Generic Modeling Environment (GME)**
GME is a tool for creating domain specific languages and has its origin at Vanderbilt University. The metamodelling language is based on the UML class diagram notation and OCL constraints[30].
Chapter 4

The PATS-Lab and its technologies

Technology
For making such an application mentioned in the introduction and further explained in chapter 5, 6 and 7, you will need platforms and APIs (Application Programming Interfaces) that has support for this. The Pats Tele-service laboratory has this support.

The Pats-Laboratory
The lab was established in September 2001. It is a co-operation between Telenor, Ericsson, NTNU and SINTEF. The laboratory provides state-of-the-art technology and infrastructures for service development and execution. The users are able to test the services on Telenors cellular networks. UiO has made a connection to this lab thru a proxy server.

The vision of the laboratory is to provide a setting for the participants so they can experiment with advanced and hybrid telecom services. Hybrid means that it is possible to offer services over a combination of heterogeneous network such as PSTN, GSM and IP. Different access types are also offered. An application may offer GUIs to both cellular phones and web browsers or applications on a computer. This is a requirement for personal mobility[31].

The lab combines platforms from traditional telecommunication and information technology.

Research in the laboratory is driven by the need to develop new services quickly and controlled. Methodologies are thus emphasized[32].

Telenor Content Provider Access (CPA)
In a situation outside of an educational institution with access to Pats, you would have to use for instance Telenor CPA. This is the service Telenor offers 3rd party developers.

“CPA represents several products enabling content providers to deliver content to Telenor Mobile’s subscribers and at the same time bill the subscribers for using the service. By doing this Telenor Mobile has made their value chain accessible to content providers of mobile services, based on a revenue sharing transaction model in return. Today CPA supports SMS, MMS, WAP and WEB” [33].

The Pats-Lab is actually connected to the same interface, but works as a proxy.
**Parlay / OSA API**

What makes the interoperability possible is the Parlay/OSA API. Parlay is a technology-independent application programming interface (API) that enables the development of applications that operate across multiple networking-platform environments.

Parlay/OSA is cooperation between many stakeholders. Many big companies are members of the group. And this is of course an important reason for using this API. When solid world wide companies are involved the chances that this will be an accepted standard is feasible. IBM, Ericsson, Telenor, Fujitsu, Oracle and Vodafone can be mentioned. They are trying to create a standard for accessing the telecommunication network.

![Figure 18 The Parlay OSA API overview](image)

**Parlay-X**

The first version that was developed has been characterized as to difficult. Parlay-X is an XML specification of OSA, and is meant to work as a layer on top of the OSA API. This would make it even easier for 3rd party developers to implement services.

Parlay-X web services are powerful yet simple, highly abstracted building blocs of telecommunications capabilities. It follows simple application semantics, allowing the developers to focus on access to the telecom capability using common web services programming techniques. Parlay-X applications can be written in any language as long as it can make the proper web service invocation.[34]
The vision and motivation is different within the consortium. The operators like Telenor, wants more traffic on their networks. The traffic in the telephone network has been reduced over the past years. This can be explained in many ways. One reason could be that people uses IP telephones like Skype or messenger services like MSN and ICQ. Instead of giving your friend a call, you just page him on your computer and can talk to him without any costs. This is of course a big competitor to the operator company’s telephone. Operators see this problem and are looking for new business models to expand. If they can develop applications that uses the network, their income will still be high and the business will be good. But it would be even better if they could create an API that was easy to use for people unfamiliar to telecom. 3rd party service developers could then develop lots of applications. They will have to pay for the use of the network, and payment arrangements with clients using the services will be handled by the operators.

Take for instance a parking payment application that lets the users pay for parking thru their mobile phones. The payment will go thru the operating company before the parking company gets the money. It will become a new part of the operator company’s business model. The failure of a project would not influence the operator as hard as it would do if the operator was the driving force behind it. This will inevitably lead to new and innovative services that more people think about new services, a classic win – win situation. This philosophy is also seen in the evolvement of 3G. Here Telenor also let’s the development of new services depend on 3rd party.[35].

In [36] the tele-statistic numbers from last years use of the mobile network was presented. In the paper it is clear that the marked is growing. The number of mobile phone subscriptions has grown beyond the number of persons living in Norway [page 9]. It is also clear that the growth in SMS messages sent has stopped. The growth is found in additional services.
3rd party service developers will be able to take advantages of the functionality in the network and develop new kinds of services that the customers demand (white). Applications will have shorter Time to Marked (TTM) because of the simplified API. This is something that the customers too will benefit from. Services can be developed quickly and on demand.

**JavaFrame**

“JavaFrame is a Modelling Development Kit that aims to improve the dependability of large complex real-time systems implemented in Java. Using JavaFrame will also make the resulting system more maintainable through the independence of system parts, and that producing a Java system with JavaFrame will be more efficient than traditional approaches.” [37]

JavaFrame comes with a Java library that supports a one to one mapping between UML 2.0 State Machines and Composite Structure. It is a layer between UML2.0 state machines concepts and the Java programming language.

An application on top of JavaFrame defines subclasses and makes objects to be used in the applications from these.
JavaFrame consists of 17 different classes the most important ones are thus:

- StateMachine
- CompositeState
- Composite
- Mediator
- Message
- State
- Scheduler

There have been some improvements to the JavaFrame Framework the last years. New add-on’s on top of this layer. First ActorFrame and then a new layer called ServiceFrame. ActorFrame comes with support for the J2EE platform, and ServiceFrame is a proposed architecture for Service Applications.[6] In this paper only JavaFrame is described and used. This is because the size of the system developed is pretty small, and the amount of work required implementing in J2EE.
Chapter 5

The Example Meal Ordering System

Background

In hospitals today it is normal that every patient get the same meal. It has been some improvements the last years, but still there is a long way to go. I have been in contact with a hospital in Norway that is forerunners in the development (called Investigated Hospital IH) The example system is in this domain and will be further explained in the following sections.

How is it done at a big Hospital in Norway?

IH is a big hospital in Norway where technology and research are in focus. Recently they have moved into a new location so the building mass is new and the infrastructure within the hospital is supposed to be indeed modernized. It is a well known hospital and they have their main assignments in the area of advanced patient treatment, research and education. Currently there are over 500 doctors and 1500 nurses employed at the hospital.

At IH the food is an important issue. They are very well aware of the importance of correct and good nutrition. The food servings is supposed to be part of the overall treatment of patients and therefore the food is to be well made, tasty and adjusted to the patients needs. In this way the patient is able to focus entirely on recovering from their illness.

To accomplish this goal it is necessary to register some information about the patients needs. So when a patient is enrolled into the hospital a personal food card is registered. This is information that will follow the patient as long as it stays in the hospital. The information on the card consists of different options.

- Default size on the meals, spanning from half to double
- Customized diet, vegetarian and minus pork (minus pork intended for Muslims)
- What kind of drink is preferred?

There is also the possibility to order extra appetising food or “wish meal”. These options are reported directly to the nurses on a day to day basis.

Further the hospital offers two main dishes each day. These meals are presented on a menu each week and are repeating them selves once a month. It is the nurses on responsibility to publish the menu each week. They also have to collect the orders from the patients.
About thirty percent of all the meals served in the hospital are special or diet orders, so the need for such an offer is obviously present.

As we know there are different restrictions in the world’s religions today, for instance would many Muslims prefer Halal meat and Jews would probably wish a kosher meal. This is not something that is offered at IH. No ethnic meals are provided.

There are four fixed servings of food, breakfast, lunch, dinner and evening. If there are any need for an intermediate serving this is also possible, but this order must be made to a nurse that brings the information to the kitchen.

The children are offered special menus and there are buffet servings for more flexible eating. There is also the possibility to order pizza, the kitchen then need the order in some advance.

An electronic system that was supposed to help the nurses with the ordering task has been tried out, but the response was not overwhelming. The nurses meant that it was way too time consuming for them to register all the information on a terminal. Their main responsibility was to take care of the patient’s health, and not ordering food.

As we see the task of feeding a great number of patients is a difficult and time consuming task. My example system the Meal Ordering System (MOS) is one potential solution that would potentially help this situation.

**My example system MOS**

The Meal Ordering System (MOS) is a hybrid context aware mobile application. It is a system that is supposed to handle food ordering in a healthcare setting thru the use of mobile phones. The healthcare sector is a growing marked, there are lots of potential customers and the field is not very well explored.

The idea is to take advantage of the fact that almost everybody has a cellular phone or will in a few years. And integrate mobile phones and their capabilities in the treatment of patients in a hospital. The importance of nutrition when people are sick does not come as a surprise. And we know that diseases and medications may influence the eating habits of the patients. This is an important issue that should be handled properly, not only for the patient’s sake but for the hospitals financial situation too.

Normally there is no choice of menu in a hospital. So if you don’t eat fish and that’s for dinner that day, you potentially would have a problem. This is likely to become a much bigger issue in the coming years. Cultures and religions are melting together all over the western world. It is already a problem for people with different religious restrictions to be enrolled into a hospital. There are no options of Jewish or Muslim food in most hospitals in Norway.
Another point is that the amount of food very often is the same for every patient. This is not feasible neither for the patient nor the hospital. An enormous amount of food goes straight into the garbage every year. At IH the number is calculated to be approximately 25 percent. It would be much better for both parts that size was handled by the individual. As mentioned above, IH has some kind of arrangement that is handling this, but not on a daily basis. And if so, it is a very time consuming task in an already busy environment so naturally it is not the main priority.

The system can also easily be expanded with other similar services. This can be ordering goods from the hospital kiosk, books from the hospitals library, making an appointment at the hospital’s hairdresser or other similar services.

The communication can be done over different kinds of networks. Mobile phones are the ones used in my test system, but also web terminals, digital TV screens and so forth could be supported. It is a fact that mobile phones are not the ideal solution for this purpose but that is not the point here either. The point is mobile personal control over meal ordering.

**How does the system work**

The system has many different scenarios. Some potential behaviour is described in more detail in the next sections, but not all. The infrastructure of the base system makes it able to cope with lots of different scenarios.

When a patient is enrolled in a hospital or healthcare institution, different data is stored electronically. This information includes the information on the food card from IH, room number, allergies and so forth.

The patient will then be presented with a menu each day. This offer is sent in an SMS message. It is sent from a terminal in the kitchen, when today’s menu is decided by the chef. The menu is based on the patient information registered on that phone number. If the patient is a vegetarian the menu will not include any meals where meat is one of the ingredients. The same will happen if the patient is registered to be a Muslim, Halal meat is then the ingredient used. Should the patient be allergic to for instance milk or potatoes, this will be excluded from the menu. In my system this information is stored in the specific system, but it would be possible to connect my system to an already existing EPJ system, this is though out of the scope of this report.

The patient could choose to reply with different optional keywords. Where the keywords could imply changes to the size, drink or meal. If none of the information is supplied, the default data will be used. If he tries to order a large menu and he is registered on a weight loss program the changes would have no effect.
If the patient has difficulties deciding which meal to choose he/she could request an MMS of a potential meal. For people that have difficulties to eat, it might help to have some options and a picture could also be to assistance.

**Sample Use Case diagram**

The Use Case Diagram in figure 21 shows some potential Use Cases of the system.

![Sample Use Case diagram](image)

**Figure 21 Some Use Cases from The Meal Ordering System**
Chapter 6
The Example System in UML 2.0

Introduction
I first started out modelling the Meal Ordering System in UML 2.0 I have used Composite Structure Diagram, Sequence Diagram, State Machine Diagram, Class Diagram, Use Case Diagram and Interaction Overview Diagram in the development of the MO System. Some of the diagrams will be shown in this chapter and the rest is found in the appendix. RSA is the tool used on most the diagrams, in some cases Visio.

The UML 2.0 Diagrams
The task started out with the definition of potential Use-Cases showing the wanted functionality the system should be able to provide. No descriptions were written, instead Sequence Diagrams were used. But before this task started it was necessary to find the concepts of my system. This is shown in the Class Diagram found in the appendix.

Composite Structure Diagrams
The Composite Structure Diagram is good for showing the communication infrastructure of the system. Figure 22 shows the outermost diagram of the Meal Ordering System, with its internal parts.

![Figure 22 MOSystem Composite Structure](image-url)
The parts are both Composites and StateMachines. StateMachines can have no internal parts but the Composites can.

The SmsHandler is responsible for the communication with the environment. If the communication type should change or needs to be extended to support for instance HTTP it would be simple to include this.

As we can see the system is build up in a star architecture where all the communication goes thru the Controller. The Controller is a Composite which has its own StateMachine ControllerSM. This StateMachine is responsible for routing and creation of SessionController objects. If the ControllerSM receives a message with an unknown routing address the creation of a new SessionController is initiated, if there already exists an entry in the routing table the message is just forwarded.

The choice of architecture is done for extendibility purposes. It has some influence on speed, but this was not the main requirement for my system.

![Figure 23 Composite Structure Diagram Controller](image)

As we can see the ControllerSM gets all the messages that come from the environment. It has one outgoing port that can be connected to many objects of type SessionController. The port fromCSMtoSC is therefore transformed into a router mediator of type MOS_RouterMediator which is a special mediator type defined in
my system which extends the JavaFrame Mediator type. It could have been solved by just using the SimpleRouterMediator already contained in the framework, but I found it feasible to use my own. The Java implementation of the mediator is found in the Appendix B page 123.

**Sequence Diagram**

The sequence Diagrams were modelled in Visio instead of RSA. This is because RSA is lacking support for this diagram type or more correctly all its functionality. Lifeline decomposition and messages to and from frames is a vital part of UML 2.0 Sequence Diagrams but is not supported in RSA.

This makes it very difficult modelling true UML 2.0 in RSA so I decided that this was better done in Visio with the UML 2.0 template [18]. The Sequence diagrams are fairly easy to draw when the structure of your system is in place. It also reveal pretty quickly if your Composite Structure diagrams are bad suited for solving the intended problems.

![Sequence Diagram with reference occurrences](image.png)

*Figure 24 Sequence Diagram with reference occurrences*
The diagram in figure 24 shows an overview of the functionality in the system. Here it is easy to see the real power of reference occurrences and alternative combined fragments. The ordering on the lifelines clearly shows that a ServiceUser must be registered before any of the other functionality can be used.

The MealOrder reference occurrence is further described in sd MealOrder. This sequence is shown in the figure below. Here we can see the more detailed description of what is going on inside the reference occurrence. Another thing we can see is that the lifeline belonging to the MOS part has gotten state like figures called StateInvariants [39] on the lifeline. This is something that visualizes what the MOS part is performing.

![Figure 25 The MealOrder referenced in overview](image-url)
The MOS Lifeline object is further decomposed into more detailed sequences. One such Sequence Diagram is shown in the figure below.

In these three diagrams we can clearly see the structuring power introduced in UML 2.0. It is easy to follow the detailing of the traces. And as we can see it is a nice error checking method to count the messages to and from the outermost main frames and compare them to the lifeline that are decomposed, in this way we are certain that the diagrams are correct decompositions concerning consistency in messages. Further we see that there are two global combine fragments with the “alt” operator. These can be found in the diagram that references this one and therefore they are made extra-global. We could probably expect that some of this functionality will be automated in
tools in the future, but since RSA does not support gates yet it becomes impossible at this stage. Global combined fragments are not supported either.

**StateMachine Diagram**

While the Sequence Diagrams are good for showing the communication between the objects they do not say anything about what’s going on inside the objects. For this purpose I have used StateMachines. They have been drawn in either Visio or RSA. When the implementation was done manually it was simpler to use only one program therefore Visio were preferred. But when the JavaFrame compiler was made available they were drawn in RSA to. Two examples (Menu and SessionController) are shown below.

![StateMachine Diagram](image)

*Figure 27 State Machine Showing the Menu states and transitions*

The first one we can see is a pretty simple one, only one state and two transitions. But there are potentially more functionality required in a more complete version and there for it was decided to separate this functionality in a separate StateMachine. Two different effects are shown, the one signal that says “GetMenu” result in an output message with the menu as parameter.

![StateMachine Diagram](image)

*Figure 28 The Update Menu Activity*
The “UpdateMenu” signal requires some more internal behavior and is therefore described further in the Activity Diagram that is attached.

The activity represents method calls to existing functionality available. The entire Java code could also have been written directly.

The next StateMachine diagram shows the SessionController. This is responsible for handling communication sessions with the patients. A new SessionController is dynamically created each time the system communicates with a user that is not already registered in an active session. A session is in this example ordering of food. It starts when the ordering process is initiated and lasts to the ordering of the specific meal is finished and registered in the OrderStore.

![StateMachine diagram](image)

**Figure 29 The Session Controller StateMachine**

The destruction of the object is based on a timer that is activated when the Idle state is entered.

As we can see RAS provides special mechanisms that can be used both to display signals and effects. If we compare the diagrams drawn in RSA with the ones shown in Appendix A, the power and advantage of using a dedicated tool is obvious. First of all we can not generate code from Visio and second the symbols uses on transitions are much better in RSA than the Visio pure textual approach. It could be argued that it is possible to draw what ever you would like in Visio but then again this includes more work.
The development process: How was it?

