

**SWEDISH SCHOOL OF ECONOMICS AND BUSINESS  
ADMINISTRATION**

# **EXECUTABLE UML**

Student: Leonardo Galvão Cavalcanti

Supervisor: Anders Tallberg

Department: Accounting	Type of Work: Master of Science Thesis
Author: Leonardo Galvão Cavalcanti	Date: 31/3/2004
<p>Title of Thesis (please write in capital letters): EXECUTABLE UML</p>	
<p>Abstract:</p> <p>The Unified Modeling Language (UML) was created as a confluence of three well-known modeling methodology. It has gained wide popularity and has become a de-facto standard when it comes to visual modeling of software systems. It is taught in a great number of school and has been used by a great number firms of industry.</p> <p>Despite its popularity and the heavy investment that has been made in UML tools and expertise, the UML is not yet readily translatable into running code. Some of the problems that have been identified with the language include a lack of a action semantics language and its sheer size.</p> <p>Some renowned software methodologists have set out to sort the problems that prevent UML from being readily executable. They have created a dialect of UML that leaves away some of UML's unnecessary features and added action semantics. Those methodologists hope that this sub-language will be at a higher level of abstraction, which would allow domain experts to take active part in the modeling of applications. This sub-language is called executable UML.</p> <p>This research aims to study this new methodology. Some of the topics researched are: can executable UML raise the level of abstraction? Can it bring productivity gains? How about the tools for executable UML on the market? What are the possible shortcomings with the methodology?</p> <p>In order to gain insights into the methodology and answer some of the questions posed above, we build a prototype of an existing system using the executable UML methodology.</p>	
<p>Keywords: <b>UML, MDA, executable UML, Model Driven Architecture, Modeling, Crebit</b></p>	

## TABLE OF CONTENTS

<b>1. INTRODUCTION</b> .....	<b>6</b>
<b>1.1 Background</b> .....	<b>6</b>
1.1.1 Software Development.....	6
1.1.2 The Unified Modeling Language.....	7
1.1.3 Executable UML.....	8
<b>1.2 Research Objectives</b> .....	<b>9</b>
<b>2. UML – THE UNIFIED MODELING LANGUAGE</b> .....	<b>10</b>
<b>2.1 Objective and Structure</b> .....	<b>10</b>
<b>2.2 Definition</b> .....	<b>10</b>
<b>2.3 Brief History of UML</b> .....	<b>11</b>
<b>2.4 UML Goals</b> .....	<b>11</b>
<b>2.5 UML Diagrams</b> .....	<b>11</b>
2.5.1 Use Case diagrams.....	14
2.5.2 Class diagrams.....	16
2.5.3 Package diagrams.....	24
2.5.4 Object diagram.....	25
2.5.5 Interaction diagrams.....	26
2.5.6 Activity Diagrams.....	30
2.5.7 State-chart diagram.....	34
2.5.8 Component diagram.....	36
2.5.9 Deployment diagrams.....	38
<b>2.6 Stereotypes and Extensibility Mechanisms</b> .....	<b>39</b>
<b>2.7 Constraints</b> .....	<b>40</b>
<b>3. EXECUTABLE UML</b> .....	<b>42</b>
<b>3.1 Objective and Structure</b> .....	<b>42</b>
<b>3.2 Introduction</b> .....	<b>42</b>
<b>What is Executable UML?</b> .....	<b>43</b>

<b>3.3 The Executable UML Language</b> .....	<b>44</b>
3.3.1 Class Diagrams in Executable UML.....	46
3.3.2 State-chart Diagrams in Executable UML .....	48
<b>4. MODEL-DRIVEN ARCHITECTURE</b> .....	<b>49</b>
<b>4.1 Objectives</b> .....	<b>49</b>
<b>4.2 Introduction</b> .....	<b>49</b>
<b>4.3 Principles of Model-Driven Architecture</b> .....	<b>50</b>
<b>4.4 The Four Levels of Model-Driven Architecture Modeling</b> .....	<b>51</b>
<b>4.5 Model-Driven Architecture Standards</b> .....	<b>52</b>
<b>4.6 Model-Driven Architecture and Executable UML</b> .....	<b>53</b>
<b>5. THE EXECUTABLE UML PROCESS</b> .....	<b>54</b>
<b>5.1 Objectives</b> .....	<b>54</b>
<b>5.2 Introduction</b> .....	<b>54</b>
<b>5.3 Modeling: Elaborative x Translative</b> .....	<b>56</b>
<b>6. EMPIRICAL RESEARCH</b> .....	<b>60</b>
<b>6.1 Objective</b> .....	<b>60</b>
<b>6.2 Research Objectives</b> .....	<b>60</b>
<b>6.3 Research Methodology</b> .....	<b>61</b>
<b>6.4 Research Design</b> .....	<b>63</b>
6.4.1 Main Considerations .....	63
6.4.2 Crebit.....	63
<b>7. RESEARCH</b> .....	<b>71</b>
<b>7.1 Objectives</b> .....	<b>71</b>
<b>7.2 Tools</b> .....	<b>71</b>
<b>7.3 Modeling Considerations</b> .....	<b>73</b>
<b>7.4 Creating an executable model of the application</b> .....	<b>74</b>
<b>7.5 The dynamic part of the application</b> .....	<b>81</b>
<b>7.6 Generating the target code</b> .....	<b>84</b>

<b>7.7 The user interface.....</b>	<b>86</b>
<b>8. CONCLUSIONS .....</b>	<b>92</b>

# 1. INTRODUCTION

## 1.1 Background

### 1.1.1 Software Development

Before the seventies, software was developed in a very undisciplined manner that was termed code-and-fix. In other words, the software development process basically consisted of programmers writing code and fixing them later as problems arise. This practice, obviously, far from guaranteed the delivery of good-quality software. On the contrary, it caused what was one of the causes of the so-called "the software crisis".

In response to this practice, Barry Boehm introduced the waterfall software development methodology. The waterfall model was greatly inspired by the practice of more mature engineering fields, such as civil and mechanical engineering. It placed a great emphasis on the activities to be performed and their ordering. The model divided the software development lifecycle into a series of phases such as requirements, product design, detailed design, coding, integration and testing. No other phase could start before all the activities pertaining to the previous phase had been completed. Much in the same way as activities in civil engineering construction, which have to follow a strict sequence: foundations have to be built before the structure; structural elements have to be in place before the walls are raised, etc. The waterfall was useful in bringing discipline to the software development process but was later found not be adequate. One of the important drawbacks of the model is that it does not account for changing requirements, which are much more volatile in software development than they are in building construction, that oftentimes caused the product delivered not to satisfy the customer's needs. Another drawback is that integration was done too late, which normally caused budget and schedule overruns. The waterfall model has been largely abandoned, but still attracts interest because of its historical significance as one of the first software development methodologies.

Since the advent of the waterfall model, a number of other methodologies have been proposed, such as the spiral model or the evolutionary approach. Lately, methodologies such as the Rational Unified Process and agile methods, such as Extreme Programming, have become the most widely accepted and used

What many or most of these methodologies brought was a great emphasis in analysis and planning. Most of them advocated for the importance of design against a pure

"code-it" approach. Because of the advantages of visual modeling, such as greater intuitivity and easiness of expression, it became one of the dominant forms of design. Design artifacts are like blueprints to construction engineering, they provide detailed specification of how the system should be built.

### 1.1.2 The Unified Modeling Language

The Unified Modeling Language, or UML for short, was the result of the so-called "notational wars". In the late eighties and early nineties a great number of object oriented methods sprung up. More than 30 methods were published during this period, each with its own notation. UML used notation and ideas from many of the methods existing at that time (Mellor and Balcer, 2002, pg. 4), specially from Booch and Jacobson's OOSE (Object Oriented Software Engineering) and Rumbaugh's OMT (Object Modeling Technique). These three authors got together and developed UML, while their main source of inspiration was these two methods, but they acknowledged to have used ideas from other methods, including Fusion, Shlaer-Mellor and Coad-Yourdon (Booch et al 1999, preface).

UML has gained wide popularity. UML is now taught in a significant number of schools and is used by a great number of firms in their software development process. There's also a great of availability of tools in the market that support UML. It can be said that UML is now a de facto standard for visual modeling of software artifacts. As Murray Cantor acknowledged in his book about object oriented project management: "there's really no other choice" (Cantor 1998, pg. 46).

Besides software development, UML has also been used in other areas, for instance Business Modeling. Together with its ebXML standard, the UN/CEFACT, has also published a methodology for business modeling based on UML, called UMM. The high availability of tools and its extensibility mechanisms as well as broad scope make UML an attractive option for modeling in other areas than software development.

Initially a proprietary technology developed inside Rational corporation, UML was later submitted to the OMG (Object Management Group) for standardization. The UML trademark and logo now belong to this organization. The involvement of the OMG with UML is considered a landmark, as the first time the organization is involved with a technology not directly related to its CORBA standard.

UML is now one of the key enablers of the OMG's model-driven architecture. This new initiative by the OMG aims at creating an architecture centered on the production and exchange of models based on inter-changeable metadata.

### 1.1.3 Executable UML

Mellor and Balcer (2002, pg.2) noted that “the history of software development is a history of raising the level of abstraction”. Computers were initially programmed in machine code, which consisted of a series of zeroes and ones to be interpreted by the processor. Later, the assembly language substituted some of these codes by easier-to-remember mnemonics. The act of programming continued machine centered, but it was made easier, some of these mnemonics could be reused as new processors were introduced. Productivity was improved. Later, high-level programming languages such as FORTRAN and BASIC were born. These new languages were closer to human language than to machine language. They also increased the possibilities of re-use, as the same code could be translated to different processors without being altered.

The crafting of design artifacts using UML can be regarded as a higher level of abstraction above 3<sup>rd</sup> and 4<sup>th</sup> generation languages of today, such as java and Visual Basic. The importance of design stressed by some software development methodologies is justifiable, but it raises a risk in the real world. Often, the design produced by design specialists is regarded as mere recommendations by programmers who will actually implement the system. The result in some cases can be disastrous. The design and the code are kept out-of-synch, which defeats the purpose of the design, it becomes incapable of describing and explaining the real system to which it does not bear close resemblance. All the investment made in design is then lost. Instead of bringing benefits, design in this case causes the cost of the software system to double, it becomes the cost of designing plus the cost of implementing it.

Executable UML, as the name implies, is an attempt at making UML directly executable or translatable to 3<sup>rd</sup> or 4<sup>th</sup> generation programming languages. UML in its current state is not directly executable. It contains a hefty number of constructs that can be used in different ways by different organizations. It also lacks action semantics to describe the steps executed by the system in response to events. Executable UML tries mitigate the risk described in the paragraph above, by making the UML artifacts an integral part of the production of systems, not just non-imperative design. Executable UML, coupled with Model-Driven Architecture, may be the foundation for a new

software development methodology in which domain (business) experts would be involved in the crafting of high-level models and technologists would be concerned with building model compilers that would translate the models to 3<sup>rd</sup> or 4<sup>th</sup> generation code. These model compilers would be created by highly skilled computer specialists using the best practices and advanced knowledge of the computer science and software technology.

Executable UML thus also aspires to become the next level of abstraction, more accessible to domain experts, who could more easily design and specify the characteristics of the system taking into account expert domain knowledge. According to the words of Mellor and Balcer (2002, pg. 5):

Executable UML is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software so that a specification built in Executable UML can be deployed in various software environments without changes.

There are already tools in the market that provide for round-trip engineering. Round-trip engineering consists of generating code from models and updating the model as the code is changed by other means. While this approach is known to improve productivity, it does not raise the level abstraction. Models become just graphical representations of the code, which determines the level of abstraction. This approach is obviously tied to the platform for which the code is generated, providing for little actual software-platform independence.

## **1.2 Research Objectives**

The objective of the research is to evaluate and understand the current state-of-the-art regarding Executable UML. We will try to evaluate tools for executable UML and try to understand the current issues about Executable UML, and whether it can successfully raise the level abstraction as it aims at.

## 2. UML – THE UNIFIED MODELING LANGUAGE

### 2.1 Objective and Structure

The objective of this chapter is to provide the reader with an overview of UML and its main concepts. The subject of UML is somewhat extensive and we obviously do not intend to provide a fully comprehensive treatment of the subject. In order to obtain a more thorough explanation of UML and its constructs please refer to Booch, Rumbaugh and Jacobson (1999), Rumbaugh, Jacobson and Booch (1998), Fowler and Scott (1999) or Quatrani (1997). This chapter is largely based on these works.

### 2.2 Definition

In the words of Booch, Rumbaugh and Jacobson (1999, preface pg. xv), the original authors of the language, UML, the Unified Modeling Language, is "a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive systems".

UML is not a programming language but rather a visual modeling language. It is a language for creating models of systems. Models are regarded as simplifications of the reality that help us capture significant aspects of it abstracting away irrelevant details that add complexity but do not help in the same degree our understanding of the aspect of the system of interest.

As it is stated in the definition in the above, UML has four primary purposes to *visualize* software systems, which means to create a graphical representation that could easily express important aspects of a system; *specify*, in other words to communicate decisions about how the system should work, its structure and behavior, to the implementers of the system; *document*, to help communicate aspects of an implemented systems to future maintainers or users of the systems. Besides those three main purposes, UML also aims at serving for the direct *construction* of system. This thesis deals with UML for exactly of this purpose, that of by translation and code generation directly constructing systems.

UML was designed primarily for use in the development of software systems, but it has been used in other areas as well. Business modeling is one example of an area that has successfully made use of UML

## **2.3 Brief History of UML**

The development of UML started when Rumbaugh joined Booch in Rational Corporation towards the end of 1994. They each had their own software development methodologies called respectively OMT and Booch. They began working towards a unified method that would combine both methods. The first draft of this unified method was published in 1995. About a year later, Jacobson joined the other two authors in the same company. Jacobson enlarged the unifying initiative to incorporate ideas and notations from his own OOSE (Object-Oriented Software Engineering) method. As a result of this, the first draft of the then renamed Unified Modeling Language was published in the middle of 1996. Later, a consortium was created that included a number of other organizations that contributed to the release of the first official Unified Modeling Language specification. Rumbaugh, Booch and Jacobson admit that even though the original idea was theirs, they receive significant contributions from other partners. Besides, UML was influenced by other methods existing at that time.

In 1997 the version 1.1 of UML was submitted to the Object Management Group as a proposal for a standard for Object-Oriented modeling. It has been accepted and taken over by the Object Management Group ever since. The organization is now the responsible for the development of UML. A number of subsequent versions of UML have been published since then, adding for instance semantics and a language for specifying constraints. The current version of UML is 1.5, but a specification for UML 2.0 is expected to be released soon.

## **2.4 UML Goals**

UML was designed according to ambitious and broad goals. The initial goal of UML was obviously to unify the methods of its initial authors. One of its main goals is to be as simple as possible, but also it aimed at being sufficiently broad in scope and capabilities so as encompass as many aspects of practical systems as possible. It also aimed at being attractive to tool vendors in order to facilitate its adoption.

UML does not specify a complete software development methodology, although a UML-based methodology called Rational Unified Process was later unveiled.

## **2.5 UML Diagrams**

According to UML's vocabulary, the language is made up of:

- Things
- Relationships
- Diagrams

Things make up the nouns of the vocabulary. For example classes, use cases, collaboration, nodes and components. Relationships are the verbs. As the name implies, relationships connect things expressing and specifying the kind of relation they hold to each other. The kinds of relationship are dependency, association, generalization and realization Booch, Rumbaugh and Jacobson (1999, pg. 24) describe diagrams in the following way:

A diagram is the graphical presentation of a set of elements, most often rendered as a connected graph of vertices (things) and arcs (relationships).

We will discuss the things and the relationships in the context of the diagrams in which they commonly used. In the subsequent sub-sections each of the diagrams shall be described and discussed.

Diagrams can also be classified according to their major area of concern, or in other words according to the main aspects of the system they intend to describe in order to visualize, specify, construct and document these aspects. They can either be *Structural Diagrams* or *Behavioral Diagrams*. Structural diagrams capture the static aspects of the system, which are normally at least relatively stable over time. Behavioral diagrams, on the other hand, intend to capture the dynamic aspects of the system, meaning their changing parts, for instance, the flow of messages between parts of the system, the activities performed by a certain class or the response of a given class to external events.

The table below groups the 9 diagrams into their main concerns and provides a little description of the purpose of each of the diagrams:

Main Concern	Diagram	Description
Structural Diagrams	Class diagram	Illustrates the static design view of a system.
	Object diagram	Illustrates data structures, the static snapshots of instances of things found in class diagrams.
	Component diagram	The static implementation view of a system.
	Deployment diagram	The static deployment view of the system.
Behavioral Diagram	Use case diagram	Organizes the behaviors of the system.
	Sequence Diagram	Focused on the time ordering of message.
	Collaboration diagram	Focused on the structural organization of objects that send and receive messages
	Statechart diagram	Focused on the changing of a system driven by events
	Activity diagram	Focused on the flow of control from activity to activity

**Table 2-1 The UML diagrams**

The diagrams shall be discussed not necessarily in the order presented in the table above, rather they will be discussed in an order that resembles roughly the order in which they are usually crafted or their level of importance.

## 2.5.1 Use Case diagrams

Use case diagrams are commonly used to capture the requirements of the system. Use cases analyses resemble the analysis of functionalities in early structural modeling techniques. The difference is that while the analysis of functionalities tries to assess the functionalities that are required from the system, use cases represent the functionalities that are required from the system *from a certain user*. Use case diagrams provide a view of the system from the point of view of its external users or environment.

The basic building blocks or things that are present in the use case diagram are use cases and actors. Actors represent a consistent role played by an external user of the system. The roles can be played by human beings or external systems or components. The same person may act as different roles when using the system depending on the separation of tasks within the organization in which he performs his duties. Use cases specify the activities and flow of events following an interaction of the system with an external actor without specifying how this behavior is to be implemented. In the words of Booch, Rumbaugh and Jacobson (1999, pg. 219):

Use cases provide a way for your developers to come to a common understanding with your system's end users and domain experts. In addition, use cases serve to validate your architecture and to verify your system as it evolves during development.