In general
The development of the system in UML2.0 was time consuming in the beginning. This was because of well known problems of object oriented development. Finding the right names and responsibility for my concepts was not straight forward. But with some discussion with my supervisor we landed on the proposed structure. It could always be claimed that the structure is pretty simple and not to complicated, and that I agree on if the system were to be developed all over again. I now have much more experience in the domain I am working with so it is obviously that the time spent the first time would be a magnitude of what it would have been the second time. When I first got the structure right, the further development went quite easy. Modeling state machines have been done on some occasions before but still it is not trivial. In the first iteration the coding was done manually, this is done with good support in the JavaFram framework which is almost a one to one mapping from model to code. The second time I used the newly created JavaFrame compiler. This automation gave me a first-hand experience with automation of system development. The second approach was much more effective and it provided the ability to just focus on the dynamic behaviour, the structure was already in place.

Tool support
For modelling the system Rationale Software Architect from IBM was my preferred choice. I have used previously released versions from Rationale and believed that this was the tool for my experiment. And for most of the tasks this was the case.

![Diagram](image)

Class, Composite Structure and State Machine Diagrams are well supported. They are easy to model and the tool gives you great support and flexibility. But when it comes to Sequence Diagrams this is not the case. This was unfortunately such a big problem
that another tool was needed. It was not possible to model correct and needed UML2.0 functionality.

The most severe shortcomings were:

- The lacking support for gates: It is not possible to send messages to or from frames. This is a main functionality for structuring in UML2.0. Interaction occurrences can not be placed inside a diagram where not all the lifelines are involved. Here it became needed to model an anonymous lifeline part as shown in the figure above.
- Lifeline decomposition: It is not possible to model lifeline decomposition properly. If you want to do this it is necessary to use notes, and there will be no formal connection between the decomposed lifeline and the target sequence diagram. This is in connection with the lacking support for gates.
- It is not possible to make combined fragments global, this compromises the UML2.0 specification.
- Modeling of continuations. This is solved with the use of a reference occurrence with the name of the target.

The two first problems were the most damaging ones but there are no code generation from Sequence Diagrams so they would not have caused any problems to that process at all. But messages out of frames are impossible to model what so ever and lead to messy diagrams.
Chapter 7
The Example System in a DSL

Introduction
After first modelling the MOS in UML2.0 and implemented it in Java I started out thinking about the DSL. A trial version of the MetaCase tool was downloaded from their site. The trial period was 30 days so I figured that it was a good idea to find out if this could be extended. This was no problem and I got a patch that made the tool available for 90 days extra. I started the work in June 2005 and closed the experiment when my license expired in the end of September.

Both the tool and the meta-language used in the tool were new to me so I had to spend some time going thru some tutorials and examples. There were good documentation and examples available with the tool. However was the process of coming up with a good idea on what to create a difficult task. I did not experience any real problems concerning the UML approach which I believe would have made it easier to get ideas on what could be done in a better way. Making the language more abstract became the main challenge. When you are going to create a DSL there are different approaches suggested depending on what kind of system you are to develop and what is available.

The different definition approaches are:

1. Domain expert’s or developer’s concepts: Using knowledge of the domain that the experts holds
2. Generation output: Look at the wanted output from the language and base the definition on this. For example an XML file
3. Look and feel of the system build: Look at the wanted GUI and how this should look like, then define the language based on these concepts
4. Analysing the variability space: Analyse the different members of the potential system family and divide commonality and variability space. Encapsulate the commonalities in the domain concepts and make the variability space possible to change.

They are very often used in some kind of combination and this was also the case in the process of developing the MOS DSL where both 1 and 4 were utilised.

The MOS Metamodel
The process was then carried out in an iterative manner. One such iteration is explained below.

For defining the MOS metamodel it was necessary to focus on the domain concepts already defined in the UML models, which concepts to include was a big issue. But I
ended up with mostly the same domain concepts used in my UML Class Diagram, which I find to be quite natural because my Class Diagram contains the domain specific concepts and more of the system. These main concepts were:

- ComHandler: Super type that the other handlers inherits from
- SmsHandler: For communication thru sms messages
- WebHandler: For communication over HTTP
- WapHandler: For communication over Wap
- Controller: The central part of the system that should control the communication between the different parts of the system. And also be responsible for creating sessions.
- SessionController: Each communication initiative is given one
- Menu: A list of meal proposals offered to the patients. Consists of meals and drinks.
- Drink: Might be in a menu
- OrderStore: Consists of many different orders placed by patients
- Order: Order belonging to a patient
- PatientStore: Collection of many patients
- Patient: The representation of the patient and its data like for instance the food card

The list above describes the concepts in the language. All of them are "objects" of the meta-language GOPPRR.

All these concepts communicate and for this the relation type “Communicates” is introduced. This relation is simply a representation of four mediators in JavaFrame that is connected to each other. The second relationship is the “Has” concept. This is similar to the UML association composition.

When the domain concepts were found the variability and commonalities needed to be analysed. Which parts should be possible to change and which parts were common to the entire system family. The Menu, Drink, PatientStore, ComHandler and the Controller were found to vary. The Variability of which communication to be used, were solved with different objects that inherited functionality from the super object ComHandler (WebHandler, SmsHandler and WapHandler). Variability in the other objects are explained in the following sections. Some parts of the analyses are found in Appendix C.

**Example System modelled in MOS DSL**

When all the concepts are in place it is possible to define different graph types. The graph type includes the objects and how they can be connected. Constraints can be set on multiplicity in connections between them.
The MOS_StaticStructure_Graph
The MOS_StaticStructure_Graph shows the different parts that the system consists of. The parts are all the objects defined in the metamodel including the “Communicates” and the “Has” relationships.

In the graph it is defined legal connections between them. An object of type SmsHandler can for instance not be connected directly to a Menu object. This conforms to the rules of the languages, and the MetaEdit+ tool that is generated from the graph definition makes sure this does not happened. It is also impossible to connect for instance two Controller objects to the menu.

Trying to connect the controller to the drink object in the figure above will result in an error message box as shown in figure 32.
There is also possible and necessary to define constraints on the parts of the graph. For instance if you need to define that an object of a type only can be in a maximum number of relations to another object you will have to define this someplace. This is done in the Connectivity Constraint Definer shown in the figure below.

If this constraint is broken it will result in an error message from the modeling tool, and in this way it guides the modeler in doing the right things. In UML there are no such tool checking mechanisms. If you want to make a connection between two classes it is nothing that stops you from doing this illegal operation. It is the modeler himself that must insert multiplicity constraint on his parts. In the DSL approach this can be handled by the expert and encapsulated in the language.

**The tool tab**
As mentioned before the MetaCase tool will automatically generate the modeling environment for you. This includes the modeling area where you place your model elements, a fixed tool tab and a graph specific tab based on your objects and relations.
As we can see of the figure above it is easy to locate for instance the controller object or one of the other domain concepts. The legal relationships can be shown with text symbols like the [Co] or as a graphical symbol. The textual variant is shown when there is not defined any specific symbol for the relationship.

The interface for creating graphs is well formed. Adding an object or a relationship is done by just right clicking the field and selecting from a list of possible types. It is possible to define whatever you will find necessary which is one of the MetaCase tool’s most effective functionality. If you want to update the graph type with a new none existing object, you just define it in the object tool and then include it in the graph type definer. The tool tab will be updated automatically. This is in accordance with the evolution of tool requirement described in [40] where it says that one of the main goals of MDA tools is that when the underlying model changes so should the tool.
The Menu object

Further if we look at the Menu object we see that this object has different properties that can be changed. This is one way of modeling different family members. The different properties can be set such that eight different systems might be generated. The information in the model is used to choose what kind of menu the system will have.

The different options are:

- Normal: Only the basic functionality is included. The menu does not consider the different patients special needs
- +BMI_Sensitive: The menu is decided based on the patient’s body mass index (BMI)
- +Religion: The menu presented is based on the patient’s registered religious beliefs
- Full: All functionality is included

The dialog box is automatically generated from the properties in object menu. It is easy and impossible to choose options that are not allowed or nonexisting.
**MOS_Menu code generation**

The data collected from the model is used to generate the code for the wanted menu. There is as we will see different levels where this can happened. In a case like this where we know the different menu types and how they are implemented it would be reasonable to use template files. There could be one master menu template and four smaller for each menu type. In this example I have chosen to use four different templates, one for each property selection.

![Figure 38 Code Generator report for Menu object](image)

The report example in the figure above shows the part where the MenuSelection property is tested. Based on the collected string the correct template file is chosen. In this approach there is no problem if you would like to extend the menu object with another MenuSelection choice. Simply edit the fixed list in the menu object’s property MenuSelection. Then edit the code generator report to include a selection for the new entry. The required template file must also be developed and made available.
The OrderStore object
Similar technique is used for the OrderStore and the Patient concept. It makes it quite easy to model the system, and if the language is defined correctly it ensures that the choices you make lead to a legal system family member. As we can see in Figure 39 it is possible to select the maximum time an order is stored in the system. This would typically be represented with a numbered collection of some sort in UML.

![Order history property dialog](image)

Figure 39 Order history property dialog

This approach is on a higher abstraction level and talking about weeks instead of array sizes makes it closer to the domain. And in theory the user of the language does not even have to know what an array is.

The MOS_PatientStore Object
The PatientStore object represents the collection of Patient objects in the system. It consists of the creator state machine and 1..* Patients. The creator state machine has the same behavior independent of the system that is made. Its task is to create, delete and route the messages to the correct patient object. Because of this homologous functionality all these behaviors are encapsulated in one object namely the PatientStore. However it is possible to choose between the context aware and the “normal” patient object.

![PatientStore with property dialog](image)

Figure 40 PatientStore with property dialog
Further you can right click the object and choose to explode it in a new graph type. In this case this graph type is set to be the MOS_Patient_StateGraph which is explained in more detail in its own section later. The encapsulation of many objects into one, raise the abstraction for the developer, and technicalities are not something he has to bother understanding or modeling. This has much in common with a UML component that consists of different internal classes that is private, but it offers a clear interface to the user / system.

The MOS_Sequence_Graph
The MOS_Sequence_Graph is used in much the same way as UML2.0 Sequence Diagrams. But there are differences. The legal actions are also here much more restricted. And functionality that is not needed from UML is not included. In the UML approach it was planned to use IBM Rational Software Architect for modelling sequence diagrams but this was not possible in the way I wanted. The decomposition and messages to and from frames restricted me from doing this. This functionality is added to the MOS_Sequence_Graph.

The MOS_System only uses Asynchrony communication and therefore this is the only available communication between the different parts in this graph type.

There is however the possibility to use the <<create>> message, but in the final version this functionality is not something that is necessary. The <<create>> functionality is encapsulated in the language because it works the same way in all the variants. The configuration of the legal bindings, relationship and roles are done in
the Graph binding definer dialog. Here it is possible to specify that an asynchronous message is a legal relationship between objects of the Controller and the MosLO_Menu. In the figure below each relationship has two roles defined: Message from and Async message to. We can see that it is possible to send a asynchronous message from objects of type MosLO_Menu, OrderStore, PatientStore and SmsHandler. And the legal receiver is MosLO_Controller.

Figure 42 Graph binding definer for MOS_SequenceGraph

If this was a UML2.0 Sequence Diagram it would have been possible to create whatever life line object you wanted. In the MOS version only concepts from the domain is possible to model. They are for simplicity also shown on the tool bar together with the legal asynchrony message.

Figure 43 MOS_Sequence_Graph ViewOrders
The MOS_Sequence_Graph is a specialisation of an already existing graph type in MetaEdit+, (the implementation of the UML language). The fact that there are already finished graph types that you can specialise or use as a super is yet another powerful and very useful functionality of the tool.

Figure 43 is another example of a potential play in the system. The different message names with parameters is easily selected or created from a property editing dialog. These dialog boxes like the one in figure 44 can also be edited such that the look and feel conforms to the developer’s wishes.

The MOS_Sequence_Graph Code Generator
The messages and the objects can also be generated from the MOS_Sequence_Graph. But since the objects are generated from the MOS_StaticStructure_Graph only message generation is shown here.
All the messages to and from objects in my domain, are asynchrony. In my target domain they are subtypes of JFrame Message, and can therefore be generated from the graph.

The report in figure 45 will result in the generation of JFrame Message code. Each distinct message in a given MOS_Sequense_Graph will be transformed to a \([\text{Message name}].java\) file.

The generated code from the message in the figure above is shown below. We can see that the parameters have been added as attributes and that the constructor has been given the right parameters too.

```java
public class DisplayOrderHistory extends Message {
    public PhoneNumber pNo;
    public int period;
    public DisplayOrderHistory(PhoneNumber pNo, int period) {
        //Todo: Fill inn constructor details
    }
```

**The MOSStateMachine_Graph**

The MOSStateMachine_Graph is closely related to UML2.0. But the transitions are specialized and called MOS_Transition. The transition consists of one input message and one action. The action can include code and an output message. This is exactly the functionality needed in the system. If this graph type is used the variations of the system increases and it becomes more likely to make mistakes. Therefore two different approaches are shown. The first one implements a quite regular StateMachine concept with few restrictions.
In this graph it is possible to name the states whatever you would like. It makes the language much more general but using it requires much more knowledge of modelling and software engineering than the more restricted one. As we can see the states are named nr1, nr2 and nr3 which has nothing to do with my health care domain.

**The MOS_StateMachine_Graph Code Generator**

The code generator for the MOS_StateMachine_Graph is as the graph itself is able to handle whatever state names supplied. A JavaFrame StateMachine has lots of code lines that are the same for every variant. This code is written directly in the generator in this example.

It would also be possible to put it in template files as done in the MOS_StaticStructure_Graph code generator. The JavaFrame StateMachine itself looks the same except the name of the class and the constructor. It is in the JavaFrame CompositeState the differences is found.
In figure 49 a report from the MetaCase tool is shown with a little editing of the lines containing “ControllerSM” it would have been suitable for almost any StateMachine.

```
static State ':State name; = new State('':State name;''); newline;

public ';'States(String sn)
{
    super(sn);
    newline;
    ':State name; = enclosingState = this;
    newline;
    public void enterState(StateMachine curSm)
    {
        entry(curSm);
        newline;
    }
    protected boolean execTrans(Message sig, State state, StateMachine curSm)
    {
        ControllerSM cSm = (ControllerSM)curSm;
        newline;
        foreach :State [UML]
        {
            newline;
            if (st == ':State name;')
            {
                newline;
            }
            newline;
            foreach >MOS_Transition
            {
                newline;
                newline;
                if (sig instanceof ':Condition;')
                {
                    newline;
                }
            }
        }
    }
```

“':State name;” spelled in green is a substitute for the object in focus. If we take a closer look from line 7 in the figure we see that the code generator traverses thru each state and adds the “.enclosingState = this” to the state name. The same “foreach” code word is used to go thru all the MOS_Transitions and get the condition the transition is triggered. This is then inputted in the code as “if statements”.

As we can see the generator produces all the code needed in the CompositeState. First all the state objects are created with the names from the model, then the constructor, and finally the execTrans method. The code within the “if statement” are not generated, but could easily be gathered from the action property of the MOS Transition. The action data must have been Java code, or some reference to a template file.
As we saw, the MOS_StateMachine_Graph was not very restricted; therefore I also tried to make a graph that was more constrained for the developer to use. This graph is created from the MOS_StaticStructure_Graph. It is done by right clicking the PatientStore object and choose explosion. Then you are able to create a new MOS_PatientState_Graph.

In figure 51 on the next page a MOS_PatientState_Graph is shown. In this graph less functionality is supported and there are only a fixed number of legal state names. This variant I believe is more correct and better in a DSL approach, and it also makes it easier to write a better code generator that generates a larger amount of the finished code. Though it involves a bit more work developing it. But once it is done it can be reused and the development of the system can be done quicker and with a better result. The more detailed the language gets, the easier it is to generate 100 percent code.

```java
public class ExampleSM_States extends CompositeState {
    static State nr1 = new State("nr1");
    static State nr2 = new State("nr2");
    static State nr3 = new State("nr3");
    public ExampleSM_States(String sn)
    {
        super(sn);
        nr1.enclosingState = this;
        nr2.enclosingState = this;
        nr3.enclosingState = this;
    }
    public void enterState(StateMachine cur fsm)
    {
        entry(cur fsm);
    }
    protected boolean executeMessage(Message sig, State state, StateMachine
    ControllerSM cur sm = (ControllerSM)cur fsm;
    if (st == nr1)
    {
        if (sig instanceof msg1) {
            
        }
        if (sig instanceof msg2) {
            
        }
    } 
    if (st == nr2)
    {
        if (sig instanceof msg1) {
            
        }
        if (sig instanceof msg2) {
            
        }
    } 
    if (st == nr3)
    {
        if (sig instanceof msg1) {
            
        }
        if (sig instanceof msg2) {
            
        }
    }
}
```

Figure 50 Generated code
As we see in the figure when a new state is added a dialog is presented where the user can choose between a number of fixed states, this insures that only suitable state names is selected. The code for the different choices can be found in already written components. The symbol representing a state is also drawn in a different way. The idea is that P is read Patient is followed by the state name. The different states could also have different symbols making the model even more visual and even more abstract.

On the tool tab we can see that there is no additional functionality beside the most basic ones known from UML2.0 State Machines. This in combination with predefined states and transitions makes the language much more resistible to miss use from the developer, and guides him in the right direction.

For setting up the explosion from the MOS_StaticStructure_Graph you must use the graph explosion definer. Add the object that you want to make an explosion for and then add the graph type. The dialog is shown in figure 52.
This is a very useful functionality that also is indispensable in the development of languages of some size. It can be used for making for instance composite structures and like this case defining an object’s state machine.

**Evaluation of the Meal Ordering System DSL creation process**

The process was very interesting and it provided lots of experiences that would have been difficult to capture from articles and literature on the subject matter, some pitfalls and its like was also discovered. The importance of planning the solution to some extent prior to implementation can not be underestimated. The fact that one version of the system already had been implemented in both UML2.0 and Java was an advantage, this version worked well and made it a little difficult to find alternative solutions in the DSL version. It probably would have given more to the process if some severe problems concerning the representation in the UML had occurred, but it did not.

**The development process: How was it?**

The goal was to develop some kind of Domain Specific Language for a Meal Ordering System family and get first hand experience on this. The first challenge was to read up on the theory and learning the tool. It took some time in the beginning but there were examples to look at. But it was difficult to actually get an idea of how the DSL could be made better than UML. The watch example that most of the tutorials is built up around is to me not very much more than some kind of state machine with different symbols. Instead of transitions trigger symbol there is the button, and the states are made with sharp corners instead of rounded ones. But this is of course a small example just used for demonstration purposes. There are also many other examples for instance the UML implementation. This one was very useful to look at and reuse concepts from.

If you are new to DSL creation, the development of the language can not be considered an easy task. There were some problems in the process. Many of these problems I believe had to do with that this was my first encounter with DSL development and there was not any finished framework that supported my wanted system family (only one finished version). There are quite many new things to
remember and it would be strongly recommended that the initial phase last for a certain time. Some time after the process was finished and the trial period of the tool had expired, a solution to my main problem in the development has occurred. I found it quite difficult to figure out how to model the dynamic aspects of the product line in a manner that made the different part consistent. Here especially the controller was difficult, for each variability point that was introduced in the other parts of the language the controller also needed additional behaviour. The solution that I think would have suited the language better is quite simple: removing all modelling functionality on the Controller and generated this part based on its connectable objects. In this way support for special messages and behaviour in for instance the Menu could have been generated in the Controller by traversing the Controllers relations including the connected objects properties.

Tool support
The decision to use the MetaCase tool was made early, this sounded like the best solution because it has been in use for many years. It does not target any specific platform or domain either which seems to be the case with Microsoft Visual Studio (C# and.Net). The tool provided great flexibility in the development of the different concepts and it did not restrain the development at any time. The report definition GUI was a little annoying in the start (seemed to lack some functionality). But there is possible to use another external editor and create your own template with different colours for different statements and so forth and just save your file with the right name. If you are used to tools like for instance Eclipse for writing your code, the editor has too little functionality. The menus for choosing predefined templates are very handy but not sufficient.

After the evaluation period of the tool was over it struck me that it might have been possible to use another editor within the MetaEdit+ environment but this was not tested. If this is possible it would be a big advantage. Then you can use whatever IDE you would prefer (for instance Eclipse). In addition writing a simple plug-in that supports the Report Definition Language would have been a great improvement.
Chapter 8

Examples of industrial DSLs

Introduction
In this chapter several Domain Specific languages that are used in the industry today are presented. And some aspects of why they are a success are presented. There are not very many examples that go deep into the languages available. But the reason is claimed to be that the firms that have successfully implemented them do not want to reveal their business advantages. There was however presented an article on SPLC 2005 [41] that listed lots of different domains where DSLs have been implemented successfully, and which approach that were taken in the development of the language.

Nokia Mobile Phone Software
In [42] a DSL developed and used by Nokia is described. The experiences and the results are presented. The DSL is made for development of mobile software that is used on Nokias mobile phones. The software inside the phones very often determines the winners. For the manufactures, time to market and quality is critical. Problem free and innovative solutions give good publicity which leads to good sales numbers. And if the company could release a product some weeks before the competitors this can translate to extra millions of dollars in income.

Therefore Nokia was looking for something that could potentially improve their productivity. Different Case tools were investigated but they did not provide what Nokia was looking for. The mappings from the tools seemed to be clumsy and to much of the domain specific understanding got lost on the way. The tools did not let them say all they wanted about their domain. Therefore they decided to undertake the development of their own CASE solution.

A decision to use MetaEdit+ was made and they hoped to increase the quality, decrease time to develop (TTD) a working system, encapsulate domain knowledge and obtain the possibility to work on a higher abstraction level.

According to [42] the results were ground breaking. In the early phases they noticed that an application that normally would take approximately 2 weeks to develop finished in only 1 day. Updating is quite normal in system development and this was no problem either. The MetaCase tool let them update the modelling language and even updated previously made models automatically for them. Changes to a traditional tool would have required programming and sharing of new versions.

They also reported up to 100 percent code generation in many cases. The coding was done in the design phase of the language and weren’t needed anymore.
Reports were also generated, this saved time and assured that the Nokia standard for documentation were followed, the reports was written only once in a correct manner.

Training costs and difficulties concerning the introduction of new team members also decreased. The language already included the domain concepts the developers were used to. And the learning curve of newcomers went from 6 month to 2 weeks. This was because the developers did not need to know or even look at the code; the method developed directed and enforced the development at a higher level.

**DSL for insurance products**

This DSL is a language used for defining insurance products. All the concepts used in the language are domain concepts that are familiar to the insurance expert. The expert on insurance can therefore draw different products and the code will be automatically generated from these. A system developer becomes redundant in the process.

![Figure 53 Modeling Insurance products that transforms to J2EE website](image)

This DSL is claimed to raise the abstraction far beyond programming and this again leads to that fact that it easily can be changed to generate some other output than J2EE websites. It is veritable a PIM in the sense of MDA, but there are no need to generate the PSM. In this case the users of the language where not supposed to be software developers themselves so special attention on the visualization was a priority. After using this language over 30 times it has proven to be over 3 times faster than writing Java directly.
**DSL for a Multi-tier Service-Oriented Application**

A commercial Customer Relationship Management application was due for reimplemention. The goal was to re-implement it as a Service-oriented, multi-tier application in C# on the Microsoft .NET platform. The developers decided to use a domain specific approach to both the external service-oriented interfaces and the management of transitions to the object-oriented internal implementation.

The reason for this approach was that the generic UML constructs were found to provide insufficient ability to capture enough semantics. Modeling a SOA architecture requires a structuring of components that are different from the one known from object oriented models. The approach in this example is not based on a new metamodel which is the case in the Nokia example. Here the UML is extended with a profile which incorporates the concepts needed to model the application in the desired way. In effect this is claimed to be a Domain Specific Language too.

The modelling is done in Rational XDE, but the code generator is substituted with a specific one. The generator used is a relatively generic one. It is based on substitution of text from a template file similar to the one used in the MOS DSL. The UML and Profile concepts reflect substitution tags which are used to substitute text in the template. In this way they are transforming the semantics of the model into executable code.

The available approaches were first investigated. Both existing UML and DSL approaches were found to be insufficient for the task, a large industrial project of this complexity needed something else.

The DSL has been successfully used on a prototype scale, and are now ready for full-scale development. The lower layers of the application such as Data Access Objects for individual tables were well suited for mass production based on a template. On the higher layers of the application the emphasis is more on standardization of naming and the generation of static structure.

It is clear that behaviour was difficult to generate automatically in the service layer less than 40% of the total amount of code is generated while in the Data Access layer approximately 80% code is generated.

**Enterprise Applications in smart-phones**

This DSL is another example where the MetaCase tool has been used. This language targets a well known domain, namely the mobile phone and its applications. The platform targeted is the Series 60 and an already existing framework that makes the development of enterprise applications possible. In this mobile phone domain there are special concepts like Note, Pop-up, SMS and Form. The language targets these concepts directly and enables the developer to work with these straight in the model. The different concepts have their own symbolic representation and the implementation details are hidden from the user. If the user knows how such
applications look like he would be able to create a system just by modelling the user interface. The model created does not have to be mapped to implementation concepts in code or UML. It will be generated based on the domain concepts. As we see from the figure the modeling language is quite domain specific. If you know mobile applications it is pretty easy to understand what kind of system this model shows.

The modeling tool also provides the user with guidance and prevents him from making illegal designs that is not supported by the platform or framework.

**Rhapsody DoDaf**

Rhapsody 6.1 is a new tool that offers both DSL and UML support. The main reason why this example is included is that it offers a predefined profile for modelling in a language called Department of Defence Architecture Framework. This example we can see shows clearer than the other that raising the abstraction with DSLs can be very beneficial.
It is possible for a DoDAF designer to specify types of DoDAF diagrams directly in the tool without any drawing of extra objects that are not supported in the language. A systems modeller can create for instance Requirements Diagrams, or Parametrics Diagrams to model performance constraints. In this way the creation of UML diagrams that might not be the most suited once can be substituted with models that better meet this specification. Looking at the figure it is obviously that this is a raise in abstraction from an alternative model in UML.

![Image of Rhapsody with DaDof profile](image)

**Figure 55 Rhapsody with DaDof profile**

**Others**

There are not many examples of working graphical domain specific languages that are described in articles or in other sources on the net. The reason for this is explained with the wishes of the companies to hold their business advantages secret.

This thesis is mainly about graphical domain specific languages, this is however not the only ones that exists. Probably the most successful DSLs known today are the small textual languages we can find in the Unix domain. LISP is on such language that is widely adopted. LISP stands for LISt Processing Programming and is a functional language based on John McCarthy's work on numerical computation, published in 1960. Lisp is probably the best example found today where the language is expressed directly in the language itself. Lisp has a minimalist syntax, closures and macros.

Other languages that can be mentioned are Lex, Yace, PostScript, AWK, SQL, etc.
Chapter 9
Discussion and conclusion

Introduction
We have in the previous chapters been introduced to and seen a description of an ordering system for the healthcare sector. The technology, application programming interfaces (API’s), a programming language and a framework suited for implementing UML2.0 StateMachines. Then two different approaches for how such systems can be developed were explained. The two modelling approaches have been the main focus and will also be the target of the discussion in this chapter but the other possible approaches will also be discussed.

I will first try to answer some general questions about the task of creating a DSL. Some additional approaches will be shed light on. Advantages and disadvantages of the different approaches will be presented. In this chapter when the term DSL is used, we are referring to graphical modelling languages that targets system development in a specific domain and are able to generate much of the code automatically from the domain concepts represented in the language. It is also different data that can be used for defining a DSL, but the discussion is mainly based on the approach where domain concepts, variability and commonalities are analysed.

There are lots of discussions on how modelling of software systems should be done. It has become a quite unison opinion that modelling is the solution and there are very few that contradicts this, but how it should be done is a big issue. As mentioned earlier the quest for raising the abstraction and thereby the productivity and the quality of the finished product has been going on for quite a while now. The discussions nowadays are heading more and more towards different approaches for how to define a DSL when this seems beneficial. It seems that there are many initiatives going one that are based on using the OMG standards as a base for creating Domain Specific Languages. The ability to use domain concepts in modelling is something that is pushed heavily from Microsoft’s Software Factories initiative, this might very well be a reason why the traditionally OMG supporters has thinking in the same direction. There are mainly three different foundations that can be used for defining a domain specific language.

The MetaCase and Microsoft’s Software Factories are two in the same category. Both MetaCase and Microsoft are basing their tool on a proprietary meta-language and not the OMG standards. The reason for this is because they mean that there are a lot of weaknesses in the OMG standards that makes them unfit for DSL creation and code generation. The other two are based on OMG standards. Before we explore these further we will look at some requirements connected to language creation and the use of modelling languages. The different approaches will then be discussed in detail.
What is required for developing a DSL?

Language development is not an easy task; there are lots of considerations and choices that must be taken. In different articles and web-logs [41, 43] it is claimed that this certainly is the case if you try to make a language that tries to fit all purposes. DSL is better in this way, because it only need to fit one specific domain. But anyway there are some comprehensive demands on the developers, their knowledge and the tools required.

Requirements that must be fulfilled if the process of developing a DSL is going to succeed is listed and explained in the next sections:

- Tool support
- Domain knowledge
- Experience from previous development of such a systems
- Framework (to some extent optional)
- Expertise in the DSL meta-modelling language and its like

Tool support

If a DSL creation process is going to be a success the most obviously requirement would be tool support. A tool that is capable of creating what the DSL makers want. The tool needs to be flexible in terms of method extensions, code generators and process support. It should be easy to create new concepts and add properties to them. One of the goals of DSLs are to raise the abstraction and therefore an integrated symbol editor will be preferred, at least the opportunity to import it from some other tool. But then it would be difficult to represent or show properties in the symbol itself. The example systems shown in chapter 8 have made the importance of some symbol editor connected to the tool clear.

The experiment has shown that it is beneficial to have the tool create the modelling user interface for you. And if you introduce a new concept this should automatically be updated in the graph editor. It would require a lot of extra work if this functionality is not offered. It is a great help in the development of the DSL, it makes you capable of testing the DSL as you develop it. Changes to the language will immediately be reflected in the case tool. Without this functionality the time needed would potentially increase a lot, which again can result in an unsuccessful attempt of reducing the overall development time of the systems.

In the tool from MetaCase this functionality is present, and works very well. It was extremely helpful to see the results and be able to test them right away. The tool is divided into two different parts Workbench and MetaEdit+. The Workbench supports the development of the domain specific meta-model that MetaEdit+ is able to reflect in its language specific modelling user interface with the specialized symbols easily available.
The concepts modelled are then directly mapped to your home-grown code generators. And this introduces another functionality that is needed or important to have. The ability to write the code generators in the tool is beneficial and eases the development process. In the MetaEdit Workbench there is a tool integrated for writing reports. Here you have access to template structures and also the different objects, properties and relations in your model. Writing the reports becomes much easier than what would have been the case if it was outside in a separate tool with no connection to the graph the generation is going to target. The fact that the building blocks of your model are available from the report definition tool can be seen as a vital part, especially if the model is large and complex. The templates available are as the modelling view automatically updated with the new concepts created in the workbench.

The tool did however not support syntax highlighting and auto-completion of reserved words. This is something that developers nowadays are used to from platforms like Eclipse and should have been included to ease the process. In the experiment this was one of the few times that the MetaCase tool did not offer all the functionality that was desirable. But this however does not qualify to be called a requirement the tool must fulfil, and the template functionality did overlap to some extent.

Both of these functionalities are something that also was mentioned to be on the high priority list in the DSL for a Multi-tier Service-Oriented Application project (described in chapter 8) where the absence of such support was said to be unnecessarily tedious[17].

**Domain knowledge**

The developers need great knowledge of the domain if the development and the result are going to be as wanted. This is part of the main point; the DSL is going to reflect the domain, so without detailed information or knowledge of the domain it will be extremely difficult if not impossible to make a successful DSL with the variability and commonalities approach. The amount of domain knowledge might be a problem in many cases. The days where every company had its own IT department is gone, often the development is outsourced then the requirement gathering becomes even more difficult. Communicating domain knowledge is a difficult task and the risk that the DSL won’t be suited for the task is absolutely present. This is in general difficulty in all software development independent of approach. But because there is a bigger investment in creating a DSL than just making a standalone system this becomes even more crucial. We can also see from the different success stories presented in chapter 8 that many are so called “in house” productions, the languages is developed by the company for the companies products (for instance Nokia [42]). This is also said to be the main target of DSLs in [44], where Dr. Steven Kelly CTO of MetaCase says:
“DSM requires domain expertise, a capability a company can achieve only when continuously working in the same problem domain. These are typically product or system development houses more than project houses”.

This is from an article published on the 3 July 2005 and it is the first occurrence of such a view found in the examined literature. Earlier papers on DSLs have not taken this stand, it has been the impression that a DSL was suited for every purposes. The new view probably relates to the research presented in [41] (Defining Domain-Specific Modelling languages to Automate Product Derivation: Collected Experiences) at SPLC 2005 where over 20 industrial DSL cases were examined.

Further on a more general basis the intended user of the language must be taken into consideration. And their modelling skills should be examined. One of the promises made by the advocates of DSL is that the abstraction will lead to that no or little modelling experience is required. If this is going to be the case the DSL must be created on the user’s terms, and are the users not familiar with traditional modelling techniques the demands for usability and vicinity to the domain is raised.

In the MOS experiment presented in this paper the intended users are system developers with some previous experience. The most advanced parts of the language are the MOS_StateMachine_Graphs and the simplest ones are the parts where property values are set from a fixed list (for example: type of menu selection dialog). Different qualifications are obviously needed for the different parts of the language and they also differing in the amount of variability. The more variability (number of potential system family members (NPS)) that are possible the more complicated the development and usage of the languages seems to become. Especially when the language constructs opens for extensions that were not planned in the variability space. The more specialised MOS_PatientState_Graph is somewhere between the restricted fixed list selection (menu object) and the more general purpose MOS_StateMachine_Graph.

**Experience from previous development of such a systems**

One important task is to map the language parts to working code. For this to be possible it is necessary or very beneficial for the developers to have done some development on the domain before. Ideally the code for the entire product family should already have been written and tested. Then the DSL could simply generate sequences of calls to the already existing code or assemble different variations of the components. This is also advised and claimed to be a good approach in articles explaining the process[25].

Regardless of when it happens the code needs to be written and assembled at some point. This means that knowledge of a language like UML is needed anyway, especially if the system to be developed is of some size. Developing large system without designing it first is difficult. And designing the code that is to be used in the DSL can not be done in the DSL that is not made yet. The documentation of the
underlying code must also exist if it shall be possible to update or extend the system. In the MOS experiment one such system existed, in an industrial setting this probably wouldn’t have been sufficient. The solutions that are selected must be well written and tested if not the new language could potentially result in mass production of bad code which contradicts the claim that a DSL improves the quality of the system. This is not an argument against making a DSL, the advocates of the DSL approach actually embraces the use of UML but only for sketching purposes. It is when it comes to generation of code and abstraction UML is claimed to be insufficient.