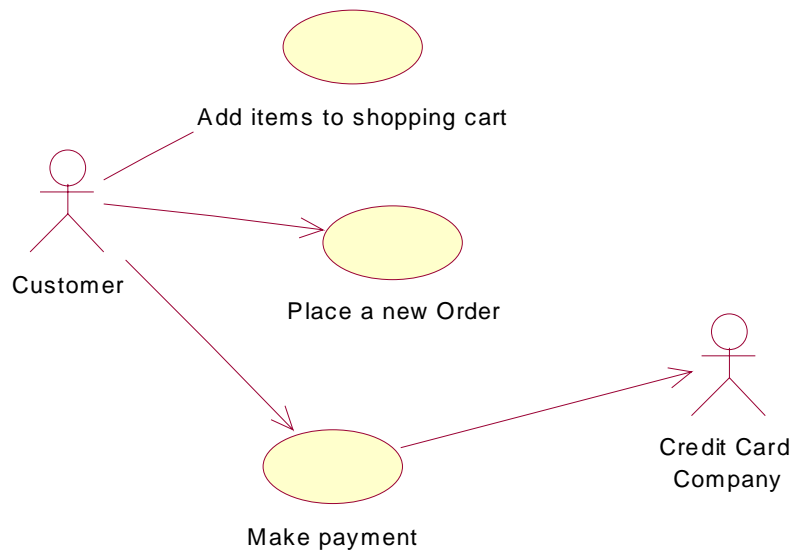
Use cases are represented as ovals with an unique name. Actors are represented as sticky figures resembling human beings as in the picture below:



**Picture 2-1 Graphical notation of use cases and actors.**

A graphical representation of a use case is usually coupled with a textual description of the flow of events and activities within the use case, explaining, for instance, pre- and post-conditions, the normal flow of events and exceptional flow of events. This description is usually kept outside of the diagram in other documentation artifact.

Arrows connect actors and use cases showing that there exists interaction between them. The direction of the arrows denote the direction of the interaction, whether from the actor to the use case, from the use case to the actor or whether it is bi-directional, happens in both directions. The figure below illustrates it:

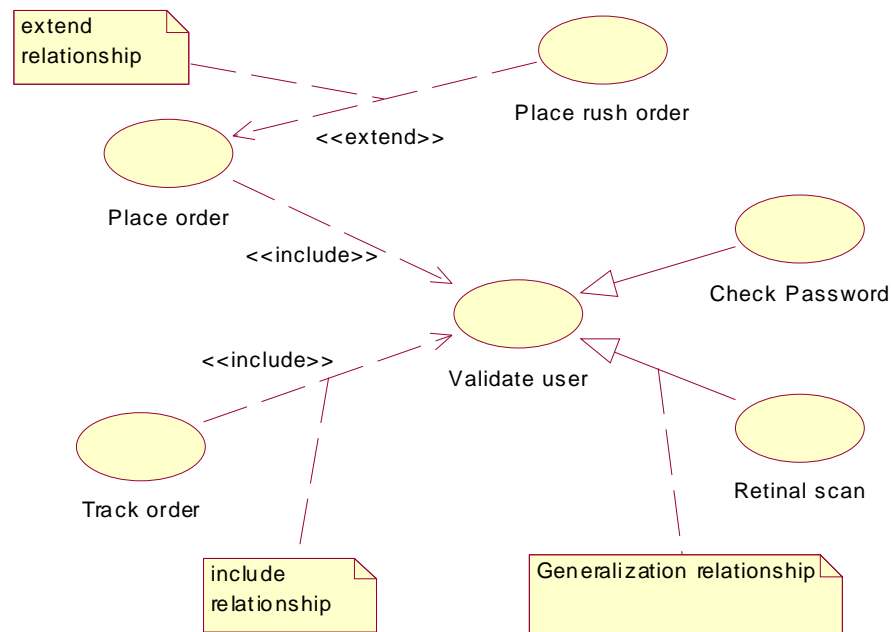


**Picture 2-2 Navigability between use cases and actors.**

Three types of relationship exist between use cases, namely a use case may include, extend or generalize another. Generalization among use cases is similar to generalization among classes, which shall be explained in the following section. A specialized use case inherits the behavior and meaning of its base (more general) use case and may override some these or add new ones. According to the definition provided by UML's original designers Booch, Rumbaugh and Jacobson (1999, pg. 227 - 228):

An include relationship between use cases means that the base use case explicitly incorporates the behavior of another use case at a location specified in the base. (...) An extend relationship between use cases means that the base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case.

Below is an example of a diagram that depicts relationships among use cases:



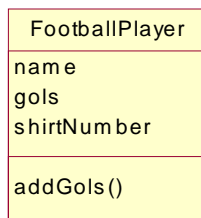
**Picture 2-3 Example diagram showing relationships among use cases.**

### 2.5.2 Class diagrams

Classes are perhaps the most important object oriented building blocks. According to Booch, Rumbaugh and Jacobson (1999, pg. 48): “a class is an abstraction of the things that are part of our vocabulary”. In other words, a class represents a blueprint for a set of objects that share common characteristics and behavior.

Classes have attributes and operations. Attributes function as specifications of a property common to all instances of the class; values of instances of the property must be within the range and conform to the data type specified by the property. Or, in the words of Booch, Rumbaugh and Jacobson (1999, pg. 50): “An attribute is a named property of a class that describes a range of values that instances of the property may hold” . Operations are like implementation of services offered to the outside world that are used to affect the behavior of the class. In other words: “an operation is an abstraction of something you can do to an object and is shared by all objects of that class” (Booch Rumbaugh and Jacobson 1999, pg. 51).

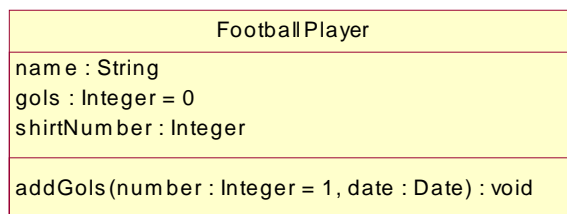
In the UML, classes are represented as a rectangle usually containing three compartments. The upper one containing a unique name for the class, the middle-compartment should hold the attributes, while the third compartment lists the operations for the class, as in the picture below:



**Picture 2-4 A class with attributes and an operation**

The name of an operation should be followed by an opening and a closing parenthesis. If the operation has arguments they may appear enclosed by this pair of parentheses. If not, the content inside the parentheses should be empty. If the operation has multiple arguments, they are separated by commas. It's not mandatory to display the arguments of the operation in the diagram. This information can be chosen to be elided for convenience.

The data types of the attributes as well as those of the arguments of the operations and the operations' return types may appear in the class notation. The type of an attribute is written after the attribute name separated by a colon. Similarly, the data type of an argument of an operation is written after the name of the argument and a colon. The return type of an operation is written after the operation name and its arguments. The picture below illustrates it:



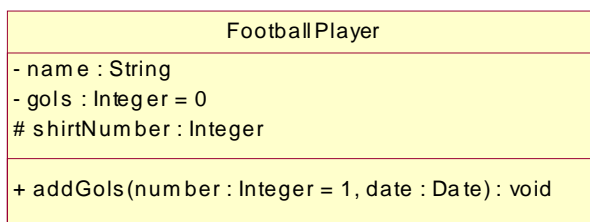
**Picture 2-5 A class with attributes and an operations. Data types are shown for the attributes, arguments of the operation and the return type of the operation.**

The class named *FootballPlayer*, depicted in the picture above, has three attributes: *name*, *gols* and *shirtNumber*, whose data types are respectively: *String*, *Integer* and *Integer*. The operation *addGols* has two arguments, namely: *number* and *date*. The data types of these arguments according to the picture above are *Integer* for the argument *number* and *Date* for the argument *date*. The return type of this operation is void.

One can also specify default values for attributes and arguments using the UML notation. For instance, in the picture above, the default value for the attribute *gols* is 0

and the default value for the argument *number* of the operation *addGoals* is specified to be the integer value 1.

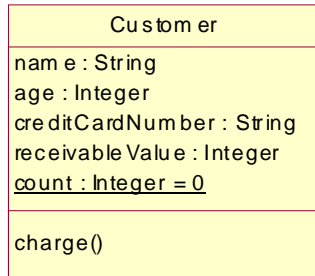
An important detail that can be added to graphical notation of a class is the visibility of its attributes and operations and its scope. Three kinds of visibility can be specified in the UML. The attribute or operation may be private, in which case it is only accessible to objects of the same class, it may be declared to be protected, i. e., accessible to objects of the same class or to objects of one of the sub-classes of the class, or it may be public, meaning that no restriction exists regarding its accessibility, in other words, they could be accessed by objects of any class. The notation is: private features are pre-pended with a minus sign “-“, protected ones with a pound “#” and public features with a plus sign “+”. In object-oriented design, it’s common that all attributes be declared private or protected and most interesting operations be publicly exposed. The picture below shows a class that has two private attributes, a protected one and a public operation.



**Picture 2-6 Class with visibility symbols.**

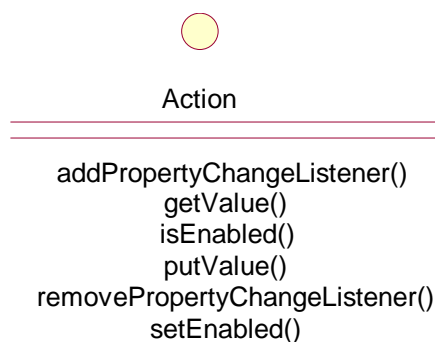
Another important aspect of the operations and attributes of a class is that of scope. Most attributes and operations are scoped in the instance of the class, meaning that operations act upon and attributes specify properties that can assume independent values for each instance of the class. The instance of the attribute thus belongs to the instance of the class not the class itself. A change in the value of an instance-scoped attribute is not propagated to other objects of the same class; attribute instances exist in isolation even though they follow a similar specification. On the other hand, attributes and operations may be defined to be scoped in the class itself, in this case there should exist a single instance of these attributes that are shared by all objects of the class. A change in the value of these attributes is reflected in all objects of the class. Class-scoped operation act upon class-scoped attributes only and they can be called from the class itself without the need for any instance to be set-up. Class-scoped, or *static* according to the object-oriented jargon, features are distinguished from more commonly used

instance-scoped features in the UML class notation by underlying the features' name and data type. For example, the attribute count of the class below is specified to have class scope, as the count of customers is increased all instances of Customer will read the same value from the attribute:



**Picture 2-7 Example of a class-scoped attribute**

There's a special kind of class that is extensively used in modeling system that is called interface. Differently from common classes, an interface has no attributes or implementation. It merely contains a set of empty operations that other classes or classifiers are to implement. An interface works as a contract between two classes, one of them requests the services specified in the interface and the other is responsible for carrying them out. The realization relationship between an interface and a class that implements its services will be discussed shortly below. Interfaces can be rendered as a stereotyped class (stereotypes will be discussed later) or more commonly they are rendered with a named circle, the operations that the interface specifies may be shown below the circle. Interfaces may also contain constants, which are rendered in a compartment above the operations, as if they were attributes. Below is an example of how an interface would be rendered; the interface shown is the Action interface of the java swing framework.



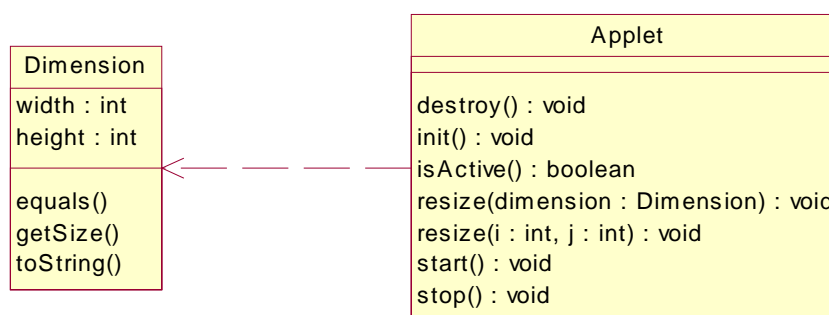
**Picture 2-8 Example of an interface: javax.swing.Action**

Rarely do classes exist in isolation. Classes are usually related to other classes. The relationships between classes and their specifications are important aspects of the system. Some authors even contend that the relationships among classes are more interesting than the definitions for the classes themselves and are more expressive of the subject matter's rules, in the words of Mellor and Balcer (2002, pg. 106):

It's a pity that "object-oriented" is such a popular term, it's the domain relationships that truly define the rules for a domain.

There are basically four types of relationship in the UML: dependency, generalization, association and realization. They will be explained in the following paragraph.

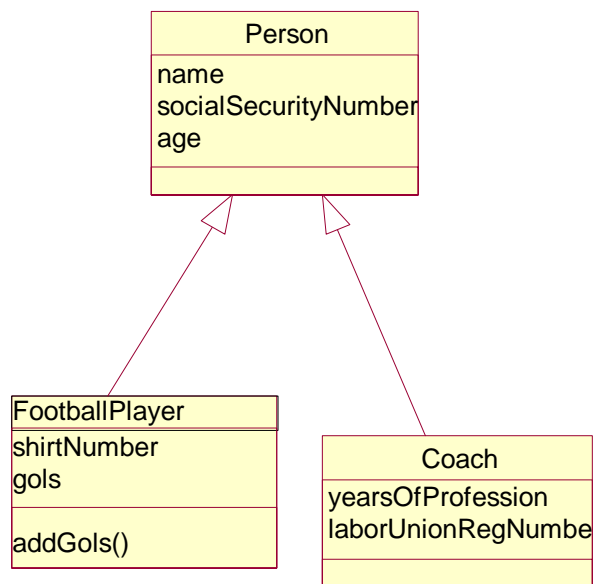
Dependency means that one class depends on the definition of another class for some of its behavior. If the definition of the class that is depended-upon changes that may affect the class that has a dependency relationship to it. A typical case of dependency relationship is when a class uses the definition of another class as an argument or return value of some of its operation. A dependency relationship is graphically rendered as dashed line directed to the class or thing that is depended upon. Below we show an example that is drawn from the java AWT framework. The class *Applet* uses the class *Dimension* as the argument of one of its operations. It's correct to say that the class *Applet* depends upon the definition of the class *Dimension*, if the definition of this class changes, this may affect the class *Applet* that makes use of some the class's data and services in one of its operations.



**Picture 2-9 Example of a dependency relationship between classes**

In a generalization relationship, a class specializes the features and behavior of another more general class. Object of the class that specializes can be used in the places where objects of the general classes are expected. This is because the specialized class inherits the features and behavior of the general class, it may add more features and behavior, it

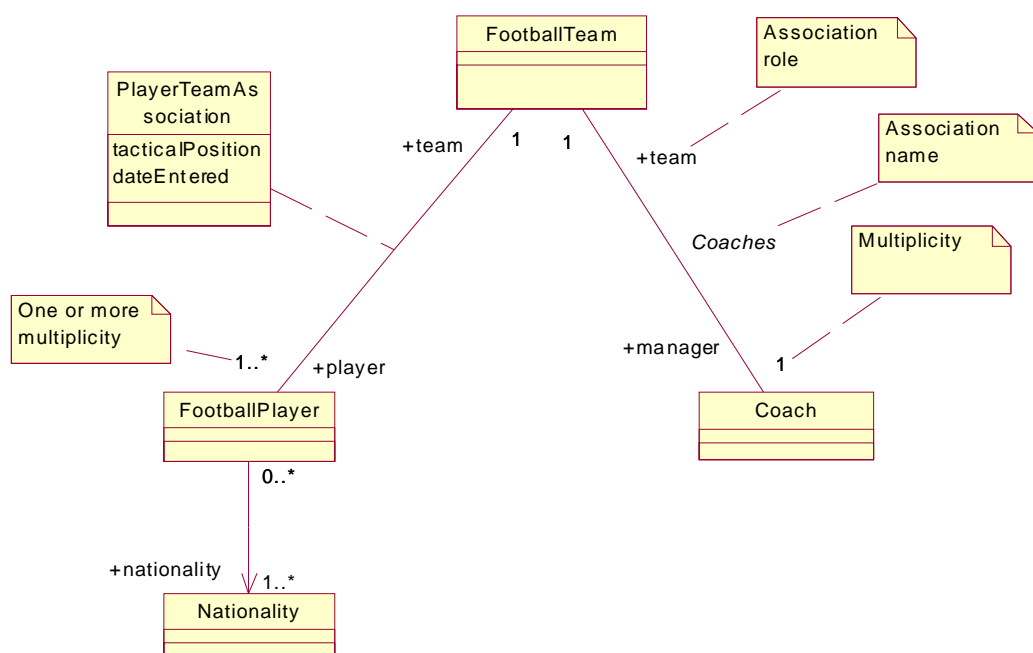
may also change the implementation of some features, nevertheless it can be assumed to hold structural similarities to the general class. That's why a generalization relationship is sometimes said to be a is-a relationship between classes, a specialized class is a kind of a more general class. Generalization is also sometimes referred to as an inheritance relationship, as the specialized class inherits features and behavior of the general class. The graphical notation for a generalization relationship is solid line with a large open arrowhead at one edge pointing to the general class. Below is an example of the graphical rendering of a generalization relationship, notice that *FootballPlayer* and *Coach* are each specializations of the *Person* class; they inherit all the attributes from this class without having to declare them.



**Picture 2-10 Example of generalization relationships**

The third type of relationship is called association. Association is a structural relationship between classes. An association relationship specifies that a class is navigable from another, in other words objects of one class are aware of the existence of objects of another class and may access their data, request services and influence behavior. An association relationship between two classes can be navigable in one or both directions. If it specifies a bi-directional navigability in the association, which means that both classes that participate in the relationship are aware of each other. In the case unidirectional navigability, only the objects of one class are aware of the objects of the other class. Associations may have a defined multiplicity, meaning that the number of objects of each class that can be linked through the association can be specified. Aggregation and Composition are types of association that have specific semantics.

Aggregation means that the relationship between the classes is a whole-part relationship, i. e., an object of a class contains objects of other classes that can be thought of as “parts” of this object. Composition is a special type of aggregation, in which the “whole” class also controls the life cycle of its “parts”, if an object of “whole” class ceases to exist; the objects of its “part” classes that are linked to it should also have their lifecycles terminated. Associations between classes can also be defined in a class that is called association class. This happens when certain properties are better understood as being part of the association than as being part of any of the participating classes. Associations are rendered as a solid line between two classes; an arrow can indicate the direction of navigability in case it happens in one direction only. The roles that each class plays in the association can be written at each end of the association together with the multiplicity for the association-end. Association may also be given a name that serves to describe and communicate the purpose of the association. In the picture below, one can see an example of a class diagram showing associations between classes with some role names, associations’ names and multiplicities specified. The picture also shows an example of an association class.