Standard notation is also useful to know when your own graph types are to be developed. It is not effective to reinvent the wheel all over again nor is it necessary. If a developer has modelled the task before, it is much easier to select the best modelling approaches. Have state machines been used before and found suitable for the task it would be advisable to use some kind of state machine in your DSL to. But potentially adding and / or removing some functionality and perhaps limit the potential names that are legal on a state or limit the number of legal transitions. The more you know about the potential family members the easier it is to make the language. A constraint on legal state names is made in the MOS experiment in the MOS_PatientState_Graph type. Once again the variability is reduced and the commonalities is increased the goal must be the perfect match between these two; enough variability to create the wanted systems but not to much as complexity increase both in development and the use of the language. However the more you use the language the more profitable it becomes. In the figure below this is shown.

![Figure 56 Product line cost estimate](image)

The same figure where total cost without product line and cost of product line were substituted with respectively complexity and number of supported variants could be used to show the impact of making a DSL to general.

**Framework (optional to some extent)**

The need for a framework is an optional requirement as stated in the headline. But if it is present it would ease the code generator development task and probably be very positive for the result. The created DSL will benefit from well proven code and the
code does not have to be written from scratch, development time decreases and the quality of the code increases (assuming that the code has been written by an expert). The framework can define “hotspots” of variability while the DSL provides you with functionality for setting the variability data.

A framework is a set of cooperating (abstract) classes for a given domain a software engineer can use to develop an application in that domain. A frameworks and a DSL have much in common. It has however become popular that frameworks are shipped together with a DSL to access the functionality available in the language.

It is a good idea to base the generation of code on a set of abstract java classes. The variability points is marked as abstract and must be overridden while the common structure for all the family members may be out of scope. This approach is obviously easier than the one without a framework and will most probably bring you a better result.

Frameworks and DSLs are quite similar in what they are aiming at. And have in many cases been used separately for serving the same purposes. But there are some mutual benefits achieved by combining them that is worth to mention. In [46] the following benefits are listed.

- It is a guide to the design of the framework. If there is no way to express a certain class or method as a language construct, it is likely that this class or method does not correspond to a natural concept of the domain.
- It encourages the development of black-box frameworks (based on composition) rather than white-box frameworks (based on inheritance).
- It gives more abstract access to the framework, hiding (encapsulating) what language is used to implement the framework.

In the development of the MOS DSL there was a framework available (JavaFrame) and it was used. But a lesson learned was that it would have been much more suited if it had been more “domain specific” to the MOS domain and not the concepts of UML 2.0 StateMachines and Composites Structures. The ideal would have been another framework built on top of the existing JavaFrame that related to the healthcare domain. The MOS DSL could then have accessed functionality in this framework. This can be seen in connection with too little experience from previous development of such a system (explained earlier). If the process was to be started all over again, much more focus on creating a healthcare framework would have been recommended.

**Expertise in the meta-modelling language and its like**

Expertise in the meta-modelling language used to develop the new domain specific language is necessary. If the language to be developed is of some size the functionality and possibilities must be known in advance. It would be possible to start developing and learn as you write, but this I believe and have experienced to some
extent from the Meal Ordering DSL creation is an disadvantage. If you know the meta language it becomes much easier to plan a quality DSL. However it is no problem to do several iterations. Updating, editing and deletion is not a problem, but it is time consuming and the chance of making severe mistakes increases.

Writing good transformers is not an easy task that either, and if you don’t know the transformation language well the solution chosen can quickly become more difficult than it actually needs to be. But the constructs of the languages are very often similar to scripting languages, and with some experience from programming or scripting the learning of the transformation language is manageable. This is the case with the Report Definition Language shipped with the tool from MetaCase. You have familiar commands like for instance: “foreach”, “dowhile”, “if then else” and so forth. In Visual Studio 2005 the language used for writing transformations is much like C# which is an advantage for people already familiar to this language or its likes (for instance Java).

What is required for using a modelling language?
This section lists some requirements that are needed for successfully using a modelling language. Both UML and different DSLs are discussed here as a whole. This is because there are much more similarities between them than there are differences. But to which extent the requirements are valid differs.

It would also differ from DSL to DSL but I will try to look at the general ones and when special considerations must be taken state that explicit. As an overview the following can be listed:

- Tool support
- Knowing the language and its related diagrams
- Domain knowledge
- The requirements

Tool support
A tool that makes you capable of using the language is needed if the positive effects of modelling are going to be fulfilled. The symbols and relations must be available in a tool whether the language is the UML or a DSL. One objective of model driven development is the automatic generation of code based on the models. This implies that the models must reside in a tool capable of handling this functionality or the tool must be able to exchange the data between those who can and not.

If you are using UML 2.0 it is no problem to find tool vendors that claims that their tool support the language. But as we have seen the UML language is big and not all legal UML features are possible to model for automatic code generation. But this will probably improve rapidly, the new specification has not been official for very long.
But we will probably not see a single tool that supports the entire UML 2.0 specification, to this it is way to comprehensive.

Most of the tools available support the creation of your own made code generators. Some are even shipped with the tool. RSA has for instance predefined transformations from UML to both Java and Enterprise JavaBeans. The Java transformation generates Java classes from UML classes. It was tested in the experiment and is to some extend helpful. When it comes to exchanging models this is normally well supported thru the use of XMI. In RSA models can easily be exported to files containing a XMI representation of the model.

With the DSL approach there are no predefined modelling tools (which are quite natural) because the languages differ. There is no use in developing your own symbols and relations if they are not presented to you in a modelling tool. If it is generated automatically or needs to be implemented in addition to the language itself, differs depending on the language workbench used for the DSL creation.

The model specific editors are also responsible for handling illegal actions made by the modeller and provide the user with appropriate error messages and guidance. There is no use in defining constraints if the tool does not reflect these restrictions. The tool must also implement the code generator. Without this functionality the language is more or less useless. When the model is made the tool must enable you to generate code directly, such that no extra mapping between the domain concepts and the implementation language is needed[42].

In the tool from MetaCase all this functionality is provided, which is an advantage. The modelling tool guides you to model legal variants based on constraints and legal relations. The usability is raised and less expertise in the art of modelling is required from the user.

**Knowing the language and its related diagrams**

This point might seem quite obviously, but the amount and what kind of knowledge differs from the two approaches. Using the UML requires knowledge of Object Oriented modelling but in the DSL approach this might not be needed. As we saw in the language for insurance products in chapter 8 only knowledge of the domain was required. The concepts and their symbols make the language self-explanatory to a user familiar with the insurance products. The language for enterprise applications from Nokia [42] is another example, it is evidently that languages can be made pretty much self-explanatory for a user familiar with the domain.

But there are also examples on DSL that requires modelling expertise similar to UML. Many of the examples seen that is used for embedded systems utilizes state machine concepts but with some additional functionality and domain like symbols (for instance the watch example). This does not necessarily make the languages easier to understand and use. But in this relation it is important to separate the domain users
from the modellers. If the language is intended for the first category there is probably only one language to learn, and then the DSL with less functionality and domain concepts will be adopted easier. Are the intended users modellers that participate in many different project this might not be the case. In [16] it is claimed that a well known language like UML has well-established tools and the community knows how to use it. With proprietary languages there is a higher learning curve, and the adoption is more difficult, this seems reasonable.

We have seen that it is possible to make the UML more domain specific but to make it self-explanatory is however difficult to accomplish. A profile with stereotypes and specialised symbols can be made, but still the modelling tool will not restrict the user from drawing something meaningless or use other relations available in the tool. The DSL reflects the meta language for the targeted domain, and illegal models are not possible to produce. This is one of the major advantages of using a well formed DSL compared to the UML. If you are going to use UML you will have to know the different diagrams much better than what is the case with a DSL, this comes on top of knowing the domain or the product you are supposed to model. The restrictions are a positive thing to the user of the language, but it can be a disadvantage in some cases too. If the language is not capable of representing what the user wants this requires modifications of the language itself.

And it is not necessarily vital that all the different diagrams in UML are familiar, but there is no tool that tells you what to use. And lots of different concepts are available.

In the MOS example system there is for instance not necessary to have knowledge of any communication other than asynchrony in the sequence diagrams. In the DSL version synchrony communication was not included and is therefore impossible for the user to apply to a sequence diagram. In UML however this is possible; there is nothing that tells the modeller that this is an illegal action in the system family.

The diagram types Object Diagram, Component Diagram, Package Diagram, Deployment Diagram, Activity Diagram Communication Diagram and Timing Diagram are not used either. It can be claimed that you only have to know those diagrams you need. On the other side this might not be the case for all situations, because this requires that the diagrams to use already are chosen. If you are supposed to develop a system you will have to know the diagrams if not you can not pick the ones suited for your assignment. In the DSL version only the diagrams needed are available.

**Domain knowledge**

A Domain Specific Language uses domain concepts as constructs. Knowledge of the meaning of these is therefore obviously necessary. It would not be possible to model a system in the language if you do not have a clear meaning of what you want to model and what your model exactly means. This underlines the importance of
communicating domain knowledge between the users and the developers in the construction phase of the language such that they agree upon the meaning of them. This also counts in favour of the “in house” approach, where both the creators and the users of the language are the same persons or at least share common knowledge of the targeted domain.

This is one of the big advantages of DSLs. Domain knowledge that is common for all systems is encapsulated in the language and hidden from the user. In the MOS DSL this is also the case. The controller object encapsulates all the details of sessions, how they are generated and how they function.

In UML much more is up to the modeller. There are no guidelines implemented in the tool and no error messages generated if a system model that is ill defined is created. Encapsulation can be achieved by the use of components but also this approach could potentially cause a bad model.

The requirements
As with all software development the requirement is important to capture also in these approaches. In DSLs there are guidelines in the language so it is not as difficult as if you were to model the system from scratch. The domain concepts make it easier to communicate requirements as long as both the developer and the customer know and share the same meaning of the concepts involved. If the tool provides you with 100 percent code generation it probably would not be that much work to do some changes though.

And one of the advantages of making a DSL is that the language could be so self-explanatory that the users themselves should be able to make systems based on their needs. The insurance DSL shown in chapter 8 figure 53 is a good example of such a domain specific language.

In UML it is more difficult to discuss requirements on a detailed level. Use Case diagrams are normally used when the customer has little or no knowledge of UML. Presenting class diagrams with different associations are not as “customer friendly” as the domain concepts and familiar symbols.

What approach to choose?
We have in the previous seen that there are numerous of different approaches that can be chosen among when a basis for system development is to be chosen. The figure below shows the initial phase (1) and some of the different choices that can be made. We have seen that there are possible to use both standard languages from OMG and proprietary languages in the development process. All the different approaches have their pros and cons. We have already seen and explained some of them, here they will
be discussed in the context of what approach to choose and in more detail. When are the different approaches most likely able to deliver a better result than the others?

Figure 56 shows an overview of the approaches that have been explained and discussed in this thesis. It is not however complete, there are other tool vendors available but they have not been the target in this paper and therefore omitted in the figure. The steps in the figure do not have to follow a sequential trace i.e. it is legal (and make sense) that all different alternatives can be chosen on the levels (backwards traversing). And in the ideal process all the traces should be investigated if necessary.

OR-gate.1 symbolises the choice between a standard and a proprietary language. This decision has a major impact. On a day to day basis it is not many that actually are thinking of the different standard around us. Almost wherever we go we will most probably be interacting with a standard or some equipment that is. This could be the GSM network, the electricity connection in your office, your web browser and so forth. The standards are there for a reason. They are made and agreed upon so that it will make our daily life simpler and less expensive.[47]

Figure 57 Overview of potential approaches (gray shaded tested in thesis experiment)
This is also the case in the world of computer system engineering. If for instance a modelling language is based upon an open standard it would be possible for different companies to make tool support. The customers will then have access to less expensive tools because of the competition. There will be no problems involved for a company to change from Rationale products to a tool from a competitive company. The models can be exported and imported between the different tools.

This again will lead to increased competition among the tool providers so that their tool always will be up to date. They must also be careful in the pricing policies, if the product is too expensive a customer could choose some other vendor. If the standard is adopted by many different companies there will also be an increased research on how to make the products even better, which again will have a positive effect.

When it comes to human knowledge this will also increase, there are one standard to learn and not tens of different ones. If a company hires a new employee it is more likely that he or she will have knowledge about for instance the UML than a proprietary language used by a few. According to [16] there is also a higher learning curve for new proprietary languages and they require more effort to adopt. This should speak in favour of the standardized approach.

But choosing a standard often means that the language is more comprehensive. It is build to support a wider range of usages and this is not all positive. The learning process of UML is far from short. According to [48] expertise in the field of object oriented modelling is not accomplished before several years of work in the field and it is not easy to identify objects. Entities of the problem are identified as objects, but it does not necessarily mean that they are useful in the design. Choosing to make a DSL can in some cases help avoiding this. A proprietary language is very often smaller than UML this could be beneficial.

The fact that a language is a standard does not necessarily mean that it support solving all kinds of problems in it or that it is best suited. And this is a reason for adopting a proprietary language (or creating one) that is specialised for the task.

We now move on down to the second level of figure 56. Here we see two different connections from OR-gate.1.1 and three from OR-gate.2.1. The left most alternative 1.1.1 (UML version based on a proprietary metamodel) is not very normal to adopt. The reason why it is included is that tool vendors that supplies DSL tools often have some kind of implementation of the UML based on their own metamodel. This is also the case in MetaEdit4.0 and it can be used as a basis. If you are dependent on for example a sequence diagram editor that has support for gates, lifeline decomposition and inner connectable elements this might be a solution. The support for inner connectable elements is rarely implemented in standard UML tools and we have also seen that a tool vendor of IBM’s size does not support all this functionality. It would be fairly simple to include this in the MetaCase tool. But an alternative like for instance Visio probably would have been both less expensive and easier, especially if nothing was to be generated from the sequence diagram.
We now move our attention to the right most approach (2.1.3) standard UML. The most obviously reason for using the UML language for system modelling is that it is a heavily adopted standard that have been around for a long time. It has proven itself useful in many different kind of system development projects and is ready to use as it is. Its expression power is evident and it has a lot of predefined diagrams for suited for describing static structures as well as interactions or behaviour. This can however easily speak in UML's disfavour to.

If you are a “Subject Matter Expert” or unfamiliar to UML and maybe has spent 20 years working in a field, the UML might seem like a foreign and very confusing language. In this case the solution could be a more restricted and “simpler to use” DSL that fits the known domain better. Introducing UML in this case would probably defeat its own end and slow down the process rather than speeding it up.

There are opponents that claim that the UML language is bad suited based on its history. This mainly is in the combination with code generation. They argue that there is a need of knowing what to do before you do it. And this has not been the case with UML. The standards that MDA is built upon were not made with the intention to support automatic code generation and therefore they claim that new languages must be defined on all levels. It is way too complicated but still it does not make it possible to solve every problem that the software developers nowadays require. UML was well suited for the task of sketching but not for automation and abstraction elevation.

In the book Software Factories[27] its even stated that it is impossible to specify the exact meaning of any UML model without additional information. As an example the black diamond relation is used to exemplify this. Questions like: Does it imply deletion propagation? Can a part change its whole? If so when? But again providing additional information should not involve more work than building a completely new language. OCL is par of the UML2.0 specification and should be well suited for the assignment.

Further the example of UMLs problems with modelling Java and C# interfaces is used as an illustration. It is claimed that this is a big issue, but later on it says that the difficulties with the Java interfaces are overcome in the new version because of the ability for UML interfaces to have attributes. The C# problem is however still unsolved. That the Java interface problem is solved shows that it is possible to correct weaknesses.

One of the reasons why the work with the new 2.0 version was initiated was to better this support. And it has according to [10]. The opponent however maintains their position based on the fact that the foundation still is the MOF meta-language and it is in this layer the problem lies. MOF is not used in the Software Factory because there are three different versions, the original and two competing versions known as EMOF and CMOF. The different camps that provide implementations (Java and CORBA) can not reach an agreement this puts the value of MOF in questions. The goal of
meta-language technologies is to provide language definition that can be interpreted or compiled such that they can produce editors and other tools for those languages. MOF covers only a small part of the tool design space. Experience is also lacking in the area of language and tool definition.[27]

Moving to 2.1.2 in the figure we find other standard languages based on MOF. This approach is not discussed in any detail, but as an example the Common WareHouse Model (CWM) can be mentioned:

“The Common Warehouse Metamodel (CWM™) is a specification that describes metadata interchange among data warehousing, business intelligence, knowledge management and portal technologies. The OMG Meta-Object Facility (MOF™) bridges the gap between dissimilar meta-models by providing a common basis for meta-models. If two different meta-models are both MOF-conformant, then models based on them can reside in the same repository.” [10]

Four approaches are now left unexplained. Firstly we take a look at the approach making use of UML’s extension mechanism profiles (described in chapter 2). This is not a first-class mechanism i.e.; it does not enable the modification of existing metamodels. This is of course something that makes it less powerful. But it is the intention to provide a straightforward mechanism for adapting an already existing metamodel and provide additional constructs that are specific for a particular domain.

The opponents of this approach claims that it is ill suited because it is not possible to remove any constraints that already are specified in the metamodel and this makes it bad suited for DSL creation. This is true to some extent, if the language to be created can’t be built on this metamodel in a reasonable way, another approach would be desirable. This will however include much more work; many of the restrictions must be built all over again. And if you only use the profiling mechanism the interoperability between different tools are conserved which is a big advantage. New constraints can however be added as you like.

If you need a first-class mechanism (the UML profiles are not sufficient) then the OMG advocates suggests that the extensibility is handled through MOF based DSLs(2.1.1.1). Using this approach let you modify whatever you would like. There are no restrictions on what is possible and allowed to do with a metamodel: meta classes and relationships can be added or removed according to your requirements. Methodology restrictions can however be imposed that tightens the gap between profiles and metamodel editing, for instance restrict the modification to only extensions.

But there are also drawbacks associated with this approach. The MOF only supports binary associations and not higher order 'N-ary' associations. This trade-off was made because N-ary relationships are claimed to be rarely used in metamodelling and the design goal was to keep the MOF interfaces simpler. It is anticipated that the MOF will be extended to support higher order associations in future.[14] That 'N-ary' associations are rarely used is however not the opinion of the people behind MetaEdit.
and Software Factories. In [27] this lacking support is proclaimed to be one of the reasons why Microsoft does not put their efforts into building their tools on OMG standards.

The MOF approach is very much similar to the ones from MetaCase and Microsoft (spanning from OR-gate 1.1.2) where the construction of the new language is done by defining a new metamodel for the purpose. Some of the difference lies in what kind of meta-metamodel they use and as we have seen; what it is capable of representing. MetaCase’s meta-metamodel is the GOPPRR (described in chapter 3). MetaEdit+ is however the most mature solution considering tool support. MetaCase have delivered tools that helps their customers improving their productivity and competitiveness thru customized development methods since 1991 and it has been applied in more than 30 countries.[25] That it is extensively used is a advantageous compared to other similar solutions like for instance the tools from Microsoft that just recently has been released in beta version 2 (September 2005) [49]. The same goes for MOF customising tools, there are not many available but OptimalJ from Compuware has some functionality for editing on the MOF layer but it is mainly used as a support tool for OMGs MDA approach and generates J2EE based on PIMs.[50]

The DSL approaches is well suited if you want to specify a terminology that adapts to a particular domain or make your own defined syntax. In UML there are several semantics that is unspecified, for example how to deal with priority when a StateMachine receives a signal, or you want to add a syntax for a construct for instance in the case of actions. Specialising UML state machines requires knowledge of the metamodel. My claim is that this is done easier in a DSL made with the MetaCase tool. [3, 14, 39]

Another thing that speaks in favour of the proprietary approach taken by MetaCase is that Language workbenches can be too immature for standards yet; the risk of premature standardisation is absolutely present. Premature standardisation can lead to bad solutions, (the field is not sufficiently explored yet). But Microsoft seems to build their tool on a metamodel that is quite similar to the already existing ones from OMG, with some exceptions like the “n-ary” mentioned above.

But one of the main reasons and assumption for choosing a DSL approach is that it is possible to generate several different systems (that share some common structure or behaviour) from the language. If the effort invested in creating the language shall pay of this is crucial. The name of Microsoft’s initiative speaks clear about the objective: creating software in a factory manner. In this way producing software much more cost efficient than the situation is today. That this approach is possible to gain positive results from are however not a unison view in the software modelling community.

In [51] Grady Booch agrees with Tom Demarco on his statements in his book “Why Does Software Cost So Much?” Where he says that:
“Factory methods for software are dead wrong, witless, and counter-effective. Organizations that build good software know that software is an R&D activity, not a production activity. Companies that try to develop software in a mass production manner will produce bad software”.

This is in sharp contrast to the vision of domain specific languages and specially the name of Microsoft’s initiative “Software Factories”. The number of potential systems that can be developed from a language has to exceed a certain number. In a presentation on OOPSLA03 [52] the number was claimed to be 3. This is not something that is specific to DSLs but to product line approaches, but it would anyway be reasonable to think that this would differ from language to language depending on the amount of work that is needed in the development of the specific language and other factors involved like for instance marked benefits of creating software faster or the knowledge of the intended users (do they know modelling up front?).

A subject that is closely related to the one just discussed is the productivity promises made by the DSL supporters. In [20] it is claimed that DSL are capable of increasing the productivity by a factor of 5-10. This number however is not based on any statistically research. There was an experiment done by the US Air Force on their message specification system C3I where over 130 developing tasks both initial and maintenance were studied. The conclusion here was 3 times better productivity than if code reuse was the approach. Worth noticing is that code reuse are not necessarily the alternative anymore. Many of the other numbers presented in literature seems to be on qualitative measurements. And it is not stated what is included in the data: is the development time of the DSL included, was there any code available and so forth. And the results can also come from the introduction of a product line approach that is not specific for a DSL. Product lines are also possible to model in UML2.0. [53].

<table>
<thead>
<tr>
<th>Ability</th>
<th>MOF-based Metamodelling</th>
<th>UML Profiles</th>
<th>DSL and MetaEdit+</th>
</tr>
</thead>
<tbody>
<tr>
<td>New attributes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>New default data type instances</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>New Associations</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>New methods</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>New types</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Adding subtypes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Deleting / Renaming types</td>
<td>No</td>
<td>No</td>
<td>Not AbleToA</td>
</tr>
<tr>
<td>Migration</td>
<td>More difficult</td>
<td>Less difficult</td>
<td>More difficult</td>
</tr>
<tr>
<td>Automatic generation of tool</td>
<td>Kind of</td>
<td>Kind of</td>
<td>Yes</td>
</tr>
<tr>
<td>Suited for code generation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1 MOF, UML and DSL capabilities (Parts of MOF and UML columns gathered from [1])
In table 1 several different abilities for the three different DSL approaches explained in the previous sections are systemised. Part of the table is collected from [1] but is extended with the DSL section and the “Automatic generation of tool” row.

Here the similarities between the MetaCase approach and the MOF based approach is clear. Both offers much more power than UML Profiles, but it’s evident that for instance changing tool vendor is more difficult with the metamodelling approaches than it is if the UML profiles approach is chosen. The modelling tool is automatically generated if the MetaCase tool is used. This is to some extent also true with the use of profiles, but this support is not as good. It is possible to apply stereotypes from a profile in the UML tools, but this comes in addition to the existing functionality that is visible. The Rhapsody DODAF example in chapter 8 is however based upon profiles, but still making the UI this domain specific belongs to the exceptions with this approach, not many tool vendors provides this support.
Conclusion

In this section I will try to answer the main questions of this master thesis based on my experiment, the previous chapters and literature found. They are presented as short, concise statements of the inferences that I have made based on the work. As the title of the thesis shows this has been an exploration and comparison of different approaches for system development.

1. It has been proved that all the different approaches explored has their benefits and disadvantages. None of them are best suited for all possible development scenarios; there is no universal best approach. Factors that influence the result are among others: number of potential different systems, available artefacts prior to development, expertise available in the target domain, amount and what kind of system development expertise is available, the intended users of the language and what kind of system is to be developed.

2. The DSL approach can be a good choice especially if it is a so called “in house production” but are not equally suited for “project houses”. This underlines the importance of special expertise. Several examples from the mobile phone industry have proven effective power. Other characteristics of these systems are their size and the importance of making systems quickly and almost on demand. There are some potential risks involved when developing your own language but they can be minimised by strictly analysing the pros and cons.

3. UML has proven useful in many industrial projects and the support for MDA is getting better. The JavaFrame compiler has shown that it is possible to generate much more than static structures from simple class diagrams. The UML Model Transformation tool (UMT) has shown good results in model to model, text to model and model to text transformations.

4. If the DSL approach is chosen (whether based on OMG or proprietary), the tools available are increasing. Present the proprietary tools seem to have the best support and MetaCase are a big contributor and are up front, but there are examples of available OMG supporting tools for instance Rapshody. This tool has however a predefined DSL included and it seems like better languages workbenches for UML is desirable. Microsoft’s tools are not yet released in its final version, but I presume this will be a quality DSL environment.

5. We have seen that it is possible to raise the abstraction by using domain specific languages. The languages can make non developers capable of creating their own systems. UML are not providing this in the same manner, but it is possible to achieve this to some extent with the use of the profile mechanism and potentially creating a computational independent model (CIM).
6. By raising the abstraction it is shown that systems in a product line can be developed faster than what is the case with alternative approaches. It is however very little documentation on when the timing started and what was available up front. There are known benefits of choosing a product line approach so it becomes difficult to estimate how much of the gains are because of abstraction raise and what is coming from the product line approach (every DSL is a product line, union of all potential variants).

7. The experiment has given indications that contradict many of the statements made about how simple it is to develop a domain specific language. My conclusion is that in such a domain and suchlike, the creation is not simple. This is also discussed on pages 77-81. The more variability that is supported in the language the more difficult it becomes to develop, but the potential profit increases at the same time.

In general it becomes a comparison of positive and negative effects that is valid for exactly the scenario targeted. More material on such comparisons would be beneficial and will probably appear as graphical domain specific languages become more widespread.

If we agree upon the importance of standards there is no doubt that the work done by OMG and its participants must carry on and are extremely important. I see the proprietary solutions in addition to the standards and not as a competitor; this work is also important and beneficial. It becomes much like the utility value of an adjustable spanner (UML) that is well suited for many different purposes, but in some occasions the need for a specialised fixed spanner (DSL) is required and might even be irreplaceable. One single spanner for each different nut requires more investments, but the advantage of having it potentially overcomes this. In the cases where both can be used but with different advantages, the pros and cons must be considered for each separate case.

The history of UML also speaks in favour of this; it is a combination of many different approaches that have been used the last decades (MSC, D.Harel’s state charts and so forth). Probably many of the DSL features will be included in UML in the future. And the proprietary approaches will absolutely contribute to this. The effort that is put into the research on making MDA based DSLs would probably not been such a big research field as it is today if it has not been for the competition from proprietary DSL vendors.

As a final remark it can be said that it seems like the proprietary approaches are more marked oriented then the approaches form OMG. OMG initiatives seems to have a longer time before the efforts of investments are paid of, while companies like for instance MetaCase are more into realising productivity gains now and not in five years, even though the solutions based on proprietary metamodels might not be the best solution for the future it is needed as an intermediate solution until the MDA
approach has matured. The MDA is trying to span more ground and is therefore naturally using more time to accomplish its vision.

**Contributions**

I have in this thesis shown that it is possible to use different approaches for developing a Mobile Meal Ordering System for the healthcare sector. The reasons why the different approaches should be chosen are described. Different literature on the subject have been presented and discussed.

The claim that DSL development is an easy task has been discussed and not found to be the case in many situations.

**Future work**

It has been shown that the different approaches have their strengths and weaknesses. Using proprietary languages is not good for many reasons in light of the benefits of a standard. The standards have some shortcomings when it comes to tool support and there might also be some ill suited parts in the languages (MOF to difficult and lacking support for “n-ary” relations, UML not clear in its interpretations of i.e. composition association). There are many different opinions. For the future it would be desirable to agree on a potentially Unified Metamodeling Language (UMML) that are better suited than MOF, simpler but still powerful. And work concerning this would be desirable.

Secondly there is no doubt that both the proprietary languages and the standards from OMG will be around for quite some time. An interesting thing to investigate and suggest a solution for; is a reference model to assist in the choosing between the different alternatives available. More statistically material than it is available today will be preferable and will most likely appear in the near future.
References

34. ParlayOrg, *The parlay community home page*.

96


List of Figures

Figure 1 Association in UML ................................................................. 14
Figure 2 Class with internal structure Part, Port and Connector .................. 14
Figure 3 Sequence Diagram with [alt] and [opt] ........................................ 16
Figure 4 Simple State Machine Diagram .................................................. 17
Figure 5 State Machine Diagram with substate, exit and entry point ........... 18
Figure 6 Internal view of State processRequest .......................................... 18
Figure 7 Stereotyped Class ..................................................................... 20
Figure 8 The Layered Architecture ............................................................... 21
Figure 9 Three Major steps in ................................................................... 22
Figure 10 Moving from Domain idea to finished product [20] ....................... 24
Figure 11 Parts of a graph in MetaEdit [24] ................................................. 27
Figure 12 Properties and non-properties description [24] ............................. 27
Figure 13 MetaEdit+ Workbench ............................................................... 28
Figure 14 MetaEdit+ Object tool ................................................................. 29
Figure 15 MetaEdit+ Graph tool ................................................................. 29
Figure 16 MetaEdit+ Property tool ............................................................. 30
Figure 17 MetaEdit+ Watch Example [26] ................................................... 31
Figure 18 The Parlay OSA API overview .................................................... 35
Figure 19 Sale of GSM-services. Millions kr ekskl. taxes ............................. 36
Figure 20 JavaFrame overview [38] ............................................................. 37
Figure 21 Some Use Cases from The Meal Ordering System ......................... 42
Figure 22 MOSystem Composite Structure ................................................ 43
Figure 23 Composite Structure Diagram Controller ..................................... 44
Figure 24 Sequence Diagram with reference occurrences ............................ 45
Figure 25 The MealOrder referenced in overview ......................................... 46
Figure 26 Decomposition of the MOS lifeline .............................................. 47
Figure 27 State Machine Showing the Menu states and transitions ............... 48
Figure 28 The Update Menu Activity .......................................................... 48
Figure 29 The Session Controller StateMachine .......................................... 49
Figure 30 Problems with gates in RSA ....................................................... 50
Figure 31 MOS_StaticStructure_Graph ...................................................... 54
Figure 32 Error message box ..................................................................... 55
Figure 33 Connectivity Constraint Definer for OrderStore ......................... 55
Figure 34 Graph tab ............................................................................... 56
Figure 35 Graph type definer for MOS_StaticStructure_Graph ................. 56
Figure 36 Menu object ............................................................................ 57
Figure 37 Menu object properties selection ................................................. 57
Figure 38 Code Generator report for Menu object ...................................... 58
Figure 39 Order history property dialog ...................................................... 59
Figure 40 PatientStore with property dialog ................................................. 59
Figure 41 MOS_Sequence_Graph GetMenu ............................................... 60
Figure 42 Graph binding definer for MOS_SequenceGraph ........................................ 61
Figure 43 MOS_Sequence_Graph ViewOrders.......................................................... 61
Figure 44 Message property dialog box................................................................. 62
Figure 45 JavaFrame Message generation............................................................... 62
Figure 46 Relation Asynchronous Message .......................................................... 63
Figure 47 Generated Code .................................................................................... 63
Figure 48 MOS_StateMachine_Graph with few restrictions................................. 64
Figure 49 Code generator MOS_StateMachine_Graph.......................................... 65
Figure 50 Generated code .................................................................................... 66
Figure 51 MOS_PatientState_Graph .................................................................... 67
Figure 52 Graph Explosion definer....................................................................... 68
Figure 53 Modeling Insurance products that transforms to J2EE website ............... 71
Figure 54 Enterprise apps in smart-phones (dsmforum.com)................................. 73
Figure 55 Rhapsody with DaDof profile ............................................................... 74
Figure 56 Product line cost estimate[45] ................................................................. 79
Figure 57 Overview of potential approaches (gray shaded tested in thesis experiment) ........................................................................................................ 85
Glossary

C#  A C++ derivative programming language that is similar in functionality, look, and feel to Java

CIM  (Computational independent model), the base PIM that expresses only business functionality and behaviour.

COBOL  One of the primary business programming languages.

Connector  An UML relation used to connect two parts of a system

CWM  (Common Warehouse Model) A language based on MOF

DAO  (Data Access Objects) Objects specialized on accessing data.

DSL  (Domain Specific Language) A language that models domain concepts and is capable of generating code directly from these.

EJB  (Enterprise Java Beans) The J2EE middle tier infrastructure designed to support business components.

GOPPRR  (Graph, Object, Property, Port, Relationship, Role) The MetaEdit+ metamodel concepts.

Graph  A collection of concepts and how they are connected

HTML  The industry standard for describing browser display

HTTP  The industry standard for communication over the Internet

J2EE  A platform-independent, Java-centric environment for building enterprise applications.

JAVA  Object oriented programming language from SUN

MDA  (Model Driven Architecture) An approach for how to specify Software systems, the different models and the transformations between them. An OMG initiative.

MDD  (Model Driven Development) Used on the approach that promotes modeling as the basis for all development, much similar to MDA but do not restrict to OMG standards.
**Metamodel** Metamodel is a model that describes the building blocks of a model on a higher layer.

**MOF** *(Meta Object Facility)* A meta metamodel that is part of the OMG standardisation work. Used to describe the UML metamodel.

**MSC** *(Message Sequence Charts)* Originally used to specify telecom services. UML 2.0 has adopted lots of the functionality concerning structuring mechanisms.

**OCL** *(Object Constraint Language)* A formal language for describing constraints on UML model elements.

**OMG** *(Object Management Group)* Industry consortium which goal is to provide a common framework for developing object oriented programming techniques.

**Parlay** API that enables access to the telecom-network.

**Parlay-X** Simplified access to the telecom-network thru the use of web-services.

**PIM** *(Platform Independent Model)* A model of a subsystem that contains no information specific to the platform or the technology that is used to realize it.

**PSM** *(Platform Specific Model)* Same as a PIM but here the platform specific information is included.

**RDL** *(Report Definition Language)* The language used for writing reports in the MetaEdit+ tool.

**SDL** *(Specification and Description Language)* One of the first languages that made it possible to specifying systems as a set of interconnected blocks (Composites).

**UML** *(Unified Modeling Language)* The Unified modelling language from OMG. Used to specify the structure and behaviour of systems.