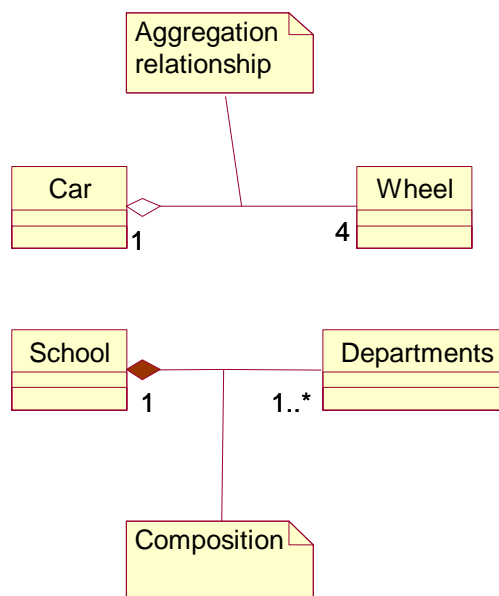


**Picture 2-11**Associations and an association class

Namely, *PlayerTeamAssociation* is an association class that contains properties that are better understood as belonging to the association between *FootballPlayer* and

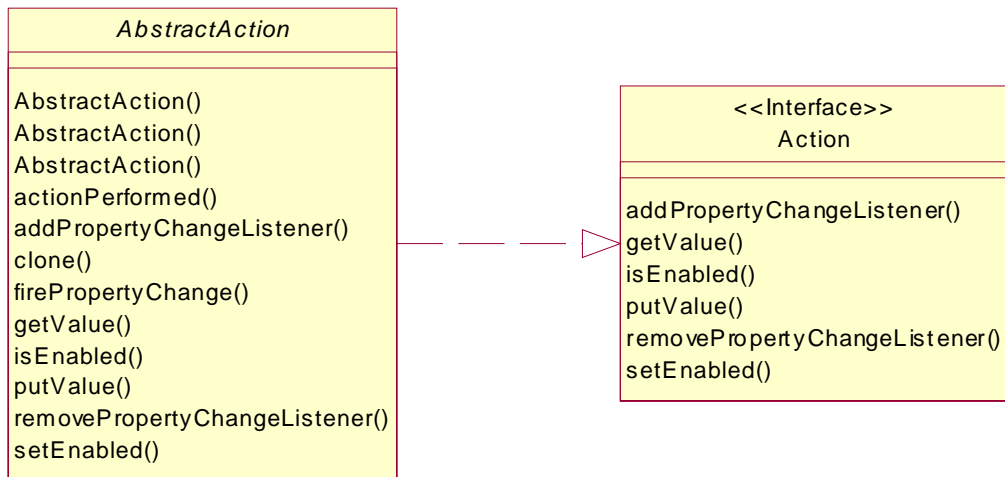
*FootballTeam* than to any of those classes, such as the position of the player in the team.

The picture below shows the rendering for an aggregation association and a composition association. There is a clear “whole-part” relationship between a car and its four wheels that justifies the aggregation relationship between them. The relationship between a school and its departments is even stronger since if the school ceases to exist so will its departments.



**Picture 2-12 Aggregation and composition relationships**

Realization is the last kind of the relationship. The realization relationship happens between an interface and a class. The class implements, or realizes, the operations specified in the interface. Objects of the class can be used in places where the services where an implementation of the interface is necessary in order to make use of the specified services. The graphical representation of realization relationship is a mixture between the notations for the dependency and the generalization relationship. It consists of a dashed line with a large open arrowhead directed from the class to the interface. Below is an example of the notation for a realization relationship. It graphically shows the relationship between a java swing Action interface and one of its implementing classes. Notice that the interface was rendered using a label denoting the stereotype. The label says “interface” and is enclosed between guillotines.

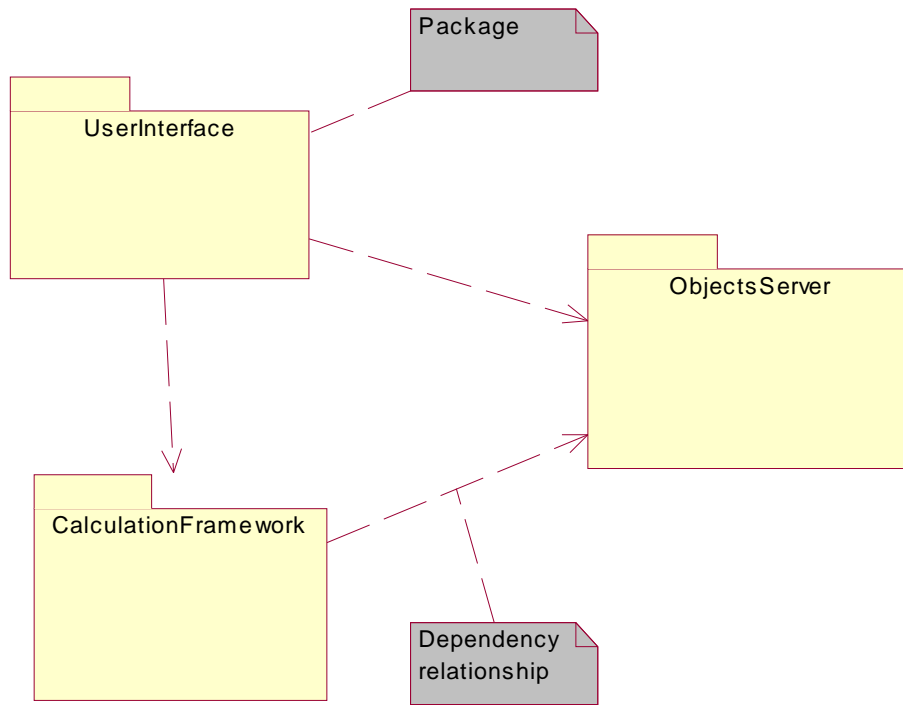


**Picture 2-13 Realization relationship between a class and an interface**

### 2.5.3 Package diagrams

Packages are grouping things. As the model grows bigger it is useful to group some of the things that belong to the model into more manageable chunks. These chunks may have names and may be thought of as a higher level of abstraction. Packages are thus groups of elements that are useful to organizing the model. These elements should preferably hold semantic closeness and be more cohesively related to one another than to elements outside the package. In the words of Booch, Rumbaugh and Jacobson: “well structured packages are therefore loosely coupled and very cohesive, with tight controlled access to the package’s contents” (1999, pg. 169).

Packages may contain other nested packages and a great of number of other modeling constructs including classes, interfaces, components, nodes, diagrams and even other elements. Graphically, a package is drawn as named tabbed folder. When the elements of a package use or in any way are related to elements of another package, the package is said to be dependent upon this other package. In other words, a change in the definition of the elements in the package that is dependent upon may affect the dependent package. A package diagram shows package and the dependency relationships between them. The package diagram may display all or some of the objects that belong to the package, though it’s not necessary. It is also possible to show the visibility of the elements inside the package, some elements are visible only to other elements of the package, others are public. Below, one can see a package diagram.



**Picture 2-14 Package diagram**

Package diagrams are usually drawn at earlier stages of software development and commonly each package corresponds to a subject matter and is commonly assigned to a separate team of developers.

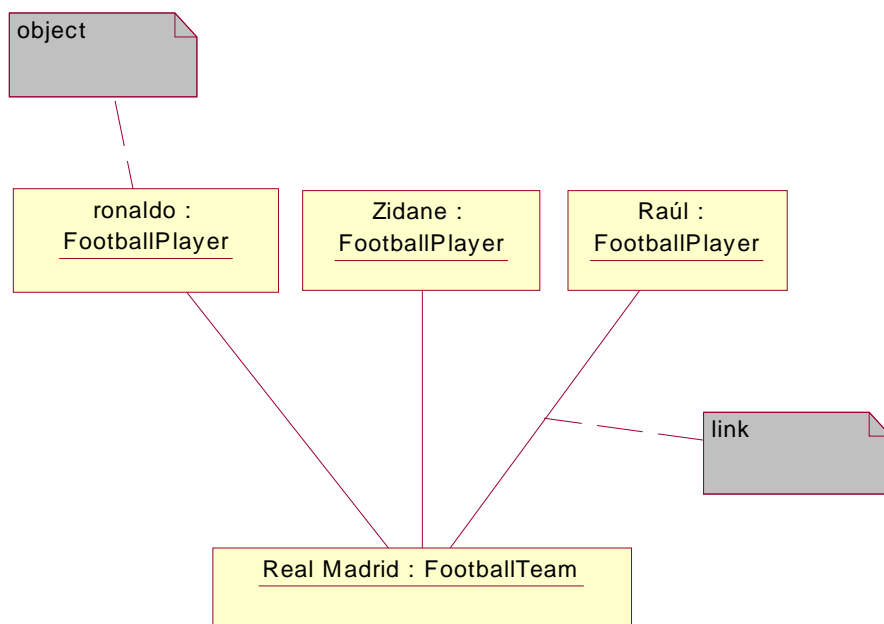
#### 2.5.4 Object diagram

Class diagrams provide the structural view of the system. Each class works as a blueprint from where objects are constructed. Associations and other kinds of relationship also specify the kinds of link that may exist between objects at run time.

For each specification of classes and their relationships there's usually an almost infinite possible set of configurations of object states and links at run-time. It's thus not normally useful to model the states of objects and links at run-time. There are some cases though when this might be justified, especially in situations where providing a snapshot of a possible configuration of objects and links among them is useful for gaining or communicating some insights about the system. For this reason, the UML provides an object diagram as well as a class diagram.

In the UML, objects are drawn as labeled rectangles. Objects may have a name. If they do, the label that's shown in the object representation should contain the object name and the class type of the object separated by a colon. Objects, on the other hand, may

lack a specific name in which case only the class of the object is shown after a colon. The whole label is usually underlined. The attributes of the object and their values (state) may be shown in a separate compartment below the object label. Links are shown as a straight line between objects. No multiplicity is allowed for links. Object diagrams should normally be consistent with the classes and relationships specified in a corresponding class diagram. They may however show a snapshot of a point in time when the system was in an unstable state during which some of the structural rules may be temporarily violated, such as in a method call. A picture of a object diagram follows. This picture shows objects of the classes specified in pictures 2-10 and 2-11.



**Picture 2-15 Object diagram**

### 2.5.5 Interaction diagrams

Interaction diagrams basically show how a set of objects interacts in order to accomplish a purpose. Modeling interactions is part of the dynamic modeling of systems.

There are two types of interaction diagrams that are semantically equivalent, which means that a diagram of one type can be transformed into a diagram of the other type with no loss of meaning. But they emphasize different aspects of the system. A collaboration diagram is one type of interaction diagram and emphasizes the structural relationships, or links, between objects. By drawing a collaboration diagram it is easy to visualize the structural relationships among the objects. Sequence diagrams, on the

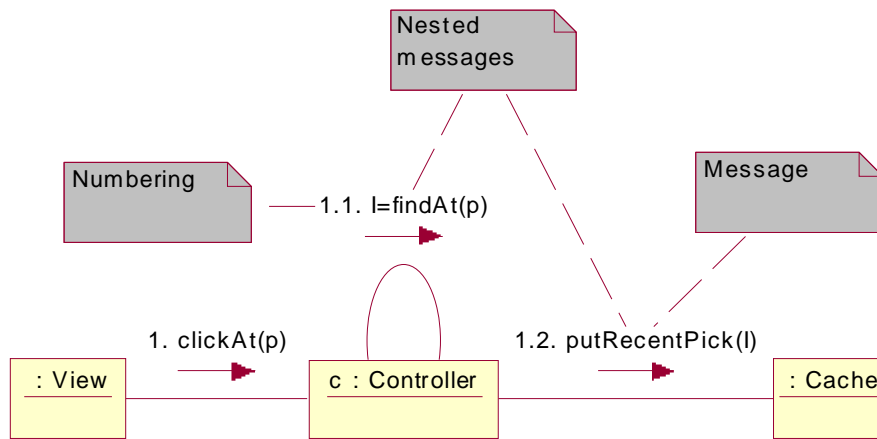
other hand, emphasize the time ordering of messages between objects. With a sequence diagram, it's easier to visualize the sequence of messages in a time scale.

A message usually abstracts one of the following actions:

- Operation call
- Return value from
- Send a signal to an object
- Create an object
- Destroy an object

A collaboration diagram may be thought of as an object diagram with the specification of the dynamic behavior by showing a sequence of messages between objects in the diagram. Arrows pointing from the sending object to the receiving object, located above or below links among objects model the exchange of messages between objects.

Message names are drawn close to this arrow to show the messages that are passed between the objects. Before the message name, a sequence number is usually indicated. From the sequence number it is possible to deduct the time ordering of the exchange of messages. Sequence numbers are also used to indicate a nested flow of control. This happens when the exchange of messages between objects causes the receiving object to send a message to another object. This second message is said to be nested inside the flow of control of the first message. Nested messages are indicated by showing a sequence number, which consists of the sequence number of the original messages, plus another number showing the sequence of nested messages (there may be more than one message nested in the flow of control of the originating message) separated by a dot. Nested messages may contain other nested messages in which case the sequence number will contain as many numbers as the nesting depth separated by dots. There are cases when the passing of message between objects will also signify the permanent transference of control from the sending object to the receiving object, in this case the sequence numbers of the messages originating from the second object will only increment the sequence number of the first object with no sub-indices. Below is an example of the rendering of a collaboration diagram.

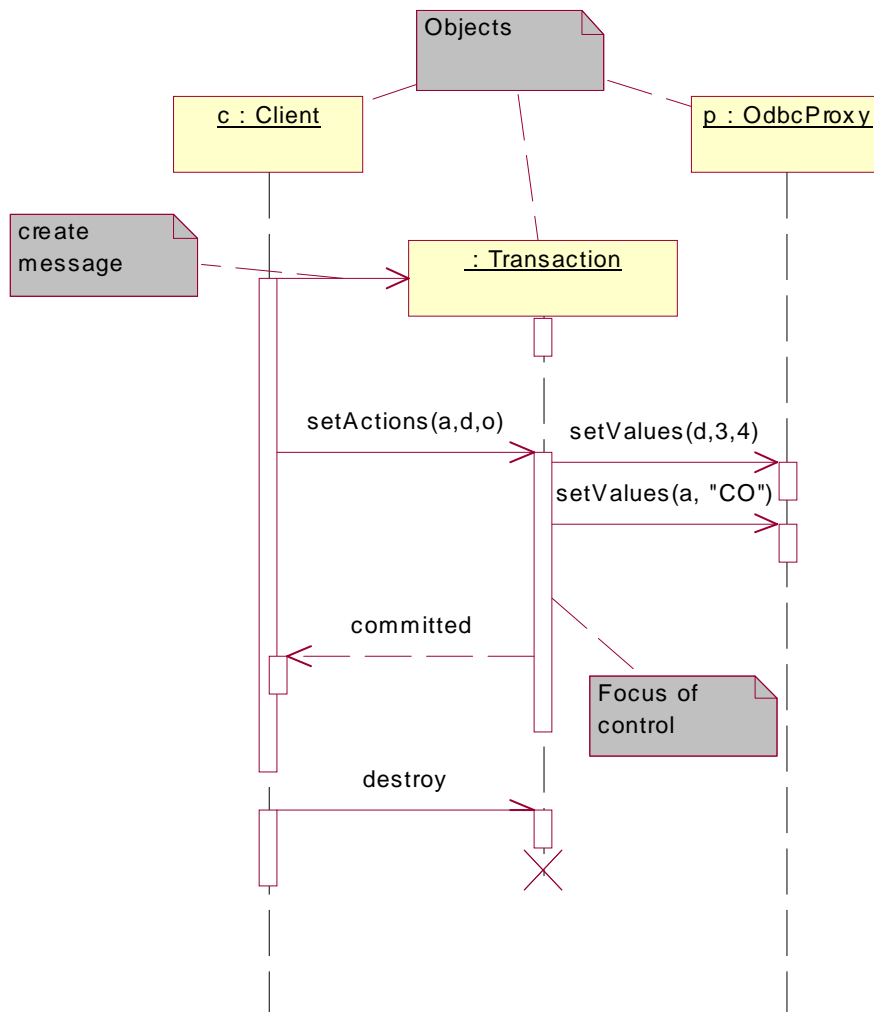


**Picture 2-16 Collaboration diagram**

As mentioned before, the sequence diagram is analogous to the collaboration diagram. The main difference is that while the collaboration diagram emphasizes the structural relationships between the objects, the sequence diagram displays in a more intuitive manner the time ordering of messages.

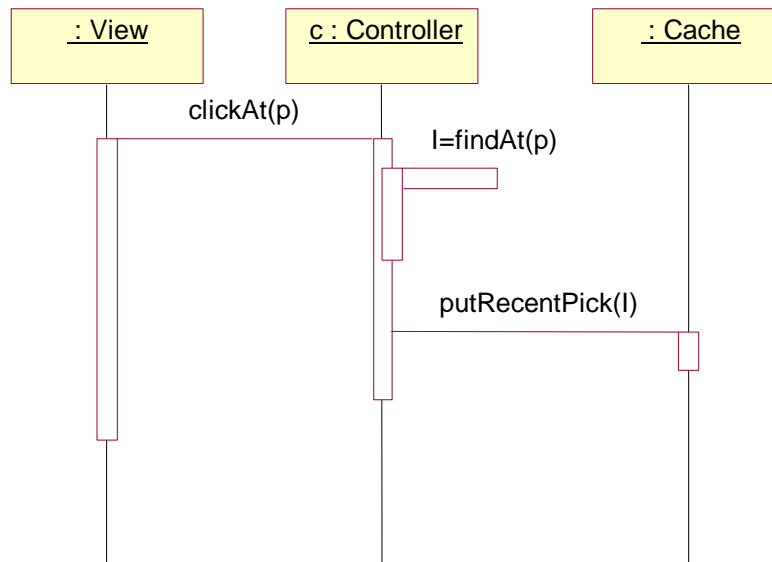
In a sequence diagram, objects are placed at the top, with the exception of objects that are created from a message of another object in the diagram. The lifeline of a object is represented as a vertical dotted line originating from the object representation. If the dotted line is interrupted, it signifies that the lifeline of the object has reached an end. Messages are shown as horizontal arrows from a point in the lifeline of the sending object to a point in the lifeline of the receiving object. The vertical axis of the diagram represents the time dimension of the exchange of messages or collaboration, and there lies the main difference to a collaboration diagram. Messages that appear higher in the vertical axis closer to the representation of the object precede in time messages that appear lower in this axis. The sequence numbers of messages may also be shown in the diagram, similarly to what happens in a collaboration diagram, even though it is not strictly necessary since it's possible to deduce the time ordering of messages from their positions in the vertical axis of the diagram. Sequence diagrams also display the flow or focus of control. The focus of control is represented as a thin tall rectangle in the lifeline of the object who holds the focus of control. The Return messages may appear at the bottom of the focus of control and are represented as a dotted arrow. The representation of return messages may be omitted. When objects create another message through the passing of a create message, the newly created objects is drawn at the end of the message arrow and vertically aligned with it, differently from other objects, whose

representations are drawn at the top of the diagram. Below is an example of a sequence diagram, extracted from Booch et al (1999, pg. 247).



**Picture 2-17 A sequence diagram**

As another example, we will show here the sequence diagram equivalent of the collaboration shown in picture 2-16.



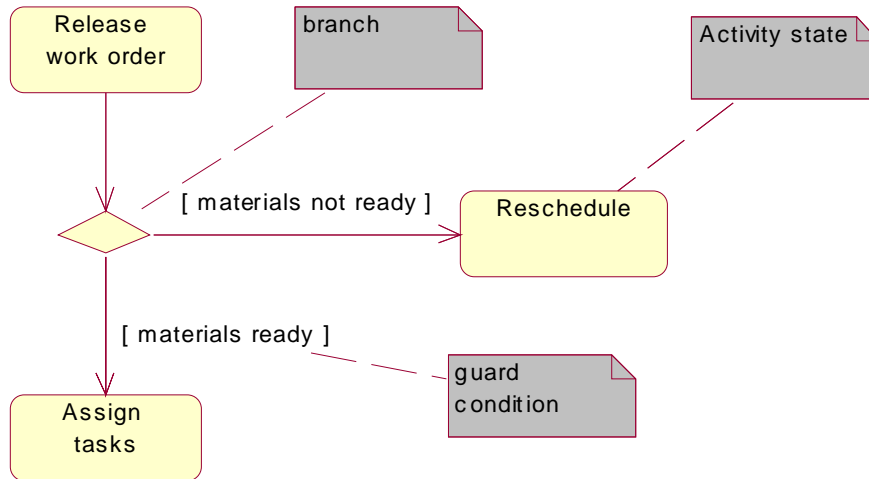
**Picture 2-18 Another example of a sequence diagram**

### 2.5.6 Activity Diagrams

Activity diagrams can be considered heirs to the old-styled flowcharts that were commonplace in structural modeling, which preceded object-oriented modeling. In spite of its well-known similarities to the traditional flowcharts, activity diagrams add some innovative concepts that were not present in flowcharts used for structural modeling of software systems. Flowcharts did not possess, for instance, the activity diagrams' mechanisms for the modeling of concurrent activities, namely forking and joining that will be discussed later. Flowcharts assumed a sequential flow of control. In the words of Booch, Jacobson and Rumbaugh (1999, pg. 81) "an activity graph is like a traditional flow chart except it permits concurrent control instead of sequential control – a big difference".

Like most other diagrams, the activity diagram is a collection of vertices and arcs. Vertices are activities and arcs are called transitions. Activities are finite computations made up of one or more atomic computations that are named *actions*. Upon completion of an activity the flow of control is usually passed to the following activity except when a branching is performed, in which case a choice is made depending on a set of values derived from the state of the system of which transition to follow. Usually many transitions originate from a branching state, each of them has a *guard condition* attached, the system will proceed to the activity to which the one transition that has its guard condition satisfied leads. It is assumed that only one the transitions originating from the branching state will have its guard condition satisfied at any one point in time,

otherwise the behavior of the system will be unspecified and the choice of the transition will become arbitrary. Below is an excerpt of a possible activity diagram that has a branch. This example was taken from Booch, Jacobson, Rumbaugh (1999, pg. 263).

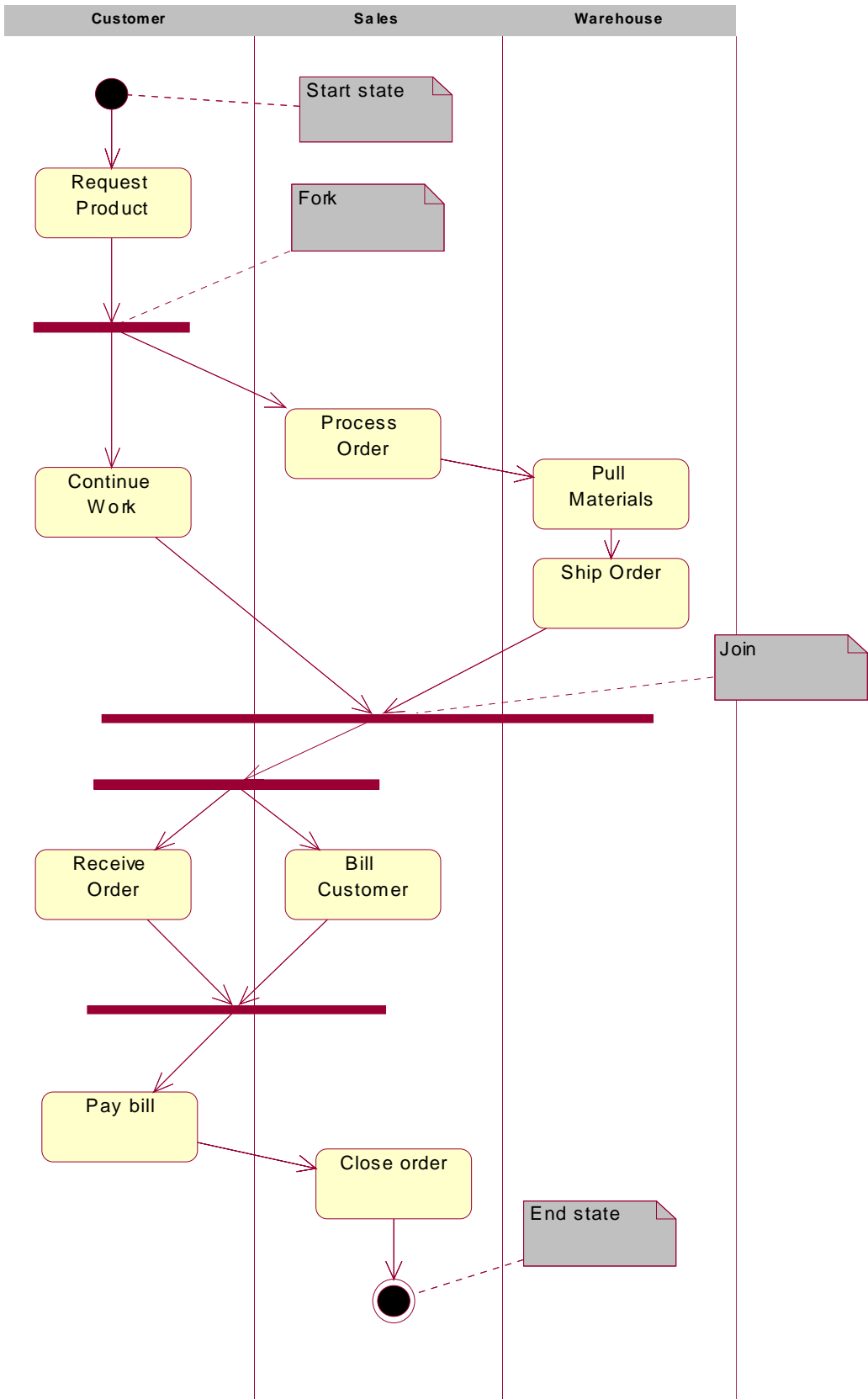


**Picture 2-19 Activity diagram with a branch**

As said before, the main innovation that activity diagrams offer in comparison to their predecessor structural flow charts is to allow for the modeling of concurrent control, or the concurrent execution of activities. This is done by forking and joining. *Forks* are pseudo-states from where many transitions can originate. These transitions will lead to activities that will be performed concurrently. *Join*, on the other hand, is a pseudo-state where many transitions are synchronized and unified into one that originates from the join. Usually, a join will wait for all the transitions that come into it to be completed before it proceeds with the transition that follows from it, but the behavior of the join can be specified to proceed with outgoing transition just after the arrival of the first transition. Forks and Joins are visually represented as thick horizontal lines.

In the real world, or in a computational system, activities may be performed by different entities or objects in the same flow of control. For instance in a business process, it's common that after the completion of an activity by some department the flow of control is transferred to another department that will be responsible for the execution of another activity. This is modeled in UML as swim lanes. Swim lanes are an area bound by two vertical lines with a name attached at the top, activities that lie within the vertical lines that comprise the swimlane are performed by the entity abstracted by it. Each activity belongs to only one swimlane, even though transitions may span more than one

swimlane. Below is an example of an activity diagram that includes forks, joins and swimlanes. Another noteworthy feature of the diagram below is the presence of start and end states, those are pseudo-states that signal the start or end of the activity flow.



Picture 2-20 Activity diagram

Activities diagram are not of great importance to executable UML. Although they provide a reasonably clear specification of an execution path, the choice was made to specify the behavior of classes by state machines in Executable UML. State machines will be explained in the following sub-section. On the other hand, Mellor and Balcer (2002, pg. 51-52) do recommend that activity diagrams be used to specify the sequence of use-cases at an earlier stage of the development process. Activities diagrams have also been used extensively for modeling business processes. Although they have been designed primarily to describe computational workflows, they have also been proven suitable for the description of workflows in the context of organizations (see Eriksson and Penker (2000)), in spite of some notable limitations (see Ambler (2002) for a critical view of the use of activity diagrams for business-process modeling).

### 2.5.7 State-chart diagram

State-chart diagrams are of utter importance for Executable UML. As will be seen later, the preferred method of modeling the dynamic behavior of classes in Executable UML is by modeling their state machines.

The statechart diagram is the graphical representation of a state machine. Booch, Rumbaugh and Jacobson (1999, pg. 290) define state machines and states in the following way:

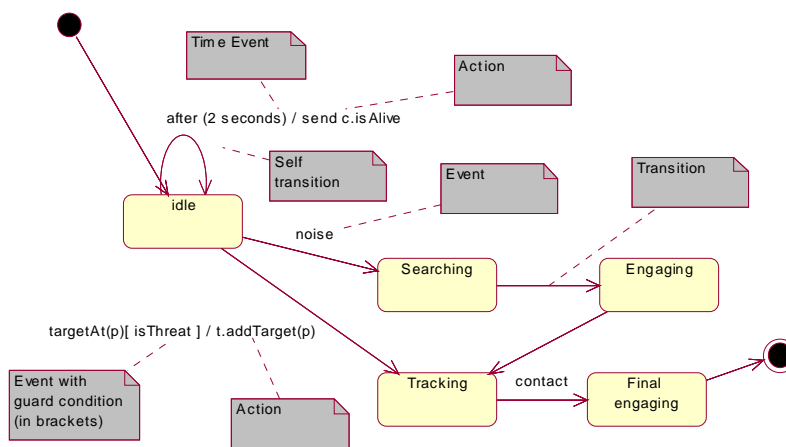
A *state machine* is a behavior that specifies the sequence of states an object goes through during its lifetime in response to events, together with its response to those events. A *state* is a condition or situation during the life of an object during which it satisfies some condition, performs some activity or waits for some event.

In other words, the state machine displays the states that a certain object might go through during its life cycle and how the object may transition from one state to another in response to certain events. A typical event that may cause the transition of states is the receipt of an external asynchronous signal sent from another object or entity. Even though synchronous method calls can also be modeled as events in a state machine. Other kinds of event may be the passage of time or a programmed change in value due to activities performed in a state.

When the object is in a certain state, some events may cause to change to another state (or a *transition* of states). On the other hand, the transition to another state may happen

only if a certain condition is met. This condition is called a *guard condition*. Similarly to what happens in a branch of an activity diagram, a guard condition may be attached to an event transition, signifying that the transition will occur only if the condition specified is met.

After the transition is fired up, an action in response to the transition will commonly be executed. An *Action* is defined as an executable atomic computation that results in a change in state of the model or the return of a value (Booch, Rumbaugh and Jacobson, 1999, pg. 290). Each transition may have a corresponding action specified that will be executed when the transition fires up. Similarly, each state may have one entry and one exit action specified that will be executed upon entry to and exit from the state, besides actions that will act in response to internal transitions (or transitions that occur without change in state). The entry action of a state is executed independently of the transition that was traversed during the change in state. If this transition specified an action, this action will be executed first, before the entry action of the target state. Below is an example that shows the statechart diagram's basic constructs from Booch, Rumbaugh and Jacobson (1999, pg. 294).

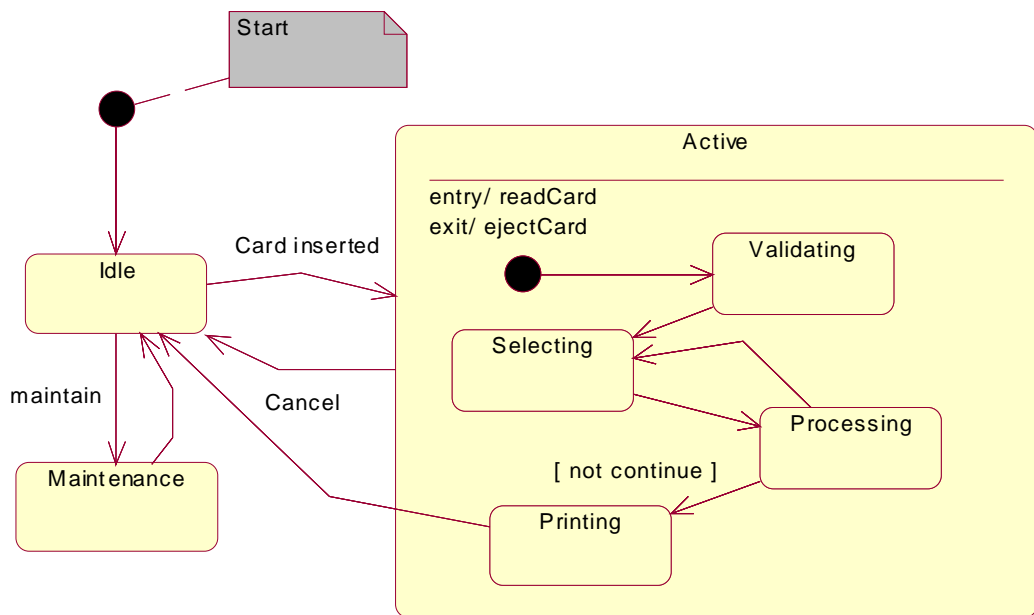


**Picture 2-21 Statechart diagram**

The diagram below above shows an example of a self-transition, or a transition that occurs from one state to itself. In the paragraph immediately above the diagram, we mentioned the possibility of internal transitions that occur inside the state. Though seemingly alike, self-transitions and internal transitions have an important difference. A self-transition causes an exit from the state, meaning that if an exit action is specified for the state it'll be executed, and upon return to the state the entry action (if specified)

will also be executed. Internal transitions do not cause exit or re-entry to the state, therefore no exit or entry actions are dispatched.

States may contain other states nested inside, meaning that while in the state the object will also alternate between the state's sub-states. Other in other words, the state contains a nested state machine that will be executed upon entry to the state. Typically, a composite state with nested sub-states contains a state machine only, but the state can also be specified to contain two or more state machines that will execute concurrently. The picture below has a state with sub-states. Note that the entry and exit actions will always be executed upon entry and exit in the composite state. You can also see a transition from a nested state straight to another state in a higher level, which is allowed. Alternatively, an end pseudo-state could be specified in the nested state machine, in which case the execution flow would proceed from a triggerless transition from the composite state to another state in the upper-level state machine or the composite state would be kept waiting for events.

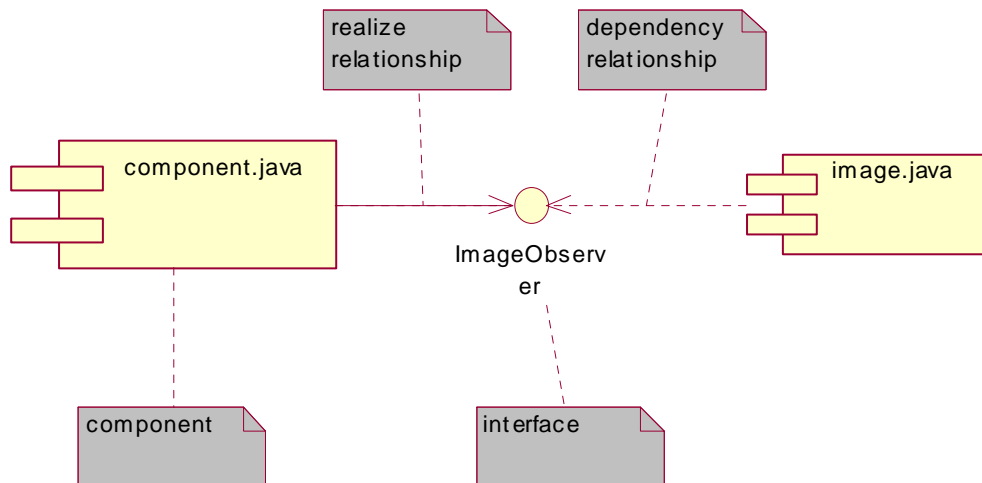


**Picture 2-22 Statechart diagram with a composite state (Booch, Rumbaugh and Jacobson, 1999, pg. 299)**

### 2.5.8 Component diagram

Component diagrams are not of great importance to executable UML. As will be seen later, the executable UML methodology tries to separate platform independent concerns from platform dependent ones. The executable UML model will typically deal with

platform independent concerns that will be translated to a platform dependent model by already existing (or developed in parallel) model-compilers. Component as well as deployment diagrams belong to the realm of platform dependent concerns because they usually involve the description of physical characteristics of the model that are by definition bound to a certain platform. They are thus of use only to the manufacturers of model-compilers who will deal with the platform-dependent aspects of the process and of practically no use to the developers of platform-independent executable UML models. According to Booch, Rumbaugh and Jacobson (1999, pg. 343) “a component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. [...] A component typically represents the physical packaging of otherwise logical elements, such as classes interfaces and collaborations”. In UML, a component is rendered as rectangle with tabs. A component commonly will realize one or more interfaces and may depend upon interfaces realized by other components or upon other components themselves. Below is an example of a typical component diagram, the relationship between a component and an interface that it realizes is represented in a way similar to the relationship between a class and interface that it implements. Note that although in the simple diagram below the interface was shown in its iconic form using its full stereotyped class notation, which could also display all the methods and constants belonging to the interface.



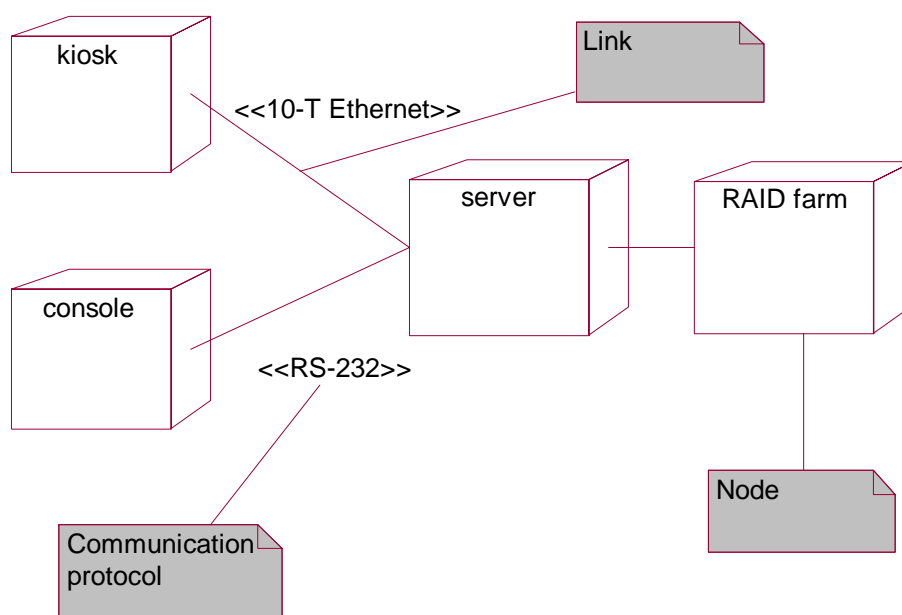
**Picture 2-23 Example of a simple component diagram (Booch, Rumbaugh and Jacobson, 1999, pg. 348)**

As it can be seen on the picture above, the notation for a dependency relationship between a component and a interface or another component is similar to the notation for dependency relationships in the class and package diagrams.