**UML Profile** A definition of a set of stereotypes, tagged values and constraints that extends the elements of the UML metamodel.

**XMI** *(XML Metadata Interchange)* An OMG standard that makes it possible to interchange of different models via XML documents.

**XML** *(Extensible Markup Language)* Industry standard for the definition, transmission, validation and interpretation of data.
Appendix A
Use-Case Model

Meal Order System

- ShowDishInformation
- MealOffer
- DisplayMealHistory
- Register Personal Information

Roles:
- Patient
- KitchenStaff
- HealthcarePersonel
Class Diagram
Composite Structure Models

MOSystem

PatientStore

OrderStore

Controller
Interaction overview Diagrams
Sequence Diagrams

ServiceUser

RegisterInformation

MealOrder

DisplayMealHistory

ShowDishInformation
A potential update or initial registration of personal information or needs. N = name, a = age s = size, sn = special needs.
Sms("Hello Gøran, todays meal: Smoked tuna with rice. Drink: Sparkling Water. Size: Medium  1=Ok  2=Menu  3=Update[params])

alt

Sms("1")

Sms("Order Made: Meal: Smoked tuna with rice. Drink: Sparkling Water. Size: Medium")

Sms("2")

Sms("1. Smoked tuna with rice, 2. Lamb chops with potatoes, 3. Pork chops with french fries")

Sms("3")

Sms("Order Made: Meal: Pork chops with french fries. Drink: Sparkling Water. Size: Medium")

Sms("3 Drink Water Size Small")

Sms("Order Made: Meal: Pork chops with french fries. Drink: Water. Size: Small")
MealOrder

:ServiceUser

MOS
Ref:MOS, MealOrder(3)

opt

mealOrder(pN)

Time: ordertime

Find preferred meal, size and drink

mealOffer(Fish and chips, water, medium)

mealOffer(meals, drinks)

Check order

mealOrder(Meat Balls, soda, large)

mealOrder(ok)

mealOrder(ok, milk, small)

mealOrder(nok, displayMenu)

Retrive todays menu

orderConfirmation(Meat Balls, soda, large)

orderConfirmation(Meat Balls, soda, small, you are too fat)
Comparison of different MDD approaches

Master Thesis
State Machine Diagrams

**Sm Patient**

- **Idle** state
  - Transition to NeedDinner with condition: (Time = dinnerTime - 3) \(\rightarrow\) needDinner(pNr, drink, size)
  - Transition to confirmation/confirmed

- **NeedDinner** state
  - Transition to Idle

- **Sm Patient** creation

**Sm Menu**

- **Idle** state
  - Transition to \(\langle\text{create}\rangle\) 

- **Sm Menu** creation

**Sm order**

- **suggested** state
  - Transition to confirmed
  - Transition to updateOrder(pNr, Meat Balls, soda, large)

- **confirmed** state
  - Transition to \(\langle\text{delete}\rangle\text{FromSystem}\)

- **Sm order** creation

**Sm SessionController**

- **Waiting for meal proposal**
  - Transition to newOrderSession(pNr, drink, size) \(\rightarrow\) getMeal(default, drink)

- **Waiting for Order**
  - Transition to mealOffer(meal, drink) \(\rightarrow\) makeOrder(pNr, mNr, dNr, size) \(\rightarrow\) mealOffer(Meal, Drink, medium)

- **Order Made**
  - Transition to mealOrder(pNr, ok) \(\rightarrow\) confirmOrder(pNr) \(\rightarrow\) orderMade(mNr, size, dNr)

- **Timer > 5min**
Idle

Waiting for patient info

Waiting for menu

Waiting for confirmation

Meal order (m,d,s): sms("Order made")

SMS(@1): sms("Order made")

SMS(@3 d s): sms("Order Made")

SMS(@2): sms("Today's Meal offer")

SMS(@1): sms("Today's Meal offer")

SMS(@2): sms("Today's Meal offer")

SMS(@1): sms("Today's Meal offer")

SMS(@2): sms("Today's Meal offer")

SMS(@1): sms("Today's Meal offer")

SMS(@2): sms("Today's Meal offer")

SMS(@1): sms("Today's Meal offer")

SMS(@2): sms("Today's Meal offer")
Appendix B
MOSysMain.java
/*
* Created on 29.mar.2005
* Göran
*/
package src;

/***
* @author Göran
*/
import se.ericsson.eto.norarc.javaframe.*;
import msg.*;
/***
* Purpose: define a class for the main program.
*/

public class MOSysMain
{
    private static String userName = "gorano";
    private static String password = "gorano";
    private static String path = "http://telenor-gw.ifi.uio.no:8181/pats/MessageSender?connectionId=stud1";
    private static String inpath = "http://telenor-gw.ifi.uio.no:8181/pats/MessageChecker";
    /////Mediator toMos;
    SmsInputMediator toMos;
    //////EdgeOutMOS MosToEnv;
    SmsOutputMediator MosToEnv;
    MOSys nOSys;
    //Environment env;//for env imp
    /**
     * The constructor for the main program.
     */
    public MOSysMain()
    {
        Scheduler scheduler = new Scheduler();
        Thread t1 = new Thread(scheduler);

        //Create the edge mediators
        /////toMos = new Mediator();//for env imp
        //Mobile receiveing
toMos = new SmsInputMediator(inpath,userName,password,4000);

        //Mobile sms sending
        MosToEnv = new SmsOutputMediator(path,userName,password);

        //GUI sending
        //MosToEnv = new EdgeOutMOS(this);//for env imp
    }
}
//Create System
mOSys = new MOSys(scheduler,toMos,MosToEnv);

/*@ Create env*/
env = new Environment("Meal Ordering System TestPane"); //for env imp
env.defMessage(Sms.class); //for env imp

//Inform gui about mediator to send
env.defMediator("To MOS", toMos); //for env imp
env.show(); /* //for env imp*/

//Start the system
t1.start();
System.out.println("***JavaFrame starting");
}

public static void main(String[] args)
{
    MOSysMain mOSysMain = new MOSysMain();
}

}
public Patient(Scheduler scheduler, Mediator toMe, Mediator toPatientStore, 
PhoneNo phoneNo, String lastName, String firstName, String defDrink, String defSize) 
{
    super(scheduler);
    this.toMe = toMe;
    this.toPatientStore = toPatientStore;
    this.phoneNo = phoneNo;
    this.lastName = lastName;
    this.firstName = firstName;
    this.defDrink = defDrink;
    this.defSize = defSize;
    toMe.addAddress(this);
    System.out.println("***Patient Created");
}

protected void execStartTransition() 
{
    states.enterState(this);
    // TODO Auto-generated method stub
}

public int getCategory() {
    return category;
}

public void setCategory(int category) {
    this.category = category;
}

public String getFirstName() {
    return firstName;
}

public void setFirstName(String firstName) {
    this.firstName = firstName;
}

public String getLastName() {
    return lastName;
}
/* @param lastName The lastName to set. */
public void setLastName(String lastName) {
    this.lastName = lastName;
}
/**
* @return Returns the maxSize.
*/
public String getDefSize() {
    return defSize;
}
/**
* @param maxSize The maxSize to set.
*@
public void setDefSize(String defSize) {
    this.defSize = defSize;
}
/**
* @return Returns the phoneNo.
*/
public PhoneNo getPhoneNo() {
    return phoneNo;
}
/**
* @param phoneNo The phoneNo to set.
*/
public void setPhoneNo(PhoneNo phoneNo) {
    this.phoneNo = phoneNo;
}
/**
* @return Returns the defDrink.
*/
public String getDefDrink() {
    return defDrink;
}
/**
* @param defDrink The defDrink to set.
*/
public void setDefDrink(String defDrink) {
    this.defDrink = defDrink;
}
}

---------------------------------------------------------------------------------------------------------------------
PatientStates.java
/*
* Created on 29.mar.2005
*
* Gøran
*/
package src;
import se.ericsson.eto.norarc.javaframe.CompositeState;
import se.ericsson.eto.norarc.javaframe.Message;
import se.ericsson.eto.norarc.javaframe.State;
import msg.*;
/**
 * @author Goran
 */
public class PatientStates extends CompositeState {

    /**
     * @param arg0
     */
    static State idle = new State("idle");
    public PatientStates(String sn) {
        super(sn);
        idle.enclosingState = this;
        // TODO Auto-generated constructor stub
    }

    /**
     *
     */
    public PatientStates() {
        super();
        // TODO Auto-generated constructor stub
    }

    public void enterState(StateMachine curfsm) {
        entry(curfsm);
        idle.enterState(curfsm);
    }

    /* (non-Javadoc)
     * @see se.ericsson.eto.norarc.javaframe.CompositeState#execTrans(se.ericsson.eto.norarc.javaframe.Message,
     * se.ericsson.eto.norarc.javaframe.State, se.ericsson.eto.norarc.javaframe.StateMachine)
     */
    protected boolean execTrans(Message sig, State st, StateMachine curfsm) {
        Patient p = (Patient)curfsm;
        if(sig instanceof GetPatientInfo) {
            GetPatientInfo gpi = (GetPatientInfo)sig;
            output(new PatientInfo(gpi.getAddress(), p), p.toPatientStore, p);
        }
        // TODO Auto-generated method stub
        return false;
    }
}

PhoneNo.java
/*
 * Created on 30.mar.2005
 * @author Goran
 */
package src;
public class PhoneNo {
    public String value;
    public PhoneNo(String phoneNo) {
        value = phoneNo;
    }
    public static PhoneNo valueOf(String phoneNo) {
        return new PhoneNo(phoneNo);
    }
    public int hashCode() {
        return value.hashCode();
    }
    public String toString() {
        return value;
    }
}

OrderStore.java
/*
 * Created on 30.mar.2005
 * Goran
 */
package src;
import se.ericsson.eto.norarc.javaframe.Composite;
import se.ericsson.eto.norarc.javaframe.Mediator;
import se.ericsson.eto.norarc.javaframe.Scheduler;

/*
 * @author Goran
 */
public class OrderStore extends Composite {
    protected Mediator toMe;
    protected Mediator toEnv;

    /*
     *
     */
    public OrderStore(Scheduler sched, Mediator toMe, Mediator toEnv) {
        super();
        // TODO Auto-generated constructor stub
Mediator fromOTOos = new Mediator();
Mediator toOfromOs = new Mediator();
addActiveObject(new Order(sched,toOfromOs,fromOTOos));

toMe.addAddress(toOfromOs);
fromOTOos.addAddress(toEnv);
System.out.println("***OrderStore Created");

OrderStates.java

package src;

import se.ericsson.eto.norarc.javaframe.CompositeState;
import se.ericsson.eto.norarc.javaframe.Message;
import se.ericsson.eto.norarc.javaframe.State;
import se.ericsson.eto.norarc.javaframe.StateMachine;

public class OrderStates extends CompositeState {
    static State idle = new State("idle");

    public OrderStates(String sn) {
        super(sn);
        idle.enclosingState = this;
        // TODO Auto-generated constructor stub
    }

    public OrderStates() {
        super();
        // TODO Auto-generated constructor stub
    }

    // (non-Javadoc)
    // @see se.ericsson.eto.norarc.javaframe.CompositeState#execTrans(se.ericsson.eto.norarc.javaframe.Message,
    // se.ericsson.eto.norarc.javaframe.State, se.ericsson.eto.norarc.javaframe.StateMachine)

    public void enterState(StateMachine curfsm) { /* @see */
}
{  
  entry(curfsm);
  idle.enterState(curfsm);
}

protected boolean execTrans(Message sig, State st, StateMachine curfsm) {
  System.out.println("Order registered in system");
  sameState(curfsm);
  return false;
}

Order.java

* *
* Created on 29.mar.2005
* *
* Goran
* /
package src;
import se.ericsson.eto.norarc.javaframe.Scheduler;
import se.ericsson.eto.norarc.javaframe.StateMachine;
import se.ericsson.eto.norarc.javaframe.Mediator;
import java.util.*;
/**
* @author Goran
* /
public class Order extends StateMachine {
  protected Mediator toMe;
  protected Mediator toEnv;
  protected HashMap patient;
  protected HashMap orders;
  static OrderStates states =new OrderStates("OrderStates");
  /**
   * @param arg0
   */
  public Order(Scheduler sched,Mediator toMe,Mediator toEnv) {
    super(sched);
    this.toMe=toMe;
    toMe.addAddress(this);
    // TODO Auto-generated constructor stub
  }

  /* (non-Javadoc)
   * @see se.ericsson.eto.norarc.javaframe.StateMachine#execStartTransition()
/*
protected void execStartTransition()
{
    states.enterState(this);
    System.out.println("Hello");
}
*/

MOSys.java
/*
* Created on 29.mar.2005
* 
* Goran
*/
package src;

import se.ericsson.eto.norarc.javaframe.*;

/**
* @author Goran 
*
public class MOSys extends Composite
{
    Controller c;
    /**
    *
    */
    public MOSys(Scheduler sched, Mediator toMe,Mediator toEnv)
    {
        super();
        /* TODO Auto-generated constructor stub
        * Create InnerParts, Menu,Controller,Order,PatientStore
        * Create the Mediators for the innerParts
        * Connect mediators
        */

        //Controller and its mediators
        //Inn till Controllern fra miljøet
        Mediator toCFromMenu = new Mediator();
        Mediator toCFromPs = new Mediator();
        Mediator toCFromOs = new Mediator();
        Mediator toCFromEnv = new Mediator(); //Router som tar alt?

        //Ut fra controlern Til de ulike parts
        Mediator fromCToMenu = new Mediator();
        Mediator fromCToPs = new Mediator();
        Mediator fromCToOs = new Mediator();
        Mediator fromCToEnv = new Mediator();

        c = new Controller(sched,fromCToMenu,fromCToPs,
        fromCToOs,fromCToEnv,toCFromEnv);
        addActiveObject(c);
    }
}
//Menu and its Mediators
Mediator toMenuFromC = new Mediator();
Mediator fromMenuToC = new Mediator();
addActiveObject(new Menu(sched, toMenuFromC, fromMenuToC));

//PatientStore and its mediators
Mediator toPsFromC = new Mediator();
Mediator fromPsToC = new Mediator();
addActiveObject(new PatientStore(sched, toPsFromC, fromPsToC));

//OrderStore and its mediators
Mediator toOsFromC = new Mediator();
Mediator fromOsToC = new Mediator();
addActiveObject(new OrderStore(sched, toOsFromC, fromOsToC));

//ConnectMediators
//Controller Out + In from Env
toMe.addAddress(toCFromEnv);
fromCToEnv.addAddress(toEnv);
fromCToMenu.addAddress(toMenuFromC);
fromCToPs.addAddress(toPsFromC);
fromCToOs.addAddress(toOsFromC);

//PatientStore Out Mediator
fromPsToC.addAddress(toCFromEnv);
//Menu Out Mediator
fromMenuToC.addAddress(toCFromEnv);
//OrderStore out Mediator
fromOsToC.addAddress(toCFromEnv);

System.out.println("***MOSystem created ");

MOS_RouterMediator.java

/*
 * Created on 30.mar.2005
 * 
 * Goran
 */
package src;

import java.util.HashMap;
import se.ericsson.eto.norarc.javaframe.Mediator;
import se.ericsson.eto.norarc.javaframe.Message;
import se.ericsson.eto.norarc.javaframe.Composite;
import msg.*;
/**
 * @author Goran
 */
public class MOS_RouterMediator extends Mediator
{ 
    HashMap mediators = new HashMap();

    public void forward(Message sig)
    {
        if(sig instanceof MOS_Message)
        {
            MOS_Message m = (MOS_Message) sig;
            String address = m.getAdress();
            System.out.println(address);

            // Sjekker om mediatoren finnes, ergoom sessionController er opprettet
            // Hvis den gjør det forwardes bare meldingen til den rette mediator
            if(mediators.containsKey(address))
            {
                Mediator med = (Mediator)mediators.get(address);
                med.forward(sig);
            }
            else
            {
                System.out.println("ERROR_ERROR_ERROR_ERROR");
                // System.out.println("No SessionController object for "+m.getAdress()+");
                // Mediator med = (Mediator)mediators.get(address);
                // med.forward(sig);
            }
        }
        else if (sig instanceof Sms)
        {
            Sms sms = (Sms) sig;
            String address = sms.getFrom();
            System.out.println(address);

            // Sjekker om mediatoren finnes, ergoom sessionController er opprettet
            // Hvis den gjør det forwardes bare meldingen til den rette mediator
            if(mediators.containsKey(address))
            {
                Mediator med = (Mediator)mediators.get(address);
                med.forward(sig);
            }
            else
            {
                System.out.println("ERROR_ERROR_ERROR_ERROR");
                // System.out.println("No SessionController object for "+m.getAdress()+");
                // Mediator med = (Mediator)mediators.get(address);
                // med.forward(sig);
            }
        }
    }
}
public Mediator getMediator(String address) {
    return (Mediator)mediators.get(address);
}

public void addMediator(String address, Mediator m) {
    mediators.put(address, m);
    System.out.println("Added to MediatorHashMap in MOS_RouterMediator");
}

public void removeMediator(String address, Mediator m) {
    mediators.remove(address);
    System.out.println("Removed from MediatorHashMap in MOS_RouterMediator");
}

public boolean isCreated(String address) {
    return mediators.containsKey(address);
}

---------------------------------------------------------------------------------------------------------------------