## 2.5.9 Deployment diagrams

The deployment diagram basically shows the hardware topology of the deployed system. A deployment diagram is a collection of *nodes* and *links*. A node is a basically piece of hardware with computational capabilities. Links are physical connections among nodes, such as an Ethernet connection or a shared bus. As mentioned before, deployment diagrams are not used in executable UML, because the executable UML methodology tries to separate platform independent concerns from platform dependent ones, to which the hardware topology surely belongs.

In the UML, nodes are represented as shallow cubes. Links are represented as non-directional associations possibly with a stereotype indicating the kind of protocol that is used for communication through the link. The components that are deployed on each node may also be displayed in the diagram. The picture below shows an example of a deployment diagram from Booch, Rumbaugh and Jacobson (1999, pg. 363).



**Picture 2-24 Deployment diagram example**

In the deployment diagram more than in others, it is common the use of stereotype icons for rendering in order to make the diagram more expressive and perhaps understandable even for the layman. Stereotype icons for nodes such as mobile phones, desktop computers and servers make the diagram much more intuitive and aid enormously in the communicating hardware topology decisions. Stereotypes and other extensibility mechanisms shall be discussed in the next section.

## 2.6 Stereotypes and Extensibility Mechanisms

Even though the UML can be regarded as a quite comprehensive language, it was designed to solve a wide range of problems and has a wide array of constructs and other expressible mechanisms, it could never be able to provide sufficient modeling capabilities to model all kinds of problems that may appear in various domains. For this reason, the UML was designed to be extensible from the outset, with some extensibility mechanisms provided for common use.

One of these extensibility mechanisms is the use of *stereotypes*. Booch, Rumbaugh and Jacobson (1999, pg. 29) define a stereotype in the following way:

A stereotype extends the vocabulary of the UML, allowing you to create new kinds of building blocks that are derived from existing ones but are specific to your problem.

In other words, it is possible to create a new building block from an existing one by defining a stereotype and attach it to an existing one. The new building block will carry a new meaning. There are basically two ways to render a stereotype. The first one is to attach the name surrounded by guillotines close to the building block or in a specific location inside it. We have seen examples of this kind of stereotype display in some examples of the previous section. For instance an interface can be rendered as a stereotyped class with the stereotype name interface displayed in guillotines below the name of the class. The other way to render a stereotyped building block is to define a separate icon for it, different from the standard representation of the existing building block, and render it. This, of course, increases the expressiveness of the diagram but requires that the users of the diagram know beforehand the meaning of the new building block without the need to associate it with an existing building block.

The other extensibility mechanism provided by UML is the use of tags. Tags are key value pairs that can be attached to building blocks to convey some special information. Examples of key value pairs that would commonly be added to building blocks are versions and authors. Tags are rendered inside curly brackets and separated by commas in some place near the building block or inside it.

Another way of extending the UML is by adding constraints. Constraints shall be discussed in the next section, where the standard object constraint language shall also be briefly touched upon.

## 2.7 Constraints

Constraints are one of the ways of extending the UML according to Booch, Rumbaugh and Jacobson. Warmer and Kleppe (1998, pg.1) provide the following the definition of constraints:

A constraint is a restriction on one or more values of  
(part of) an object oriented model or systems.

Constraints can be written in natural language, but that would probably cause enough ambiguities that would probably defeat the purpose of making the model more precise.

There is trade-off between how well you can formally specify your model and the number of people that can really understand and make use of it. The UML, as a visual language, is designed to expressive. It also has become a de facto standard, meaning that the number of people that can understand concepts expressed in UML and utilize the language is large and increasing. On the other hand, the UML has some limitations when it comes to specifying a model precisely and unambiguously. There exists, though, mathematical methods that can be used to formally specify a model very precisely. Nonetheless, models created in such a way are accessible only to a small number of people. Perhaps, the best way one can bridge this gap and a level of precision to the UML is to use a standard, formal and yet simple constraint language. There exists one such a language and its name is Object Constraint Language, which has also become a standard and part of the UML specification. To just illustrate the power and usefulness of the language, the UML meta-model makes extensive use of it to clear a reasonable number of ambiguities and increase its precision.

A group led by Jos Warmer and Steve Cook of IBM developed the Object Constraint Language (OCL) as part of a business-modeling project in 1994. It later became part of IBM's UML submission to OMG as the recommended language for defining constraints in models and was officially adopted as part of UML version 1.1 (Eriksson and Penker, 2000, pg. 137.)

OCL is simple and helps a great deal in adding precision to a UML model. UML and OCL have been used in such eclectic applications as business modeling, software development and even the modeling and precise description of legislation (See Engers et al, 2001).

The constraints expressed in whatever constraint language are a good way of expressing business rules when doing business modeling. The Object Constraint Language was created for a business modeling project and that illustrates how an important clearly expressing constraint is in the business modeling field.

It is not in the scope of this work to provide a comprehensible description of the syntax of OCL, for a detailed description of OCL syntax with full explanations please consult Warmer and Kleppe (1998).

### **3. EXECUTABLE UML**

#### **3.1 Objective and Structure**

The purpose of this chapter is to explain in more detail the concepts related to Executable UML and how it differs from standard UML. The explanations given in the previous chapter about the UML and its main concepts give us a solid basis from which to start this chapter.

#### **3.2 Introduction**

Code generation was always at the core of the UML's objectives. As we saw in the previous chapter, Booch, Rumbaugh and Jacobson (1999, preface pg. xv), the original authors of the language, define the UML in the following way: UML is "a graphical language for visualizing, specifying, *constructing* and documenting the artifacts of a software-intensive systems". In their own words, the UML is also a language for constructing, i. e. building, software intensive systems.

According to Jacobson (Mellor and Balcer, 2002, pg. xxiii) "creating a modeling language that is also executable has long been a goal of the software community".

If constructing software systems straight from UML models has always been a goal and was always at heart of the UML initiative, why hasn't such a promise ever become true? There are a number of reasons that hinder the UML in its current state-of-the-art to effectively be used for generating executable software systems. First of all, the UML is considered too large, the number of features it has is too large and sometimes it is difficult to assess which kind of diagram or feature is appropriate for a certain context. Second, and perhaps more importantly, it lacks a clearly defined semantics for actions. This issue was addressed in the version 1.4 of the UML that incorporates some specifications for action semantics. Nevertheless no specific concrete syntax was specified for this and the specification leaves some gaps to be filled. Besides, it was noted that the UML specification creates too much coupling between the various diagrams and constructs. The UML clearly lacks a modular specification that would allow the use of selected parts of it while parts that are not interesting to the problem at hand could be left away. This issue is being addressed in the version 2.0 of the specification that intends to specify a language core surrounded by optional modules that could be added or not to the core.

## What is Executable UML?

Executable UML can be defined as a UML profile, or a series of restrictions on the standard UML to make it executable plus the precise specification of action semantics.

Balcer and Mellor (2002, pg. xxvii) define Executable UML in the following way:

It [Executable UML] is a profile of UML that allows you, the developer to define the behavior of a single subject matter in sufficient detail that it can be executed.

In order to make UML executable the first step to reduce the vocabulary and features of standard UML so as to clear ambiguities and allow for a more precise description of the problem at hand. The second and more important is to separate platform-independent concerns from platform dependent ones. In the executable UML methodology a platform independent model is created using executable UML features. This model can be translated to various platforms by using platform-dependent model compilers. It is expected that a variety of model compilers be crafted for various platforms. The model compiler makes use of design patterns and usually translates the executable UML into code to a certain platform.

The model written in executable UML is thus one level of abstraction above the code written in some programming language. As we mentioned in the first chapter, Balcer and Mellor (2002, pg. 2) noted that the history of software development is a history of raising the level of abstraction, they also explained:

As we moved from one language to another, generally we increased the level of abstraction at which the developer operates, requiring the developer to learn a new higher-level language that may then be mapped into lower level ones [...].

It is expected that the level of abstraction intended by executable UML would allow domain experts to take part in the modeling process with an expected improvement in the quality of the models and perhaps with the creation of domain-specific model libraries. It is also expected that model compilers be written by software experts using the most advanced software methodology and design patterns with a corresponding increase in the quality of the generated software. Fowler (1997, pg. 3) clearly expresses the importance of a greater participation of domain experts in the modeling process, he wrote:

One of the main reasons I use analysis and design techniques is to involve domain experts. It is essential to have domain experts involved in conceptual modeling. I believe that effective models can only be built by those that really understand the domain.

Executable UML uses a reduced set of the UML constructs. It also makes use of more precise action semantics. Model compilers are used to translate executable UML models to various platforms.

Executable UML is also one step towards model driven architecture. Model Driven Architecture (MDA) is a new initiative by the Object Management Group that has a similar objective to that of UML, namely to allow that platform independent models be created and later translated to a platform dependent one. MDA tries to build upon existing standards such as the UML, XML, CORBA, MOF and others. MDA is broader in scope than executable UML and has the strength of being developed by an established standardization group. Executable UML can be thought of as one way of implementing Model Driven Architecture or better as an aid to it, providing insights that will be useful in perfecting it and increasing its possibilities. Balcer and Mellor (2002, pg. xxvii) define in the following way how executable UML may support the Model Driven Architecture initiative:

This initiative is in its early stages, but its goal is to allow developers to compose complete systems out of models and other components. This goal requires at least an interface as contract, and behind the interface, the ability to express a solution without making code decisions.

It is not by chance that Mellor and Balcer's (2002) book is called Executable UML – *A Foundation for Model Driven Architecture*. The book is cited in the recommended readings in the Model Driven Architecture's web-site (<http://www.omg.org/mda>).

### **3.3 The Executable UML Language**

The purpose of this section is to discuss the Executable UML language and how it differs from standard UML. The Executable UML can be thought of as a dialect of UML, while it retains many of the UML's features there are some important differences that should not be overlooked.

The Executable UML does not use all of the UML's diagrams and constructs, many are thought to be redundant. The executable UML is intended to model precisely a system, any constructs or building blocks that could introduce ambiguity should be left out of the model.

The most fundamental modeling diagrams in executable UML are the class and state-chart diagrams. Classes should be modeled with a degree of precision. A state machine is attached to each class to describe its dynamic behavior and lifecycle. In other words, in Executable UML, each class is thought to have a state machine that responds to events. The actions that are taken in response to events or on a certain state are specified precisely using some sort of action language. The specification 1.4 of the UML includes the specification of action semantics even though no concrete syntax for those semantics is specified. Nevertheless, a number of action description languages already exist, examples are SMALL or TALL that were used in projects that adopted the early Shlaer-Mellor object oriented methodology. Typically, the executable UML tool will provide an explanation about the action language syntax or syntaxes that the tool can interpret.

Use case and activity diagrams are not an integral part of the Executable UML but they are recommended as methods for gathering requirements before the model is constructed. Activity diagrams are used to show the sequence of use cases and branching possibilities. Collaboration and sequence diagrams can be used to gain insights into the system or in some cases for visualizing aspects of the system after it has been built. They are not executable either.

The Executable UML methodology suggests that a system be partitioned into domains or subject matters. In the words of Mellor and Balcer (2002, pg. 30) "a domain is an autonomous, real, hypothetical or abstract world inhabited by a set of conceptual entities that behave according to characteristics". Each domain being modeled separately. Bridges are defined between domains, with some requirements being placed from one domain into another and connector points being defined for their exchange of information. The various domains and their dependency relationship are commonly displayed using package diagrams.

In the sub-section below we will how both class diagrams and statechart diagrams should be crafted in Executable UML.

### 3.3.1 Class Diagrams in Executable UML

As mentioned before, class diagrams are perhaps the most important diagrams in object-oriented modeling, as classes constitute the main building blocks around which object-oriented systems are designed and constructed.

Classes diagrams in Executable UML differ little from those of standard UML. The notation for the class construct is exactly the same, with a three sectioned rectangle. In Executable UML, it is essential to model all attributes that define a class. They should not be overlooked. Each attribute of course should have a data type. In Executable UML, there are two different kinds of data type that can be given to an attribute. There can be domain-specific data types that are specific to the vocabulary of the domain. On the other hand, the Executable UML has a set of pre-defined data types that are called core data types or fundamental data types. The core data types are Boolean, string, integer, real, date and timestamp. Domain specific data types are usually derived from core data types.

Operations aren't in Executable UML as fundamental as attributes for the definition of a class. Plenty of the dynamic behavior of a class is described in the Executable UML in the state machine associated with the class. Much of the behavior that would be commonly modeled as an operation is modeled as a set of actions executed inside a state in the state machine that defines the lifecycle of the object. This set of actions would be executed in response to some external signal that leads the state machine to transition to the state. It is common in Executable UML that no operations be defined for a class at all. The third compartment of the class notation could then be filled with a list of the event signals that the class's state machine is able to process. It can be expected that the receipt of those signals could trigger some sort of executable activity that could be analogous to the execution of an operation. In order not to avoid that this list of signals be taken for a list of operations, it is advised that this compartment be stereotyped as an event compartment, in which case the name *event* in guillotines would appear at the top of the compartment before the list of events. In spite of this, some operations can be defined for the class. In this case the set of actions associated with the state could be written on a note attached to the operation in a constraint-like notation.

Perhaps the most interesting differences between the class diagrams in standard UML and the way they should be crafted in Executable UML are found in the relationships between classes in the diagram. The UML defines four types of relationship between

classes, namely: dependency, generalization, association and realization. Dependency and relationship aren't useful for UML purposes and are ignored altogether.

Dependency, on the other hand, can be used to transmit some ideas visually but has no executable meaning at all. The UML also defines some special adornments for some special types of association, these are aggregation and composition. Aggregation has no deep semantics and is ignored. Composition is avoided as well, but for different reasons. Even though the semantics of compositions are well defined, the type of containment or control relationship between a class and its composed parts is regarded more as a platform dependent decision and is left for the model compiler. One-way associations between two classes are also not recommended. Normally we have two kinds of relationship between classes in an Executable UML model: bi-directional associations and generalizations.

There only four multiplicities allowed in executable UML at any association end: 1..1 (one and only one), 1..\* (one or many), 0..1 (zero or one) and 0..\* (zero or many). Standard UML allows that the multiplicity of an association end be specified in much more precision, such as 4..\* or 1..6, these kinds of multiplicity aren't allowed in Executable UML. Every association in Executable UML should have a unique name. This name is arbitrary and need not communicate the purpose of the association. It should be basically a label to identify the association unambiguously, such as R1 or R15. Role names are invariably attached to each association end. These specify the role that each class play in the association and should be easily readable by humans trying to get a grip of model. Sometimes it is difficult to find a noun that clearly expresses the roles that each class plays in the association, in such cases verb phrases can be used instead. As an example, in a relationship between a class Author and a class Book the following verb phrases in italics can be a substitute for the role name at each association end: Author *wrote* Book and Book *was written by* Author. Association classes are allowed. They are an important and powerful modeling mechanism in Executable UML that help a great deal finding the right abstraction and modeling some real-world concepts in the best possible way.

As for generalizations, all generalization relationships in Executable UML must ideally be disjoint and complete. Disjoint means that an object can be an instance of any sub-class of a certain more general class, but it cannot be an instance of more than one of the sub-classes of this class. The majority of the most popular object-oriented programming allows only disjoint sub-classing. In most object-oriented programming

languages objects cannot be constructed from two sibling classes. There is no restriction that prevents that this kind of abstraction be used in standard UML, even though it rarely finds parallels in 3<sup>rd</sup> generation programming languages. Complete means that all subclasses of a certain more general class are shown in the diagram, there is no hidden sub-classes. A tag written {disjoint, complete} can be attached close to a set of generalization relationships leading to a certain general class to indicate that this principle is observed. Another thing to take into account when using generalization in Executable UML is that the Executable methodology requires that all branch classes, i.e., all classes that have sub-classes, be abstract. In other words, no branch classes can be instantiated. Only leaf classes, those classes that have no sub-classes, can have instances. The UML has a special way of expressing that a class is abstract. The name of the class is usually drawn in italics to specify that it is abstract. The use of this notation is recommended. Alternatively, a tag {abstract} can be displayed close to the class name in the diagram for the same purpose.

### 3.3.2 State-chart Diagrams in Executable UML

State-chart diagrams are the preferred way to model the dynamic behavior or lifecycle of an object instead of the other behavioral UML diagrams such as activity diagrams, collaboration diagrams and sequence diagrams, even though those diagrams can be used at some stages of the modeling processing, specially as an aid to gaining an understanding into some of the desired behavior of the system.

State-chart diagrams in Executable UML do not differ greatly from those in standard UML. Nevertheless, some things have to be kept in mind. The Executable UML methodology recommends against using complicated chains of nested state machines. It is recommended that state-chart diagrams be kept simple and possibly flat (with no states having nested sub-states). Perhaps the most striking aspect in which state-chart diagrams in Executable UML differ from their counterparts in standard UML is that in Executable UML all the actions that are taken upon entry to or exit from a state as well as in response to internal or external transitions must be clearly and precisely specified using some sort of action language. A specification for action semantics was incorporated to the version 1.4 of UML but no concrete languages that comply with the semantics was specified. The standard version of UML does not mandate the actions dispatched from a state machine depicted in a state-chart diagram be precisely specified, while in Executable UML this is a must.

## **4. MODEL-DRIVEN ARCHITECTURE**

### **4.1 Objectives**

Executable UML can be thought-of as part of a greater initiative that works towards the same goal, namely to make models and the modeling process more valuable to organizations. This initiative is called Model-Driven Architecture, which is currently under the auspices of the Object Management Group, the same body that takes care of UML.

The objective of this chapter is to present the model-driven architecture and how executable UML fits into it.

### **4.2 Introduction**

Model-Driven Architecture is a new initiative by the Object Management Group that aims at putting models at the core of the software-development process. Currently, models are normally made at initial stages of the software development process at the so-called design phases. One may notice the need for an initiative such as Model-Driven Architecture by just paying attention to the way models are currently used. It is true that at initial stages of the software development process design models help to organize complex ideas and to form a high-level view of the system that is to be built. Design also can help in solving some complex problems early on in the development process. On the other hand, as quickly as design is released to developers, it may be regarded as a mere recommendation for how the system should be built, with little guarantee that the system will actually be built as specified. If some design decisions have to be re-visited, and decisions are taken to build the system differently during the implementation phase, rarely are the design models updated to reflect those decisions, which in effect will render the design models ineffective as a means to document and describe the system to future maintainers, which may be different people from the ones that are responsible for implementing the system.

Recently, we have seen the proliferation of the so-called agile-methods, which advocate putting a great emphasis in the code in detriment to other software artifacts just because the design artifacts all too often end up serving poorly their intended purpose. Iterative methods, such as the Rational Unified Process, try to solve the problems mentioned above by having short iterations that repeat all the software development steps from analysis to implementation.

Another reason behind model-driven architecture is to avoid platform volatility. As technology progresses in a fast pace, new platforms are quickly introduced. Commonly, software written for a certain platform has to be discarded when a decision to move to another platform is made. Software has then to be written from scratch, most models created for a certain platform have also to be discarded. If we are able to create a platform independent model that could be translated to platform dependent ones in a number of platforms, as is the aim of model-driven architecture, many of the problems arising from platform volatility could be avoided. Specially, high-level domain-specific decisions could be re-used, as the platform changes but the same platform independent model of the domain is kept.

### **4.3 Principles of Model-Driven Architecture**

The main principle behind model-driven architecture is the separation between a platform-independent model (PIM), which is created at the analysis phase of software development, and a platform-dependent model (PSM), which is generated from the PIM in the design phase with some degree of automation.

The concept of a platform-independent model as well as the concept of platform-dependent model are not necessarily new. The main contribution of model-driven architecture is that the transformation from the platform-independent model to the platform-dependent model is made with a high degree of automation by software tools, but still including some possibility of manual tuning or intervention. This way a platform-independent model could be ported to different platforms, the issue of platform volatility could also be partly solved, as new platforms are introduced new transformations could be defined for the PIM to those fresh platforms, which would guarantee that the ideas and concepts expressed in the PIM could be re-used and would have their lifetimes unlimited by platform volatility.

As the final step, the platform-dependent model would be translated to code. This is regarded as a trivial step as the PSM and the code are in a similar level of abstraction.

The PIM and the PSM could be expressed in the same language. Because the UML has become such an accepted language for modeling and is fully compliant with the MOF and other OMG standards, it sounds just natural to write both the PIM and the PSM in this language. It does not have to be like this though. In fact, the PIM and the PSM could well be written in different languages. For example, the PIM could be written in

UML and the PSM using ER diagrams. Model-driven architecture does not mandate or require that the PIM and PSM be both in the same language.

The platform-independent model is in a high level of abstraction. It's expected that by raising the level of abstraction and the hope the model would have a greater lifetime, a great deal of expertise could be brought in the confection of those models. The higher level of abstraction could signify that domain experts could have a more active role in the developing the system by actively contributing to the development of the platform-independent model.

#### **4.4 The Four Levels of Model-Driven Architecture Modeling**

Model-Driven Architecture defines four levels of modeling. At level M0, there are objects living and interacting in a real system. At level M1, there are models that define the structure and behavior of those objects. M1 models, on the other hand, are written in some language. The language constructs have to be defined somewhere. At level M2, there exist meta-models, or models for how models M1 can be built. For instance, at level M2 one can find the meta-model of UML, or a model of how M1 models can be written in the language. Meta-models themselves have to be written in some language, that is where the level M3 comes into play. The level M3, or meta-meta-modeling defines how meta-models defining modeling languages can be built, or a language for defining meta-models.

We could continue this way until an arbitrary modeling level M would be defined. In other words, there could be a level M4 defining a language for M3, and a level M5 from which M4 models are instanced and so on. However, the Model-Driven Architecture initiative has defined that the M3 model is written with the M3 language, so there is no M4 level to speak of in model-driven architecture.

In model-driven architecture, the language at level M3 is MOF, or meta-data object facility, an important OMG standard. The MOF model, or the model of MOF, is written in MOF itself, as explained above. Meta-models, or models defining modeling languages are instances of the MOF model and thus written in the language defined by it. For instance, there exists a meta-model for UML and it's written according to what is specified in the MOF model.

## 4.5 Model-Driven Architecture Standards

There are a number of standards developed or endorsed by the Object Management Group that are considered to have a role in the greater Model-Driven Architecture initiative. The main concepts and principles behind model-driven architecture can be implemented without the use of the standards. Nonetheless, the standards help a great deal in establishing common grounds on which tools can base in order to achieve a degree of interoperability. They also help sort out some issues.

Perhaps the most important standard related to model-driven architecture is the MOF or Metadata Object Facility. As explained above, it's basically a language for defining meta-models or a meta-language.

The UML is also an OMG standard and play an important role in the Model-Driven Architecture initiative.

Another very interesting standard is the XML metadata interchange or XMI for short. XMI defines a standard way for coding models and meta-models in XML format. The standard defines a way to generate a XML DTD for any MOF compliant language or meta-models. Models written in the language can then be coded in XML, taking the generated DTD into consideration. The DTD establishes a common ground for tools to exchange and interpret the models. Because the MOF meta-model is written in MOF itself according to the OMG specification, also an XML DTD can be generated for the MOF meta-model, which means that models written in MOF (which in effect are meta-models) can also be coded and exchanged in XML. A number of DTDs have already been generated for some of the most popular MOF meta-models, including the model of MOF itself and UML. The XMI-based XML DTD for UML has become increasingly popular as a vendor-neutral format for exchanging UML models among tools, which has led many people to believe that XMI is a UML standard. In fact XMI can be used for the exchange of models written in any MOF-compliant language, including but not just UML.

The UML profiles are also one standard that contributes to model-driven architecture. UML profiles are basically a way to define UML dialects using UML standard extensibility mechanisms (stereotypes, tags, etc). A number of profiles have been created or are being created for some popular platforms; those profiles will be a convenient language to write PSMs for those platforms. In other words, it's common to

envisage a scenario in which PIMs will be written in standard UML and PSMs in UML profiles corresponding to the target platform.

Besides XMI, the OMG and other organizations have also defined API mappings for querying models written in some MOF-based language. For instance, JMI, or java metadata interchange, defines a standard way to define java APIs to query models written in some language whose meta-model has been defined using MOF (for instance UML). Those APIs permit querying the state of a given model using standard APIs and also serve as standard interfaces for models repositories. There are similar APIs mappings for CORBA and other languages.

#### **4.6 Model-Driven Architecture and Executable UML**

Executable UML is intrinsically related to model-driven architecture. As an evidence of that, the authors of the main book about executable UML have mentioned model-driven architecture in the title of their book (Balcer and Mellor, 2002). Executable UML can be thought of as one way of implementing the model-driven architecture concept, with one notable exception though. While model-driven architecture recommends that a platform-independent model be transformed into a platform-specific one before it is translated into code, executable UML ignores this intermediate step. Most executable UML tools will translate models straight into code without generating a platform-specific model. On the other hand, the executable UML model compiler can be thought of as analogous to a transformation definition, where the rules about the transformation are declared.

## **5. THE EXECUTABLE UML PROCESS**

### **5.1 Objectives**

The objective of this chapter is to explain how executable UML might have an impact into how software is produced and delivered. Perhaps even more important than any possible productivity gains is the new ideas that the executable UML approach offers into how software might be ideally produced.

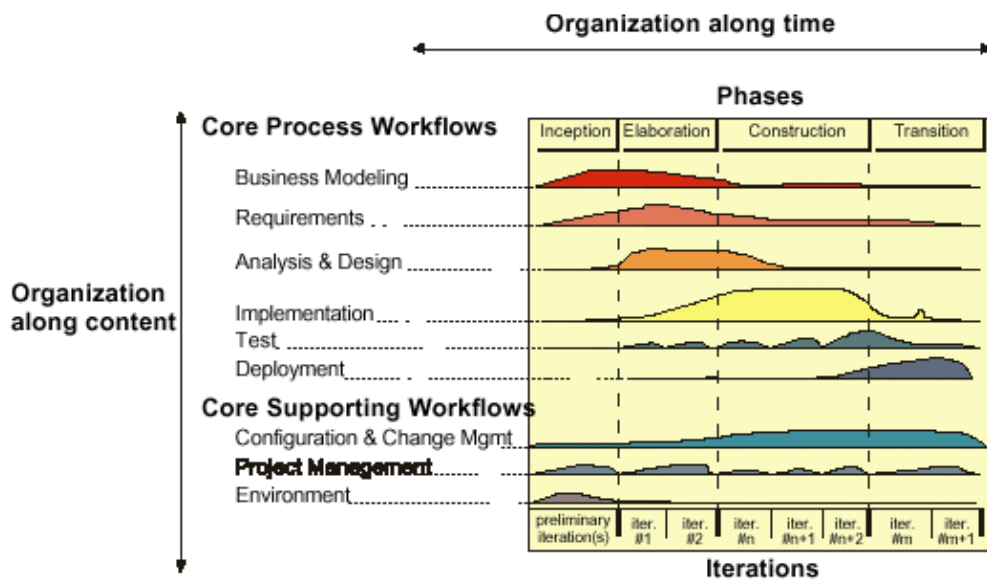
### **5.2 Introduction**

For long, the issue of how software might be best developed has been subjected to intense study. We can, with a certain degree of surety, affirm that so far the software development methodologies that have been created and utilized have achieved only partial success at best. They have almost all fallen short of their high expectations. The software development issue remains an area that still requires intensive investigation.

Firstly, it was attempted to utilize ideas from mature engineering fields to the construction of software. A new discipline was created and named software engineering, just to show how the ideas taken from engineering fields influenced this new discipline. The main ideas consisted, in summary, to separate planning and implementation, in the same way as engineering fields separate design and construction. Design requires extensive thinking in order to tackle all the risks and clear out any doubts before actual construction starts. Design is the most intellectually demanding task, while construction is by far the costlier. Construction should thus never start before planning is done completely, otherwise costs might escalate, as doing changes to an already built construction is difficult. This approach has proven inadequate to software development due to the fact it is often difficult to establish precisely the requirements and often impossible to avoid changes as the product is developed. The strict separation of product development into phases with a strict ordering is thus not suitable for software development. Part of this problem is resolved by adding iterations to the product development cycle, in which requirements are revisited and design continues.

Currently, it seems that the software development methodology that is more widely applied in companies developing commercial software is the Rational Unified Process, developed in Rational with the collaboration of the original authors of UML. Rational Unified Process or RUP for short is an iterative approach for software engineering. The

software development process is divided into four phases namely inception, elaboration, construction and transition. Each of those phases is divided into a number of iterations. At the end of each iteration a set of artifacts has to be delivered, which include certain pre-defined documents as well as prototypes and executable software. During each phase content-related workflows (or set of activities) are executed in various degrees. Those include requirements, analysis, design, implementation and testing. Each phase involves those in different doses. For instance the construction will involve more implementation than the inception phase, though the latter will normally include a certain degree of implementation too, for instance in order to create prototypes. The picture below shows the rational illustrates some of the main ideas behind the Rational Unified Process and its organization along axes. From this picture, one can see that the execution of the main workflows happen during all phases of the project life-cycle, though in different degrees;



Picture 5-1 The rational unified process, Kruchten (2001).

The authors of UML have indicated that UML would work best in conjunction with a development methodology that has similar characteristics of those of RUP. In the words of Booch, Rumbaugh and Jacobson (1999, pg. 33):

The UML is largely process-independent, meaning that it is not tied to any particular software development lifecycle. However, to get the most benefits from the UML, you should consider a process that is use case driven, architecture-centric and Iterative and incremental.

### 5.3 Modeling: Elaborative x Translative

Modeling was always at the core of the software development methodologies. Because their main source of inspiration, at the beginning, was the processes used in mature engineering fields, they drew from them the concept of blueprint. Software models would function as engineering blueprints, capturing the main aspects of the system to be constructed. Initially models were thought of as the rigid output of the design phase. Construction was to follow as strictly as possible what was specified in the models or software blueprints. Later the elaboration concept was introduced. Meaning during various phases of the project modeling is done in a certain degree of detail. At the initial phases of the project, for instance analysis, general models are crafted. Those models do express general ideas about the system to be developed but not in too much detail. As the project progresses to the design phase, details are added to model. In the implementation phase, models have sufficient detail to be implemented seamlessly. The elaborative approach is also present in RUP. It seems that the UML was strongly influenced by the elaborations principles. The path from an analysis model to an implementation model is well described in Booch, Rumbaugh and Jacobson (1998, pg. 16), although one has to also acknowledge that the authors considered that this process could be partly automated by a code generator.

Executable UML, on the other hand, advocates a different approach to modeling, which is called the “translative” approach. According to the executable UML methodology the activities of analysis and design should be completed separated. Modeling is done mostly in analysis. The models that are the output the analysis phase are not to be refined in design but rather translated to executable code using the model-translators, which are the output of the design phase. In other words, the design activities aim at developing a model-compiler to the target platform using the most adequate design patterns. The analysis activities aim at creating executable models of the subject matter in study. The same executable model can be translated using different model-compilers, or a single model-compiler can be used to translate many executable models.

The main proponents of the Executable UML have spoken fervently against the elaboration approach. In the words to Starr (2002 , pg. 22)

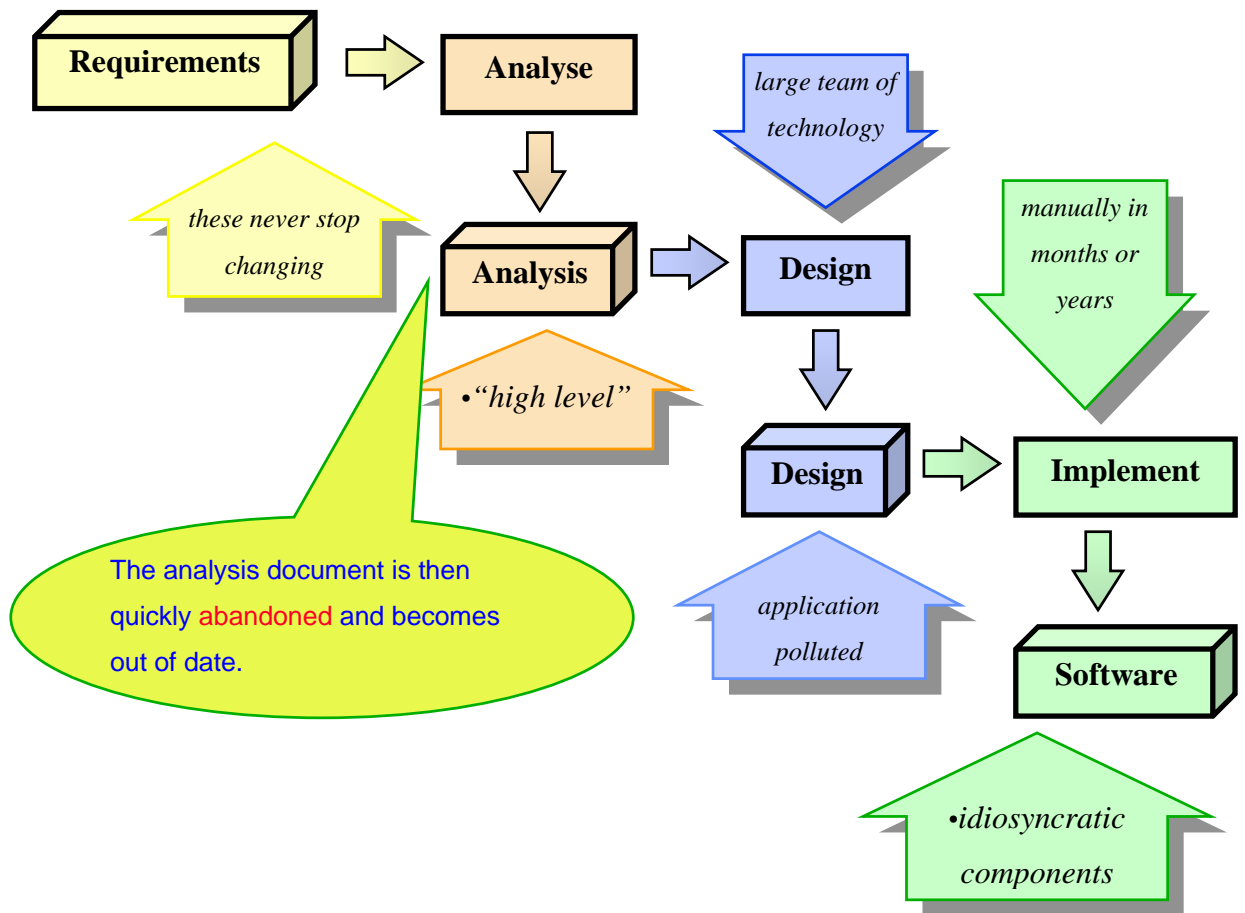
Elaboration destroys the original specification. In fact, the process of mucking up the specification with implementation details usually starts long before the specification itself is complete. So the

boundary between specification and implementation  
is always fuzzy with this approach.

Starr (2002, pg. 22-23) mentions stability and reduced risk as some of the main advantages of the translative approach. The translative approach allows for greater stability because analysis models can survive bad-designs and are not spoiled even if design decisions are proven later to be inadequate. Some the reasons for reduced risk according to Starr (2002, pg. 23):

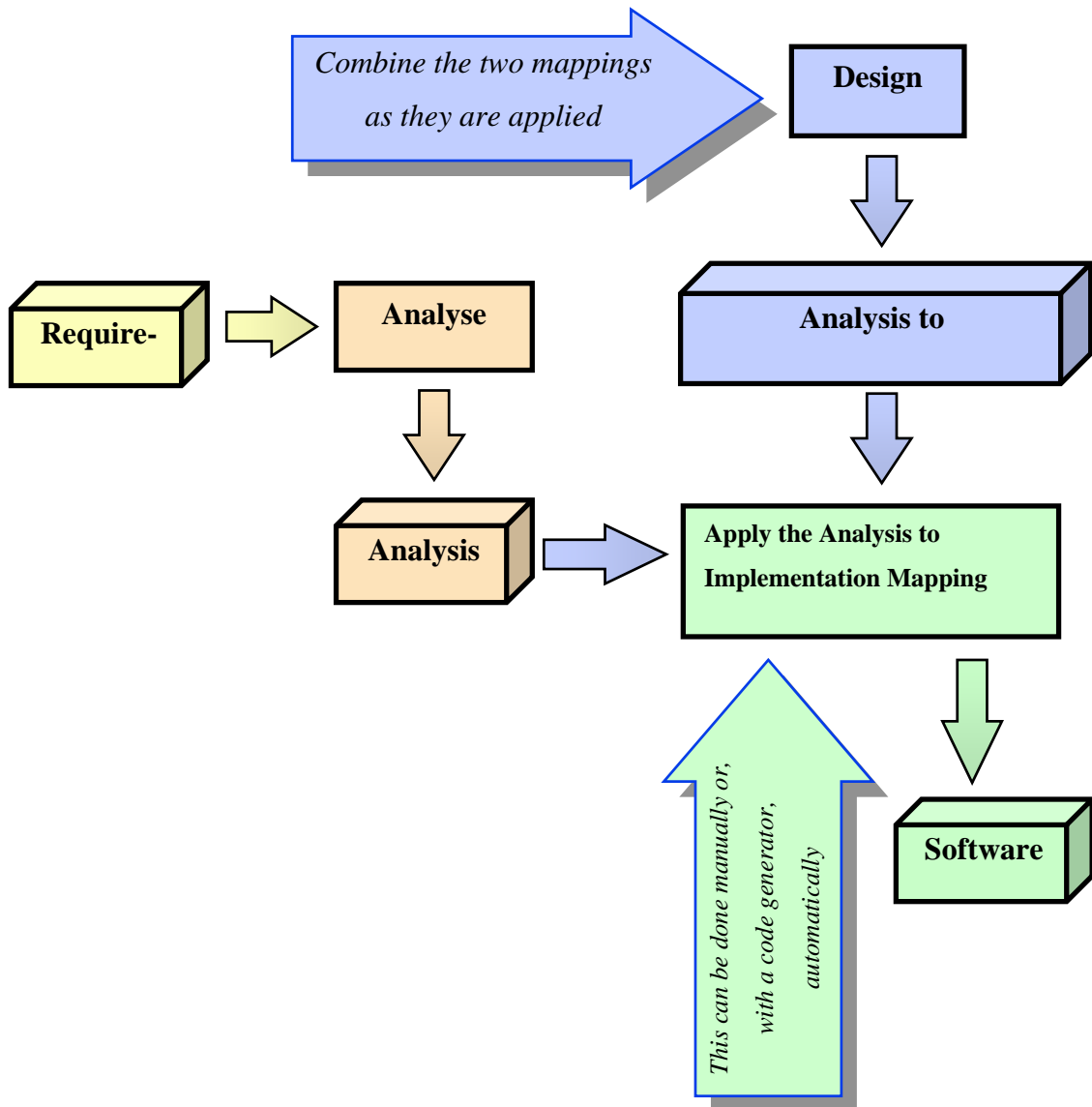
Imagine a worst case scenario where your programmers are total idiots. They devise a translation scheme that yields piggish code that would bring a Pentium 8 to its knees. Okay, sp you have to throw out the whole design and start over. But the models are safe. Since they weren't elaborated, they didn't melt down with the rest of the design.