MessageTokenizer.java
/*
 * Created on 12.apr.2005
 * 
 * Goran
 */
package src;

import java.util.Vector;
import java.util.StringTokenizer;
import java.util.regex.Pattern;
import java.util.regex.Matcher;
/**
 * @author ??
 */
public class MessageTokenizer {
    private String message;
    private Vector words;

    public MessageTokenizer(String message) {
        this.message = (message!=null) ? message : ""
    }

    public MessageTokenizer(String message,int wordStartIndex) {
        String regex;
        if (wordStartIndex==0) {
            this.message = message;
        } else if (wordStartIndex>0) {
            regex = "\s*";
            for(int i = 0; i < wordStartIndex; i++) 
            
        }
regex += "\\S+\\s+";
}
regex += "(.*$\");
this.message = getMessagePart(regex,message);
} else {
    throw new IndexOutOfBoundsException("wordStartIndex must be greater than 0");
}
if (this.message==null) this.message = "";
}

public MessageTokenizer(String message,String wordsToRemove) {
    // If wordsToRemove contains two words, the two first words from 'message' will be removed
    // even if the two first words from 'message' match or not
    this(message,getWordCount(wordsToRemove));
}

public String getMessage() {
    return message;
}

public String getWord(int i) {
    String word;
    if (words==null) loadWords();
    try {
        word = (String)words.get(i);
    } catch (ArrayIndexOutOfBoundsException obex) {
        word = null;
    }
    return word;
}

public String getWords(int start,int end) {
    StringBuffer output;
    output= new StringBuffer();
    for(int i=start;i<end;i++) output.append(getWord(i)+" ");
    return output.toString().trim();
}

public String getReminder(int start) {
    return getWords(start,getWordCount());
}

public String getLastWord(int i) {
    if (words==null) loadWords();
    return (String)words.get(words.size()-1-i);
}
public int getWordCount() {
    if (words==null) loadWords();
    return words.size();
}

public static int getWordCount(String input) {
    if (input==null) input = "";
    return new StringTokenizer(input).countTokens();
}

public boolean matches(String regex) {
    return matches(regex,message);
}

public static boolean matches(String regex,String input) {
    Pattern p = Pattern.compile(regex);
    return p.matcher(input).matches();
}

public String getMessagePart(String regex) {
    return getMessagePart(regex,message);
}

public static String getMessagePart(String regex,String input) {
    Matcher matcher;
    Matcher matcher = Pattern.compile(regex).matcher(input);
    if (matcher.find())
        return matcher.group(1);
    else
        return null;
}

public Vector getMessageParts(String regex) {
    Matcher matcher;
    Vector vector;
    vector = new Vector();
    Matcher matcher = Pattern.compile(regex).matcher(message);
    while (matcher.find()) vector.add(matcher.group());
    return vector;
}

public static boolean isNumber(String input) {
    return matches("[^\d]*",input);
}

private void loadWords() {
    StringTokenizer tokenizer;
    words = new Vector();
    if (message==null) message = "";
    

tokenizer = new StringTokenizer(message);
while (tokenizer.hasMoreTokens()) words.add(tokenizer.nextToken());
}

public static void main(String[] args) {

    String input = "word0 word1 word2 word3 word4";
    MessageTokenizer token = new MessageTokenizer(input);

    System.out.println("Message: "+ token.getMessage()+ ");
    System.out.println("Word count: "+ token.getMessageCount()+ ");
    for(int i = 0; i < token.getMessageCount(); i++) {
        System.out.println("Word "+ i + ": " + token.getMessage(i) + ");
    }

    System.out.println("MessageParts: "+ token.getMessagePartsPlus(\"(d.)\") + ");
    for(int i = 0; i < token.getMessageParts().size(); i++) {
        System.out.println("MessagePart "+ i + ": " + (String)token.getMessageParts().get(i) + ");
    }

    System.out.println("isNumber "+ MessageTokenizer.isNumber(\"234345\"));
    MessageTokenizer token2 = new MessageTokenizer(input, 2);
    System.out.println("token2: "+ token2.getMessage()+ ");
}

MenuStates.java
/*
 * Created on 29.mar.2005
 *
 * Goran
 */
package src;

import se.ericsson.eto.norarc.javaframe.CompositeState;
import se.ericsson.eto.norarc.javaframe.Message;
import se.ericsson.eto.norarc.javaframe.State;
import msg.*;
/**
 * @author Goran
 */
public class MenuStates extends CompositeState{
    static State idle = new State("idle");
public MenuStates(String sn) {
    super(sn);
    idle.enclosingState=this;
    // TODO Auto-generated constructor stub
}

public MenuStates() {
    super();
    // TODO Auto-generated constructor stub
}

public void enterState(StateMachine curfsm)
{
    entry(curfsm);
    idle.enterState(curfsm);
}

protected boolean execTrans(Message sig, State st, StateMachine curfsm)
{
    // TODO Auto-generated method stub
    Menu menu = (Menu)curfsm;
    if(sig instanceof GetMeal)
    {
        GetMeal m = (GetMeal)sig;
        output(new MealOffer(m.getAdress(),menu.getMeal()),menu.toEnv,menu);
    }

    return true;
}
/**
 * @author Gøran
 */
public class Menu extends StateMachine {
    static MenuStates states = new MenuStates("Menu");
    protected Mediator toMe;
    protected Mediator toEnv;

    String[] meal = {"Chicken and Rice","Lamb chops and potatoes","Pork chop and rice"};

    /**
     * @param arg0
     */
    public Menu(Scheduler sched, Mediator toMe, Mediator toEnv) {
        super(sched);
        this.toMe = toMe;
        this.toEnv = toEnv;
        //this.meal = meal;
        toMe.addAddress(this);

        System.out.println("***Menu Created");
        // TODO Auto-generated constructor stub
    }

    /* (non-Javadoc)
     * @see se.ericsson.eto.norarc.javaframe.StateMachine#execStartTransition()
     */
    protected void execStartTransition() {
        states.enterState(this);
        // TODO Auto-generated method stub
    }

    /**
     * @return Returns the meal.
     */
    public String[] getMeal() {
        return meal;
    }

    /**
     * @param meal The meal to set.
     */
    public void setMeal(String[] meal) {
        this.meal = meal;
    }
}

Environment.java
/**
 * Created on 29.mar.2005
 *
 * Gøran
 */
package src;
import java.util.Vector;
import se.ericsson.eto.norarc.javaframe.Mediator;
import se.ericsson.eto.norarc.javaframe.MessageGUI;
import msg.*;
/**
 * @author Gøran
 */
public class Environment extends MessageGUI {

    /**
     * @param headerText
     */
    public Environment(String headerText)
    {
        super(headerText);
        System.out.println("***Environment Created");
    }

    /* (non-Javadoc)
     * @see se.ericsson.eto.norarc.javaframe.MessageGUI#generateMessage(java.lang.String,
     * java.util.Vector, se.ericsson.eto.norarc.javaframe.Mediator)
     */
    public void generateMessage(String msg, Vector params, Mediator med)
    {
        //TODO: Generate message based on user selection
        String p0 = (params.elementAt(0).toString());
        String p1 = (params.elementAt(1).toString());
        String p2 = (params.elementAt(2).toString());

        if (msg.compareTo("MealOrder")==0)
        {
            addLog("\nSend command MealOrder of ");
        }
        else if (msg.compareTo("MealHistory")==0)
        {
            addLog("\nSend command MealHistory of ");
        }
        else if (msg.compareTo("DisplayMealInfo")==0)
        {
            addLog("\nSend command DisplayMealInfo of ");
        }
        else if (msg.compareTo("RegisterPersonalInfo")==0)
        {
            addLog("\nSend command RegisterPersonalInfo of ");
        }
    }
}
else if (msg.compareTo("Sms") == 0)
{
    addLog("Send command Sms to: " + p0);
    med.input(new Sms(p0, p1, p2));
}

public void forward(Message sig)
{
    if (sig instanceof Sms)
    {
        Sms sms = (Sms) sig;
        //main.env.addLog("Status from the MOSystem: " + sms.getMessage());
    }
    else if (sig instanceof MealOffer)
    {
        MealOffer m = (MealOffer) sig;
        //main.env.addLog("MealOffer from system: " + m.getMeal());
    }
    else
    {
        System.out.println("Error: Unexpected signal to EdgeOut Mediator");
    }
}
import java.util.HashMap;
import se.ericsson.eto.norarc.javaframe.Composite;
import se.ericsson.eto.norarc.javaframe.Mediator;
import se.ericsson.eto.norarc.javaframe.Scheduler;
import se.ericsson.eto.norarc.javaframe.StateMachine;

/**
 * @author Gøran
 */
public class ControllerSM extends StateMachine
{
    HashMap sessionControllers = new HashMap();
    static ControllerSM_States states = new ControllerSM_States("ControllerSM_States");
    protected Mediator toCSMFromC;
    protected MOS_RouterMediator fromCSMToSC;
    Controller parent;

    public ControllerSM(Scheduler sched, Mediator toCSMFromC,MOS_RouterMediator fromCSMToSC,Controller parent)
    {
        super(sched);
        this.fromCSMToSC = fromCSMToSC;
        this.toCSMFromC = toCSMFromC;
        this.toCSMFromC.addAddress(this);
        this.parent = parent;

        states.enterState(this);
    }

    protected void execStartTransition()
    {
        states.enterState(this);
    }
}
package src;

import msg. *
import se.ericsson.eto.norarc.javaframe.Mediator;
import se.ericsson.eto.norarc.javaframe.Message;
import se.ericsson.eto.norarc.javaframe.State;
import se.ericsson.eto.norarc.javaframe.StateMachine;
import se.ericsson.eto.norarc.javaframe.CompositeState;

/**
 * @author Gøran
 */
public class ControllerSM_States extends CompositeState
{
    static State onlyState = new State("OnlyState");

    public ControllerSM_States(String sn)
    {
        super(sn);
        onlyState.enclosingState = this;
    }

    public void enterState(StateMachine curfsm)
    {
        entry(curfsm);
        onlyState.enterState(curfsm);
        System.out.println("Controller States : Initial");
    }

    /* (non-Javadoc)
     * @see se.ericsson.eto.norarc.javaframe.CompositeState#execTrans(se.ericsson.eto.norarc.javaframe.Message,
     * se.ericsson.eto.norarc.javaframe.State, se.ericsson.eto.norarc.javaframe.StateMachine)
     */
    protected boolean execTrans(Message sig, State state, StateMachine curfsm)
    {
        ControllerSM cSm = (ControllerSM)curfsm;

        if (sig instanceof Sms)
        {
            System.out.println("ControllerSM: Sms received");
            Sms sms = (Sms) sig;
            if (cSm.fromCSMToSC.isCreated(sms.getFrom()))
            {
                System.out.println("Session Object found");
                output(sig, cSm.fromCSMToSC, cSm);
            }
            else
            {
                System.out.println("No Entry in MOS_RouterMediator, no SessionController: Create SC!");
                cSm.parent.createSessionController(sms.getFrom());
            }
        }
    }
output(sig,cSm.fromCSMToSC,cSm);

} else if (sig instanceof MOS_Message) {
    System.out.println("ControllerSM: MOS_Message received");
    MOS_Message m = (MOS_Message)sig;
    if (cSm.fromCSMToSC.isCreated(m.phoneNo.value)) {
        System.out.println("Session Object found");
        output(sig,cSm.fromCSMToSC,cSm);
    } else {
        System.out.println("No Entry in MOS_RouterMediator, no SessionController: Create SC!");
        cSm.parent.createSessionController(m.phoneNo.value);
        output(sig,cSm.fromCSMToSC,cSm);
    }
} else if (sig instanceof MealOffer) {
    System.out.println("ControllerSM: MealOffer received");
    MealOffer m = (MealOffer)sig;
    if (cSm.fromCSMToSC.isCreated(m.phoneNo.value)) {
        System.out.println("Session Object found");
        output(sig,cSm.fromCSMToSC,cSm);
    } else {
        System.out.println("No Entry in MOS_RouterMediator, no SessionController: Trying to create!");
        cSm.parent.createSessionController(m.phoneNo.value);
        output(sig,cSm.fromCSMToSC,cSm);
    }

    return false;
}
import se.ericsson.eto.norarc.javaframe.*;
import msg.*;
/**
 * @author Gøran
 * /
 public class Controller extends Composite
 {
  //private HashMap sessionControllers = new HashMap();
  //disse inngår i mitt lille prosjekt
  private Scheduler sched;
  private Mediator toCFromEnv;
  private Mediator fromCToPs ;
  private Mediator fromCToEnv ;
  private Mediator fromCToOs ;
  private Mediator fromCToMenu ;
  private MOS_RouterMediator fromCSMToSC;
  /**
   *
   */
  public Controller(Scheduler sched,Mediator fromCToMenu,Mediator fromCToPs,
   Mediator fromCToOs,Mediator fromCToEnv,Mediator toCFromEnv)
  {
   super();

   //Inngår i test
   this.sched=sched;
   this.toCFromEnv = toCFromEnv;
   this.fromCToEnv = fromCToEnv;
   this.fromCToMenu = fromCToMenu;
   this.fromCToOs = fromCToOs;
   this.fromCToPs = fromCToPs;

   // ControllerSM and its Mediators
   Mediator toCSMFromC = new Mediator();
   fromCSMToSC = new MOS_RouterMediator();
   addActiveObject(new ControllerSM(sched,toCSMFromC,fromCSMToSC,this));

   toCFromEnv.addAddress(toCSMFromC);

   System.out.println("***Controller created");
  }

  //Metode som oppretter et nytt sessionController objekt, bassert på en
  //String address som hentes ut fra Message sig.
  public void createSessionController(String address)
  {
    System.out.println("Creating new Session Controller for: "+address);
    Mediator toScFromCSM = new Mediator();
    Mediator fromScToC_Ps = new Mediator();
    Mediator fromScToC_Env = new Mediator();
    Mediator fromScToC_Os = new Mediator();
  }
Mediator fromScToC_M = new Mediator();
SessionController sc = new SessionController(sched, toScFromCSM, fromScToC_Ps,
fromScToC_Env, fromScToC_Os, fromScToC_M);
addActiveObject(sc);
Mediator m = new Mediator();
m.addAddress(toScFromCSM);

// Legger CSM ut Mediatoren inn i routeren
fromCSMToSC.addMediator(address, m);

// Setter adressene til utMediatorene til SessionControlleren
fromScToC_Ps.addAddress(fromCToPs);
fromScToC_Env.addAddress(fromCToEnv);
fromScToC_Os.addAddress(fromCToOs);
fromScToC_M.addAddress(fromCToMenu);

} public void removeSessionController(String address) {
    System.out.println("Removing sessionController: "+address);
}

} 

PatientStore.java
/*
 * Created on 29.mar.2005
 * @author Gøran
 */
package src;
import se.ericsson.eto.norarc.javaframe.*;
/**
 * @author Gøran
 */
public class PatientStore extends Composite {

     public PatientStore(Scheduler sched, Mediator toMe, Mediator toEnv) {
         super();
         Mediator toPatientFromPs = new Mediator();
         Mediator fromPatientToPs = new Mediator();
         addActiveObject(new Patient(sched, toPatientFromPs, fromPatientToPs,
new PhoneNo("91390914"), "Haugen", "Øystein", "Milk", "Large");
     // TODO Auto-generated constructor stub
toMe.addAddress(toPatientFromPs);
fromPatientToPs.addAddress(toEnv);
System.out.println("***PatientStore Created");
}
package src;

import se.ericsson.eto.norarc.javaframe.*;

public class SessionController extends StateMachine
{
    static SessionControllerStates states = new SessionControllerStates("SessionController");
    protected Mediator fromScToC_Ps;
    protected Mediator fromScToC_Env;
    protected Mediator fromScToC_Os;
    protected Mediator fromScToC_M;

    public SessionController (Scheduler sched, Mediator toScFromCSM, Mediator fromScToC_Ps, Mediator fromScToC_Env, Mediator fromScToC_Os, Mediator fromScToC_M)
    {
        super(sched);
        this.fromScToC_Env = fromScToC_Env;
        this.fromScToC_M = fromScToC_M;
        this.fromScToC_Os = fromScToC_Os;
        this.fromScToC_Ps = fromScToC_Ps;

        toScFromCSM.addAddress(this);
        System.out.println("***SessionController Created");
        // TODO Auto-generated constructor stub
    }

    /* (non-Javadoc)
     * @see se.ericsson.eto.norarc.javaframe.StateMachine#execStartTransition()
     */
    protected void execStartTransition()
    {
        states.enterState(this);
        // TODO Auto-generated method stub
    }
}
SessionControllerStates.java

/*
* Created on 30.mar.2005
* Goran
*/
package src;
import se.ericsson.eto.norarc.javaframe.*;
import msg.*;

/**
* @author Goran
*/
public class SessionControllerStates extends CompositeState
{
    //Diverse data for mellomlagring
    protected Patient p;
    protected String[] menu;
    MessageTokenizer messageTok;
    String[] order = new String[3];

    static State idle = new State("idle");
    static State waitingForMenu = new State("waitingForMenu");
    static State waitingForConfirmation = new State("waitingForConfirmation");
    static State waitingForPatientInfo = new State("waitingForPatientInfo");

    /**
     * @param arg0
     */
    public SessionControllerStates(String sn)
    {
        super(sn);
        idle.enclosingState = this;
        waitingForMenu.enclosingState = this;
        waitingForConfirmation.enclosingState = this;
        waitingForPatientInfo.enclosingState = this;
        // TODO Auto-generated constructor stub
    }