The picture below illustrates the elaborative approach and some of its flaws according to the thoughts some of its critics from the Executable UML camp.



Picture 5-2 Elaborative approach, taken from Kennedy Carter.

The translative approach, on the other hand, is illustrated in the picture below, alongside some of its intended benefits.



**Picture 5-3 The translative approach, Kennedy Carter.**

The translative approach is far from widespread in today's software development, but we believe that its main ideas are sufficiently powerful to have the potential to change how software is produced and delivered and perhaps could lead to a new software development paradigm based on the automated translative approach. We believe that the translative approach do possess some advantages specially when it comes to making better use of analysis models and re-use of analysis decisions.

## **6. EMPIRICAL RESEARCH**

### **6.1 Objective**

This chapter introduces the empirical research that was done in order to reach the objectives of the research.

### **6.2 Research Objectives**

As mentioned in chapter 1, the objective of this study is to understand and evaluate the current state-of-the-art regarding executable UML. The UML is a comprehensive and powerful technology that has become a de-facto standard for visual-modeling of software systems. Despite its pervasiveness - one can estimate that a great portion of companies producing software utilizes the UML to a certain degree - many feel that the UML is not being utilized at its full potential. Some of the problems that have been identified is that UML is too big and in many ways ambiguous and perhaps most important of all it is not executable. Besides, it seems that the elaborative approach currently used in modeling does not allow for the best utilization of analysis models, which run the risk of being melted away as new detail is added and models are transformed into more less abstract representations.

Given those problems, Mellor accompanied with Balcer and others set out to sort some these problems in order to create a sub-language of UML that could be executed as well as some guidelines on how to best apply this language with the aim of model-execution. This sub-language was called Executable UML. We refer to Executable UML as a sub-language of UML that can be executed following the principles outlined by Mellor, Balcer and others.

There are already some proof-of-concept of Executable UML, especially in the real-time and embedded systems arena. We believe that because electronics manufactures use extensively diagrams similar to state diagrams in the development of their hardware, it seems all practical to use them also to develop their software too. But we fear that little has been said about its practical applicability to, for instance, business and management software. There comes the objective of this study, we would like to evaluate the current state of the art regarding Executable UML used for the development of real-life business and management software. Below are some of the questions that would like to answer with this research:

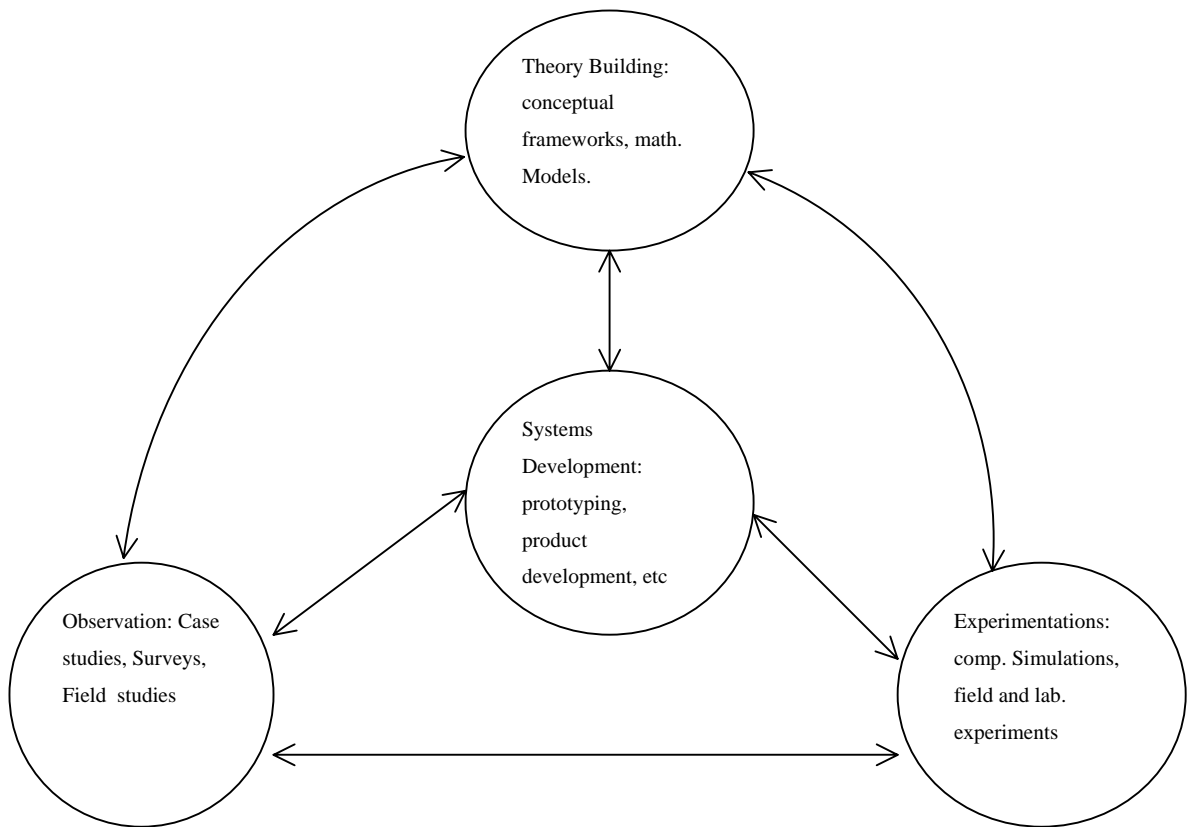
- What is the availability of tools for executable UML? Can they be practically used for software development?
- Does Executable UML really raise the level of abstraction? Does it help in the goal of having re-usable analysis models that could be easily translated?
- Does Executable UML provide productivity gains when developing business software?
- What are the practical shortcomings with the executable UML approach when developing business software?

### 6.3 Research Methodology

Nunamaker, Chen and Purdin in their highly influential article *Systems Development in Information Systems Research* (1991) laid the groundwork for using systems development as an effective tool in Information Systems Research. The authors demystified the view of some more conservative methodologists who often overlooked systems development as an effective method for research in the new field of Information System in favor some more traditional methods. The authors explained that

Perhaps the major motivation in computing and computer application research is, “what can be automated and how can it be done efficiently and effectively?”. This is consistent with the concept – development – impact model. It suggests that “theories” are needed to identify what broad classes of things can be automated, “instantiations” are needed to provide a continuing test bed for the theories, and that “evaluations” of particular instances (systems) are needed to quantify success or failure of a system in both technical or social terms. Systems development provides the exploration and synthesis of available technologies that produces the artifact (system) that is central to this process.

Nunamaker, Chen and Purdin proposed the following approach to multi-methodological information research:



**Picture 6-1**

We feel that the framework proposed by Nunamaker et al is very suitable for the problem at hand, specially the last sentence of the quotation above “Systems development provides the exploration and synthesis of available technologies that produces the artifact (system) that is central to this process”. Our goal is to evaluate an existing technology to software development, we feel that applying and actually using the technology is one of the best ways to understand the issues related to the this technology. Our method is actually a mixture of system development and experimentation. It is perhaps more in the experimentation side, but since we are evaluating a systems development methodology, experimentation will invariably involve the development of a system, thus it must involve both. It will involve prototyping a system using a given development methodology (Executable UML), the main objective is not the final piece of software, but rather to gain insights into the development methodology regarding the development of a real system, the final software product resulting from the experimentation is not necessarily of utmost value.

## 6.4 Research Design

### 6.4.1 Main Considerations

As explained in the sub-chapter above we would like to experiment with a particular development methodology, in this case Executable UML, by developing a prototype of a software system. The system itself need not be innovative or complicated, actually we feel that the more simple and commonplace this system would be the better it would suit our objectives, meaning that our conclusions would apply to a great number of current software projects. As are particularly interested in the applicability of the Executable UML for the development of business and management software, an ideal target software system would be a simple business software. Because the software itself is not the main product of the study, we believe that prototyping an existing system is superior to developing a brand-new one, as this approach would involve creative work that is not in line with the objectives of the research. Prototyping an existing software also has the advantage that we would in effect be involved in the development of a software system that has practical applicability, this would not be guaranteed if we attempted to devise an innovative system that has not been used in practice. Taking these issues into consideration we found that Crebit, a basic accounting software developed in the Swedish School of Economics would fit very well our study goals. We will as the empirical part of this study create a prototype of Crebit to a different platform to which it is currently targeted, using Executable UML. Crebit will be introduced in the following sub-section.

### 6.4.2 Crebit

Crebit is a bookkeeping software designed and implemented by professor Anders Tallberg at the Swedish School of Business and Economic Administration. It is especially suitable for education purposes, though its use is not intended to be restricted in this field. Crebit is a Windows software. It is can be used only in this platform, this is because professor Tallberg thought that this operating system was prevalent among students and small enterprises, so that support for any other operating system would not be fundamental.

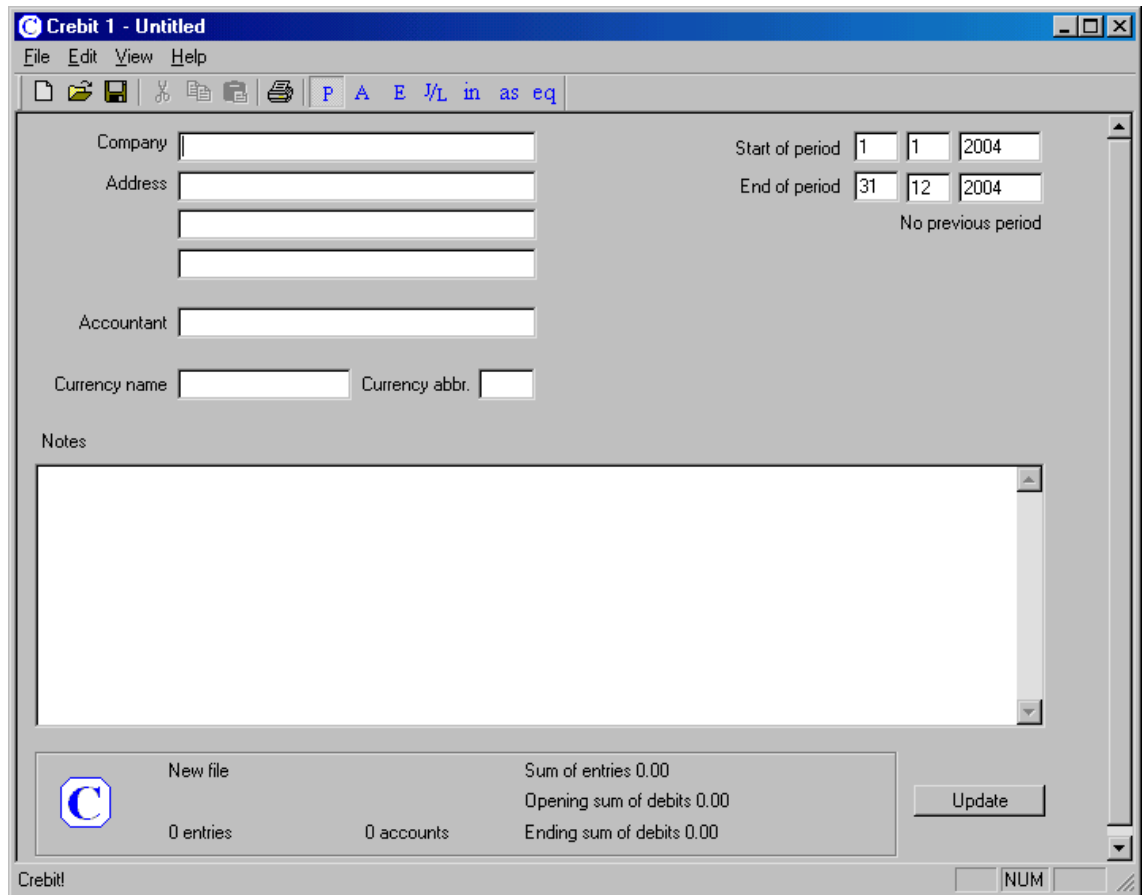
Crebit had a number of objectives that affected its design choices:

- It was intended to be simple and transparent, very easy to use and avoiding any sort of extra unnecessary complications that are not strictly related to the issue of bookkeeping.
- It should not be excessively automated, so as to rob students of gaining insights into the subject of bookkeeping and basic accounting, though it must automate some of the most mundane operations.
- The program should do the calculations right. Although seemingly an obvious requirement, professor Tallberg noted that many of the commercial packages on the market do not count right.
- Flexibility. The program should be flexible and not tied to the specificities of a certain country's accounting procedures.
- Ergonomic and Effective. It should be easy to perform the functions in the software.
- Lastly, the program was also intended to be multi-lingual. This perhaps due to the bi-lingual character of Finland and a growing need for internationalization.

The program uses flat-files to store its data and does not rely on any underlying database system. This adds simplicity to the implementation and use of the system, even though some of the principles of database-oriented system have been applied internally whenever possible.

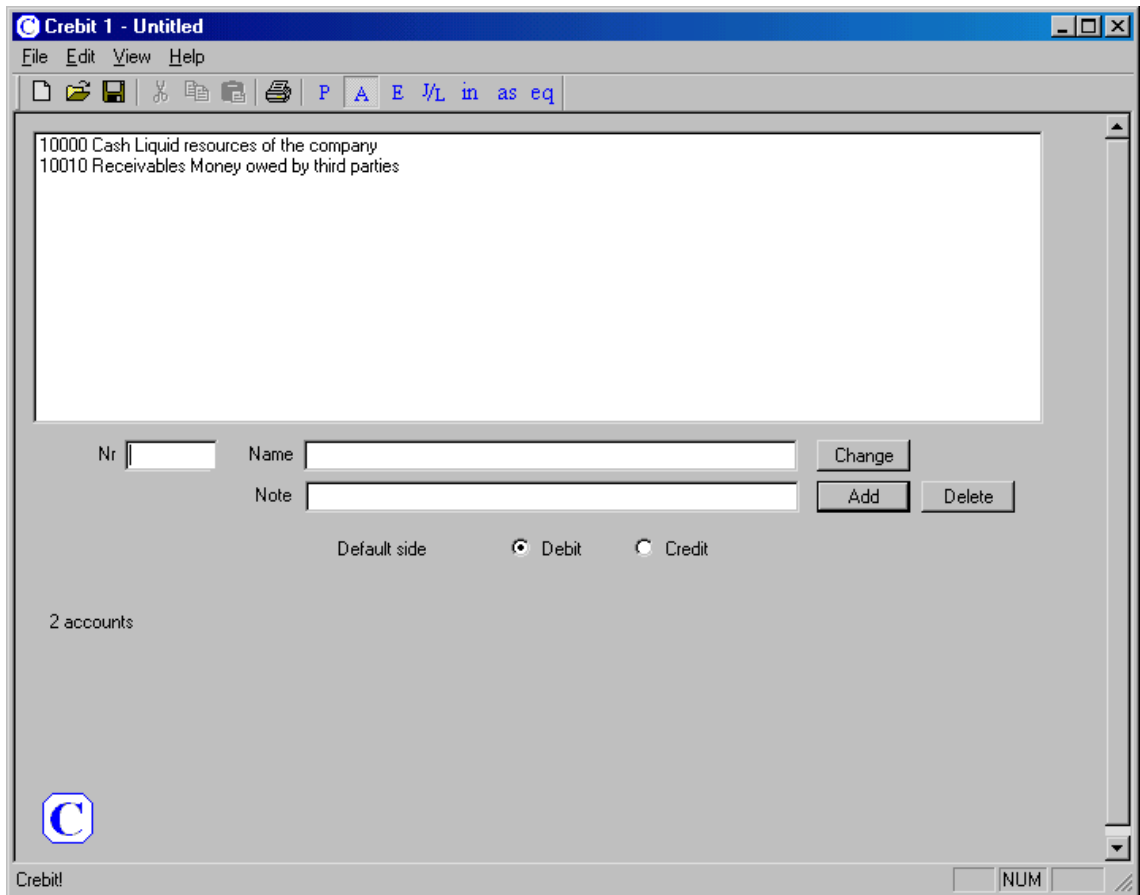
The installation of the program is very simple, which greatly supports some of its main objectives. In fact, the only thing necessary is to download or copy Crebit's executable file. No set up is required, everything is contained in one single small file.

Below one can see Crebit's main window and the area where data related to the company can be inputted.