    /**
     *
     */
    public SessionControllerStates()
    {
        super();
        // TODO Auto-generated constructor stub
    }

    public void enterState(StateMachine curfsm)
    {
        entry(curfsm);
        idle.enterState(curfsm);
    }

    /* (non-Javadoc)
     * ..."
protected boolean execTrans(Message sig, State st, StateMachine curfsm)
{
    SessionController sc = (SessionController)curfsm;

    // TODO Auto-generated method stub
    if (st == idle)
    {
        if(sig instanceof Sms)
        {
            System.out.println("Sms recived in Session Controller");
            Sms sms = (Sms)sig;
            System.out.println(sms.getFrom()+sms.getTo()+sms.getMessage());
            messageTok = new MessageTokenizer(sms.getMessage());
            String command = messageTok.getWord(3);
            System.out.println("Command: "+command);
            if (command.equals("order"))
            {
                output(new GetPatientInfo(sms.getFrom()),sc.fromScToC Ps,sc);
                waitingForPatientInfo.enterState(curfsm);
            }
        }
        if(sig instanceof MealHistory)
        {
        }
        if(sig instanceof DisplayMealInfo)
        {
        }
    }
    else if (st == waitingForPatientInfo)
    {
        if(sig instanceof PatientInfo)
        {
            System.out.println("PatientInfo recived in Session Controller");
            PatientInfo pi = (PatientInfo)sig;
            p = (pi.getPatient());
            output(new GetMeal(pi.getAdress()),sc.fromScToC M,sc);
            waitingForMenu.enterState(curfsm);
        }
    }
    else if (st == waitingForMenu)
    {
        if(sig instanceof MealOffer)
        {
            MealOffer m = (MealOffer)sig;
        }
    }
System.out.println("MealOffer recived in Session Controller");
menu = m.getMeal();
order[0]=m.getMeal()[0];
order[1]=p.getDefDrink();
order[2]=p.getDefSize();
String mm="Hello "+p.getFirstName()+" "+p.getLastName()+" Todays Meal:
"+m.getMeal()[0] + ", Drink: "+p.getDefDrink()+" Size: "+p.getDefSize()+" Menu: 
\[@1\]:OK \\[@2\]:Update";
output(new Sms(m.getAdress(),"2034",mm),sc.fromScToC_Env,sc);
waitingForConfirmation.enterState(curfsm);
}
else if (st == waitingForConfirmation) {

if(sig instanceof Sms) {
 System.out.println("Sms recived in Session Controller");
Sms sms = (Sms)sig;
messageTok = new MessageTokenizer(sms.getMessage());
String command = messageTok.getWord(3).toLowerCase();
System.out.println("Command: "+command);

//Command number from Meal proposal
if (command.equals("@1")) {
 output(new Sms(p.phoneNo.toString(),"2034","Order confirmation:"+command),sc.fromScToC_Env,sc);
 System.out.println("Order: "+ order[0] +" "+order[1] +" "+order[2]);
 output(new MealOrder(p.phoneNo.toString(),order),sc.fromScToC_Os,sc);
 idle.enterState(curfsm);
}
else if(command.equals("@2")) {
 output(new Sms(p.phoneNo.toString(),"2034","Todays menu:
01."+menu[0]+"n02."+menu[1]+"n03."+menu[2]+" Reply[nr drink size]"),sc.fromScToC_Env,sc);
sameState(curfsm);
}
else if(command.equals("@3")) {
 order[1]=messageTok.getWord(4);
 order[2]=messageTok.getWord(5);
 output(new Sms(p.phoneNo.toString(),"2034","Order confirmation:"+command),sc.fromScToC_Env,sc);
}
System.out.println("Order: "+ order[0]+" "+order[1]+" "+order[2]);
output(new MealOrder(p.phoneNo.toString(),order),sc.fromScToC_Os,sc);
idle.enterState(curfsm);

//Dinner number from menu
else if(command.equals("01"))
{
    order[0]=menu[0];
    order[1]= messageTok.getWord(4);
    order[2]= messageTok.getWord(5);
    output(new Sms(p.phoneNo.toString(),"2034","Order confirmation:"+command),sc.fromScToC_Env,sc);
    System.out.println("Order: "+ order[0]+" "+order[1]+" "+order[2]);
    output(new MealOrder(p.phoneNo.toString(),order),sc.fromScToC_Os,sc);
    idle.enterState(curfsm);
}
else if(command.equals("02"))
{
    order[0]=menu[1];
    order[1]= messageTok.getWord(4);
    order[2]= messageTok.getWord(5);
    output(new Sms(p.phoneNo.toString(),"2034","Order confirmation:"+command),sc.fromScToC_Env,sc);
    System.out.println("Order: "+ order[0]+" "+order[1]+" "+order[2]);
    output(new MealOrder(p.phoneNo.toString(),order),sc.fromScToC_Os,sc);
    idle.enterState(curfsm);
}
else if(command.equals("03"))
{
    order[0]=menu[2];
    order[1]= messageTok.getWord(4);
    order[2]= messageTok.getWord(5);
    output(new Sms(p.phoneNo.toString(),"2034","Order confirmation:"+command),sc.fromScToC_Env,sc);
    System.out.println("Order: "+ order[0]+" "+order[1]+" "+order[2]);
    output(new MealOrder(p.phoneNo.toString(),order),sc.fromScToC_Os,sc);
    idle.enterState(curfsm);
}
return true;
/*SessionController sc =(SessionController)curfsm;
return true;*/
Messages

MealOffer.java

```java
package msg;

public class MealOffer extends MOS_Message {
    private String[] meal;

    public MealOffer(String phoneNo, String[] meal) {
        super(phoneNo);
        this.meal = meal;
    }

    public String[] getMeal() {
        return meal;
    }
}
```

GetMeal.java

```java
package msg;

public class GetMeal extends MOS_Message {
    public GetMeal(String phoneNo) {
        super(phoneNo);
    }
}
```

Status.java

```java
package msg;

public class Status {
    // Method definitions...
}
```
package msg;
import se.ericsson.eto.norarc.javaframe.Message;

public class Status extends MOS_Message {
    public String stat;
    
    public Status(String stat) {
        super(stat);
        this.stat = stat;
        // TODO Auto-generated constructor stub
    }
    
    public String messageContent() {
        return "Status: "+ this.stat;
    }
}

package msg;

public class RegOrder extends MOS_Message {
    private String meal;
    private String drink;
    private String size;
    
    public RegOrder(String phoneNo, String meal, String drink, String size) {
        super(phoneNo);
        this.meal = meal;
        this.drink = drink;
        this.size = size;
    }
    
    public String getDrink() {
        return drink;
    }
}
/*
   * @param drink The drink to set.
   */
   public void setDrink(String drink)
   {
       this.drink = drink;
   }

   /**
   * @return Returns the meal.
   */
   public String getMeal()
   {
       return meal;
   }

   /**
   * @param meal The meal to set.
   */
   public void setMeal(String meal)
   {
       this.meal = meal;
   }

   /**
   * @return Returns the size.
   */
   public String getSize()
   {
       return size;
   }

   /**
   * @param size The size to set.
   */
   public void setSize(String size)
   {
       this.size = size;
   }
}

RegisterPersonalInfo.java
/*
 * Created on 29.mar.2005
 * 
 * Goran
 */
package msg;
import se.ericsson.eto.norarc.javaframe.Message;
/**
 * @author Goran
 */
public class RegisterPersonalInfo extends MOS_Message {
    /**
     * 
     */
    public RegisterPersonalInfo(String pNr) {
        super(pNr);
        // TODO Auto-generated constructor stub
    }
PatientInfo.java

/*
 * Created on 12.apr.2005
 * 
 * Goran
 */
package msg;
import src.Patient;
/**
 * @author Goran
 */
public class PatientInfo extends MOS_Message {

    private Patient patient;

    public PatientInfo(String phoneNo, Patient p) {
        super(phoneNo);
        this.patient = p;
        // TODO Auto-generated constructor stub
    }

    public Patient getPatient() {
        return patient;
    }

    public void setP(Patient p) {
        this.patient = p;
    }
}

OrderConfirmation.java

/*
 * Created on 06.apr.2005
 * 
 * Goran
 */
package msg;
/**
 * @author Goran
 * Message from user to confirm an MealOffer
 * The confirmation can be just that, or an small upupdate
 */
public class OrderConfirmation extends MOS_Message {

    public OrderConfirmation(String phoneNo) {
    
```
super(phoneNo);
   // TODO Auto-generated constructor stub
}

MOS_Message.java
/*
 * Created on 30.mar.2005
 * Gøran
 */
package msg;
import se.ericsson.eto.norarc.javaframe.Message;
import src.PhoneNo;
/**
 * @author Gøran
 */
public class MOS_Message extends Message
{
   public PhoneNo phoneNo;
   /**
    *
    */
   public MOS_Message(String phoneNo)
   {
      super();
      this.phoneNo=new PhoneNo(phoneNo);
      // TODO Auto-generated constructor stub
   }
   public String getAdress()
   {
      return phoneNo.toString();
   }
}

MealOrder.java
/*
 * Created on 29.mar.2005
 * Gøran
 */
package msg;
import se.ericsson.eto.norarc.javaframe.Message;
/**
 * @author Gøran
 */
public class MealOrder extends MOS_Message {
   private String[] order;
   /**
    *
    */
   public MealOrder(String pNr,String []order) {
      super(pNr);
this.order = order;
   // TODO Auto-generated constructor stub
}
/**
 * @return Returns the order.
 */
public String[] getOrder() {
    return order;
}
/**
 * @param order The order to set.
 */
public void setOrder(String[] order) {
    this.order = order;
}

---------------------------------------------------------------------------------------------------------------------

MealHistory.java

/*
 * Created on 29.mar.2005
 *
 * Goran
 */
package msg;
import se.ericsson.eto.norarc.javaframe.Message;
/**
 * @author Goran
 */
public class MealHistory extends MOS_Message {
    /**
     *
     */
    public MealHistory(String pNr) {
        super(pNr);
        // TODO Auto-generated constructor stub
    }
}

---------------------------------------------------------------------------------------------------------------------

GetPatientInfo.java

/*
 * Created on 12.apr.2005
 *
 * Goran
 */
package msg;
/**
 * @author Goran
 */
public class GetPatientInfo extends MOS_Message {
    /**
     * @param phoneNo
     */
    public GetPatientInfo(String phoneNo) {
        super(phoneNo);
        // TODO Auto-generated constructor stub
    }
}
DisplayMealInfo.java

/*
 * Created on 29.mar.2005
 * 
 * Gøran
 */
package msg;
import se.ericsson.eto.norarc.javaframe.Message;
/**
 * @author Gøran
 */
public class DisplayMealInfo extends MOS_Message {
    /**
     */
    MeaInfo meali;
    public DisplayMealInfo(String phoneNo,MeaInfo meali) {
        super(phoneNo);
        this.meali =meali
        // TODO Auto-generated constructor stub
    }
}
Appendix C

Commonalities and Variabilities

<table>
<thead>
<tr>
<th>Commonalities</th>
<th>Variabilities and parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Behaviour</strong></td>
<td></td>
</tr>
<tr>
<td>C1</td>
<td>Offer menu to patient</td>
</tr>
<tr>
<td>C2</td>
<td>Handled order</td>
</tr>
<tr>
<td>C3</td>
<td>In a fixed sequence</td>
</tr>
<tr>
<td>C4</td>
<td>Users trigger behaviour by sending SMS</td>
</tr>
<tr>
<td><strong>Static Structure</strong></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>Patient Data</td>
</tr>
<tr>
<td>C6</td>
<td>Controller / Session Controller</td>
</tr>
<tr>
<td>C7</td>
<td>Signals to and from user are SMS</td>
</tr>
<tr>
<td>C8</td>
<td>System has a given amount of legal signals</td>
</tr>
<tr>
<td>C9</td>
<td>Menu, Meal objects</td>
</tr>
<tr>
<td>C10</td>
<td>Same platform (PATS)</td>
</tr>
<tr>
<td>C11</td>
<td>Same API, workflow, policy</td>
</tr>
<tr>
<td>C12</td>
<td>Communicates thru terminals</td>
</tr>
</tbody>
</table>

Variability space

<table>
<thead>
<tr>
<th>Variability</th>
<th>Description</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>Menu varies depending on the application</td>
<td>Meals offered (DAI) #DeriK [DAS] Dynamic context Aware Order-Confirm, Order-Check, Confirm set</td>
</tr>
<tr>
<td>V2</td>
<td>Actions applied vary depending on application (Patient)</td>
<td>Same as above, depending on application</td>
</tr>
<tr>
<td>V3</td>
<td>Sequential order of signals may vary depending on the [1:1]</td>
<td>[Help], [order], and [customize menu] Language, [spaces]</td>
</tr>
<tr>
<td>V4</td>
<td>Number of applications can vary</td>
<td>Depending on application, some provides help</td>
</tr>
<tr>
<td>V5</td>
<td>Applications can be a combination of sub applications</td>
<td>Comes with Java VM</td>
</tr>
<tr>
<td>V6</td>
<td>SMS can vary in form and content</td>
<td></td>
</tr>
<tr>
<td>V7</td>
<td>Signals can vary</td>
<td></td>
</tr>
<tr>
<td>V8</td>
<td>Different kinds of operating systems is supported</td>
<td></td>
</tr>
<tr>
<td>V9</td>
<td>Number of signals can vary depending on each app</td>
<td></td>
</tr>
<tr>
<td>V10</td>
<td>Different kinds of terminals can be used</td>
<td></td>
</tr>
</tbody>
</table>
Graph documentation: ExampleSM
Status:
Personnel:
Documentation:

Graph picture: ExampleSM

<table>
<thead>
<tr>
<th>Object</th>
<th>Type of object</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start</td>
<td>Start</td>
<td></td>
</tr>
<tr>
<td>nr1</td>
<td>State [UML]</td>
<td></td>
</tr>
<tr>
<td>nr2</td>
<td>State [UML]</td>
<td></td>
</tr>
<tr>
<td>nr3</td>
<td>State [UML]</td>
<td></td>
</tr>
</tbody>
</table>

: Start
Properties:

Start relationships:
### In role | In relationship | With object(s) | In role
---|---|---|---
: From | MOS_Transition | nr1 | To

#### Start links:

<table>
<thead>
<tr>
<th>Link type</th>
<th>Graph's name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>none</td>
</tr>
<tr>
<td>Explosions</td>
<td>none</td>
</tr>
</tbody>
</table>

nr1: State [UML]

#### Properties:

<table>
<thead>
<tr>
<th>State name</th>
<th>nr1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal activities</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
</tr>
<tr>
<td>Symbol transparent?</td>
<td>F</td>
</tr>
</tbody>
</table>

nr1 relationships:

<table>
<thead>
<tr>
<th>In role</th>
<th>In relationship</th>
<th>With object(s)</th>
<th>In role</th>
</tr>
</thead>
<tbody>
<tr>
<td>: From</td>
<td>MOS_Transition</td>
<td>nr3</td>
<td>To</td>
</tr>
<tr>
<td>: From</td>
<td>MOS_Transition</td>
<td>nr2</td>
<td>To</td>
</tr>
<tr>
<td>: To</td>
<td>MOS_Transition</td>
<td>Start</td>
<td>From</td>
</tr>
<tr>
<td>: To</td>
<td>MOS_Transition</td>
<td>nr2</td>
<td>From</td>
</tr>
<tr>
<td>: To</td>
<td>MOS_Transition</td>
<td>nr3</td>
<td>From</td>
</tr>
</tbody>
</table>

nr1 links:

<table>
<thead>
<tr>
<th>Link type</th>
<th>Graph's name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>none</td>
</tr>
<tr>
<td>Explosions</td>
<td>none</td>
</tr>
</tbody>
</table>

#### nr2: State [UML]

#### Properties:

<table>
<thead>
<tr>
<th>State name</th>
<th>nr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal activities</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
</tr>
<tr>
<td>Symbol transparent?</td>
<td>F</td>
</tr>
</tbody>
</table>

nr2 relationships:
### nr2 links:

<table>
<thead>
<tr>
<th>Link type</th>
<th>Graph's name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>none</td>
</tr>
<tr>
<td>Explosions</td>
<td>none</td>
</tr>
</tbody>
</table>

### nr3: State [UML]

**Properties:**

<table>
<thead>
<tr>
<th>State name</th>
<th>nr3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Internal activities</td>
<td></td>
</tr>
<tr>
<td>Documentation</td>
<td></td>
</tr>
<tr>
<td>Symbol transparent?</td>
<td>F</td>
</tr>
</tbody>
</table>

### nr3 relationships:

<table>
<thead>
<tr>
<th>In role</th>
<th>In relationship</th>
<th>With object(s)</th>
<th>In role</th>
</tr>
</thead>
<tbody>
<tr>
<td>: From</td>
<td>: MOS_Transition</td>
<td>nr2</td>
<td>: To</td>
</tr>
<tr>
<td>: From</td>
<td>: MOS_Transition</td>
<td>nr1</td>
<td>: To</td>
</tr>
<tr>
<td>: To</td>
<td>: MOS_Transition</td>
<td>nr1</td>
<td>: From</td>
</tr>
</tbody>
</table>

### nr3 links:

<table>
<thead>
<tr>
<th>Link type</th>
<th>Graph's name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decomposition</td>
<td>none</td>
</tr>
<tr>
<td>Explosions</td>
<td>none</td>
</tr>
</tbody>
</table>