**Picture 6-2 Crebit's main window**

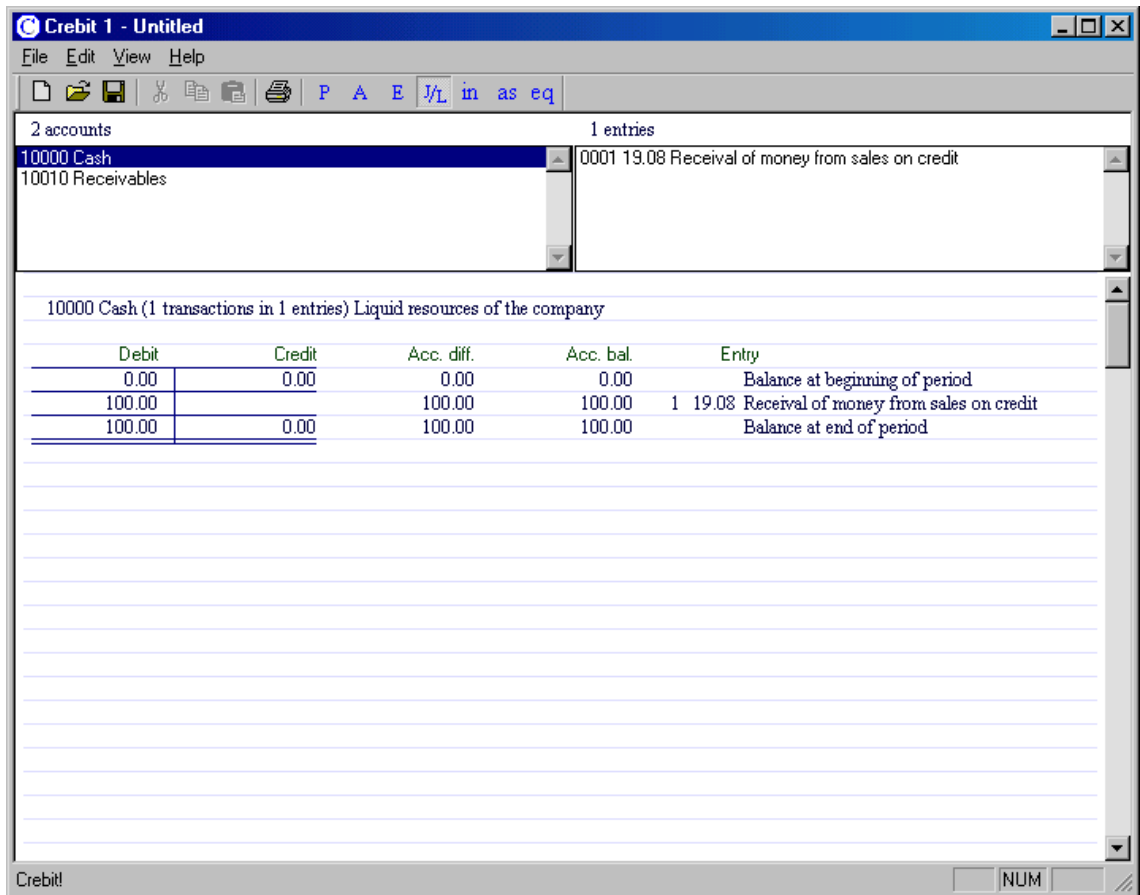
The user can navigate through Crebit's various views by toggling one of the seven toggle buttons in Crebit's toolbar, or selecting the desired view from the *View* menu. In the picture above, the parameters view is shown, which allows the user to enter or update data related to the company. The accounts view is shown below, there, the user might easily set up the chart of accounts for the company, by adding new accounts and modifying or deleting existing ones.



**Picture 6-3 Accounts view of credbit**

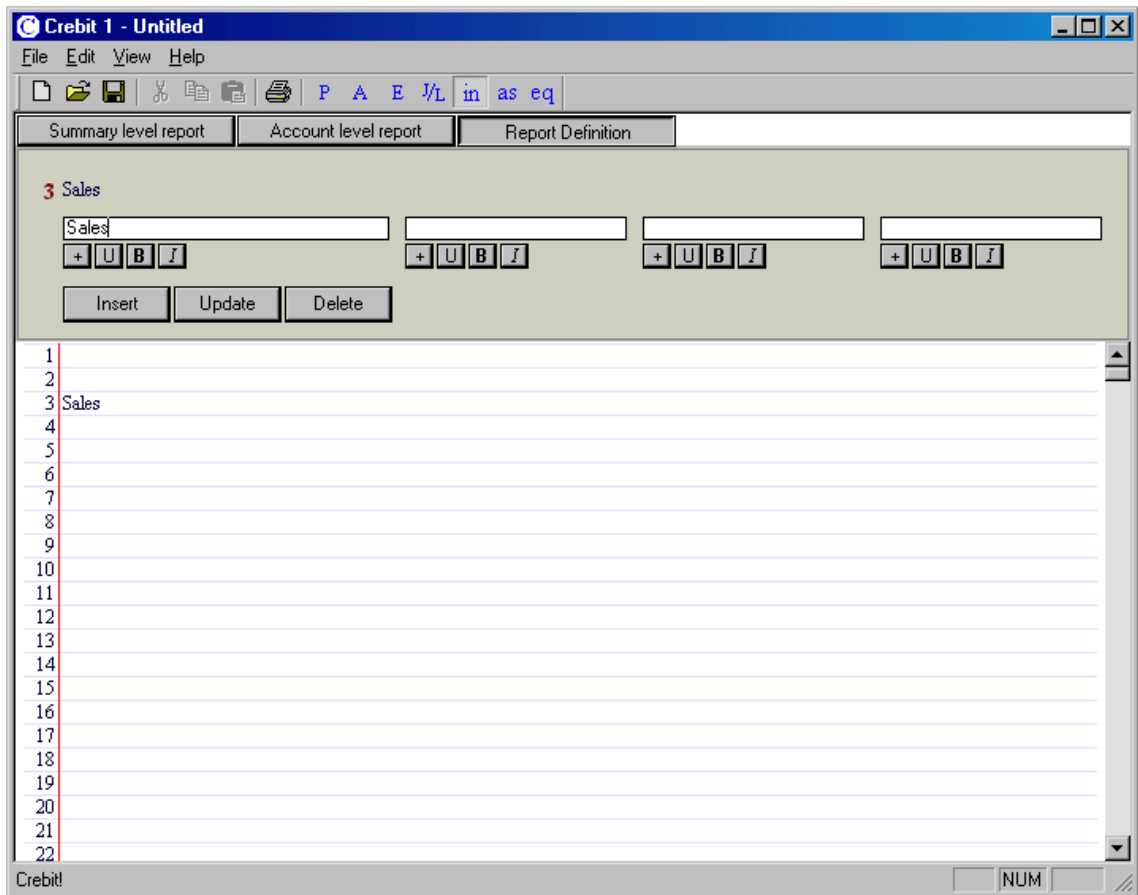
The entries view is used for adding, deleting or modifying entries to the report.





**Picture 6-5 Jornal/Ledger view**

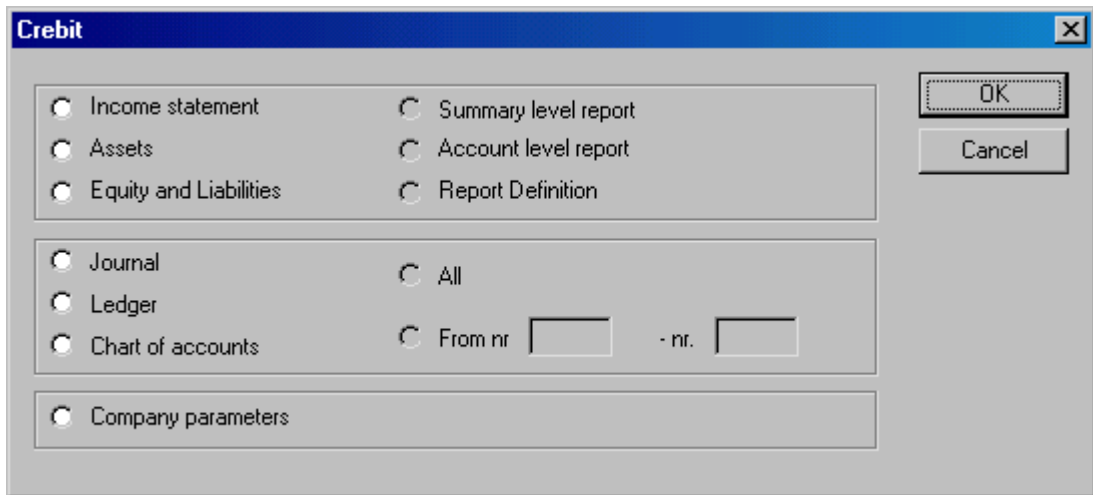
The other three views are all alike. They allow for the selection of values for certain reports and their formatting. The picture below shows one of such reports being defined:



**Picture 6-6 Income view**

The reports that can be defined similarly are the income statement, and the assets and liabilities and equity reports. The entries to those reports are defined by special codes that select a sum of accounts to be added to the report. So, the user itself must define which values shall enter each of the reports.

A number of accounting reports can be outputted from the system on the request from the user, as the picture below shows. The reports may also be previewed before they are actually printed.



**Picture 6-7 Printing reports from Crebit**

For more information on Crebit please consult Tallberg (2003).

## **7. RESEARCH**

### **7.1 Objectives**

The purpose of this chapter is to present the actual research and findings.

### **7.2 Tools**

We started our research by looking at the current tools for executable and specially one that would suit our needs. As the reader might probably guess, there is not at the moment a great deal of tools that support the Executable UML methodology as proposed by Mellor et al, though there are a great number of tools that perform some model transformation of some kind, including reverse, forward and round-trip engineering.

Apparently most tools for Executable UML target the development of embedded real-time software. As mentioned before, the electronics and hardware industry were the quickest to adopt the Executable UML concept. This is perhaps because the companies participating in this industry are already accustomed to dealing with reactive and event-driven concepts, specially in the design of their hardware, thus abstract their software as a collection of interacting event-driven state-machines is not a hard step.

Perhaps the most important tool in the market for Executable UML is Bridge Point from Project Technology. Both the authors of the Executable UML book, Mellor and Balcer, are senior managers of this company. We could then expect that the software developed would follow closely the concepts stated by the authors.

It is not greatly easy to acquire an evaluation version of this tool. In order to evaluate the software, a form has to be filled from the company's web-site. After filling and submitting the form, the company will again enter in contact with the prospective evaluator in order to ask further questions. The license key given is tied to the serial number of the hard disk, therefore it can't be transferred to another computer.

Bridge Point is said to have a reasonably intuitive user interface, and the manuals aren't too lengthy or complicated. Currently model compilers for C and C++ are offered by the company, though they don't accompany the evaluation version. A java model compiler seems to be available for internal use of the company. This model compiler isn't easily provided for third parties though.

The Kennedy Carter company, reached on the web at <http://www.kc.com>, is another company marketing Executable UML tools. From their web-site, it's possible to download a free a reduced version of an Executable UML tool called iUML. The freely available copy of their tool is not capable of any code generation though. The Kennedy Carter company also markets a configurable model-translation framework that can be used to perform model translations with flexibility. The name of the framework is ICCG. The author of this thesis tried to acquire an evaluation copy of ICCQ for testing from the Kennedy Carter company, but the representatives of the Kennedy Carter company were reluctant to allow a copy of their product, they claimed that the software is used internally in the development of their product line and can be released only under special conditions.

PathMATE from Pathfinder MDA is a very interesting product. Instead of having its own tool and user interface for capturing models, it does that from popular UML modeling tools such as Rational Rose, the market leader in UML tools. So, instead of having to learn how to deal with a specific user interface in order to design models, the user can use one of the popular UML tools supported by PathMATE to write his or her models, the tool will then capture the models and effectuate the translations. This has the potential of significantly speeding up the learning curve. Translations are available to C++ and Java.

Kabira technologies is also known for offering an executable UML tool. We didn't try with their products.

There exists a electronic group dedicated to the issue of executable UML tools, which can be reached at (<http://groups.yahoo.com/group/ExecutableUMLTools/>), the number of messages is very limited to date.

Aside from Executable UML tools, we mentioned that many (if not most) UML tools support code generation of some kind. Some have very good specialized round-trip engineering capabilities. Round-trip engineering being defined as simultaneous transformation from model to code and vice-versa, such as Together from TogetherSoft, which has become an increasingly popular tool. Round-trip is not considered analogous to Executable UML insofar as the models are not considered to be in a higher level of abstraction than the code, they are considered to be just a different representation of the underlying low-level code. Nonetheless, round-trip engineering has been used

successfully in the software-development industry and are said to provide some productivity gains.

Many other Model-Driven Architecture tools are cropping up, of which we can cite Compuware's OptimalJ tool or Methanology's MDE. Because MDA is such a loose concept, it's difficult to define how those tools are aligned with Executable UML, though they are built around the idea of model transformation.

We opted to use Pathfinder's PathMATE in the development of our prototype. The reasons are plenty. PathMATE's integration with popular UML tools such as Rational Rose, currently the market leader for UML tools, makes it easier to quick-start with the tool. In addition, and perhaps more importantly, PathMATE was the only one that had ready-made translation for java. There is no doubt that C and C++ make up a potent and versatile family of languages, but in the recent past the advantages of a smaller, more compact and more object-oriented language with a wide range of available standard APIs for various purposes such as java for the development of business applications have become clear. Java and its range of accompanying technologies have become increasingly popular and are known to provide productivity gains. We would like to try transformations to java. Pathfinder's personnel have also been very helpful, providing assistance quickly whenever necessary.

### **7.3 Modeling Considerations**

Perhaps the most interesting part of the research is to create an executable model of a software and understand how Executable UML could aid in the translation of such a model to code.

In the development of our prototype, we intend to use the MVC pattern. The MVC (or model-view-controller) pattern has been used successfully in the development of software and has normally been taken into consideration when the development of software that includes user interface components. It consists basically in dividing the application into three layers: one of them containing the relatively dumb user-interaction components that have the sole purpose of interacting with the user, at the bottom layer there are the objects that model of the real life application, this is the model part of the framework, between the model and the view sits the controller which is responsible for receiving and interpreting the events received from the user and acting upon the model or updating the view accordingly. The Executable UML seems more

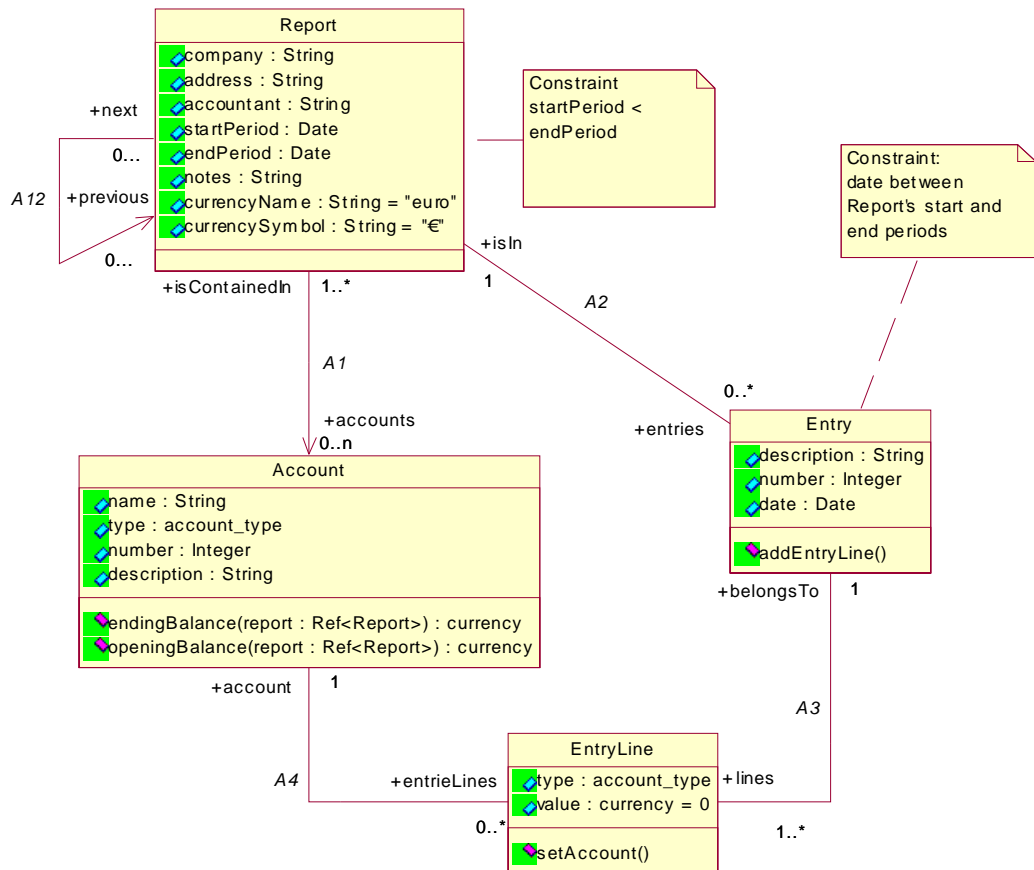
suitable for dealing with the model part of the application, especially when our goal is to create re-usable platform-independent model of a certain subject-matter.

Unfortunately, the UML has little to say about user-interaction components. The UML is largely deemed unsuitable for modeling and designing such objects. We will deal with the issue of user interfaces at later section of this chapter. Though not a widespread practice, the UML, and state-diagrams in particular, can be used advantageously in the development of the controller part of the application. We will also discuss this issue at a later section of this chapter.

#### **7.4 Creating an executable model of the application**

The first and one the most exciting parts of this research was to develop an executable model of the application using the Executable UML methodology. This model will be translated to java using Pathfinder MDA's PathMATE tool.

Below is our final model of a bookkeeping software using the executable UML methodology.



**Picture 7-1 Model of bookkeeping software**

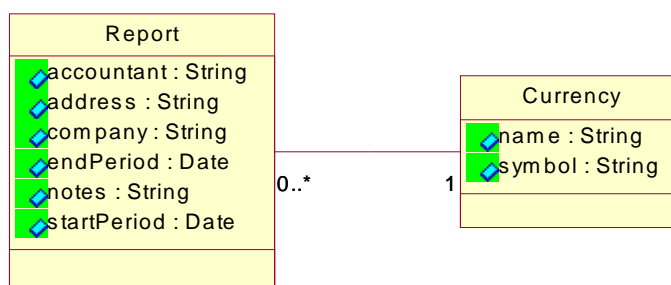
This model depicts the core of the application with the most essential classes. A report with some attributes is linked to a number of accounts and entries. An entry is made up of entry lines, each linked to one account. As one can see, the diagram above was written using the Executable UML conventions, all associations have a unique name, consisting of a letter plus a unique number, no aggregations or compositions, most relationships are simple binary associations.

We used basic data types as much as possible except in the case of account type, for which we defined an enumerated type with two possible values: debit or credit.

Two constraints could be used with advantage, one that specifies that the start date must be inferior to the end date of a report at all times (an invariant constraint) and another that specifies that the date of an entry pertaining to a given report must be within the report's period. These constraints were presented informally in semi-natural language in the diagram above. Alternatively, the constraints could be written in a semi-formal language, such as OCL or using action language. We did not write the constraints

formally because the tool that was used to translate the model to code did not have support for constraints, this logic had to be added to the view layer of the application. The tool also did not have support for a primitive date type, we worked around this by defining a custom data type called Date that serves as an alias for a primitive String type. We could alternatively have stored the date values as integers, meaning for instance the number of days after a certain selected date, we opted for strings instead in order not to avoid impairing flexibility, though both solutions are equally good.

By taking a close look at the diagram above, it is possible to notice that the Report class is not in its normal form. This is because there's a clear dependency between the currencyName and currencySymbol attributes. Clearly, it is possible to determine one from the other, the currency symbol could be established from the currency name and vice-versa. Ideally, both attributes would be extracted from the Report class in order to create a separate Currency class, as in the picture below.

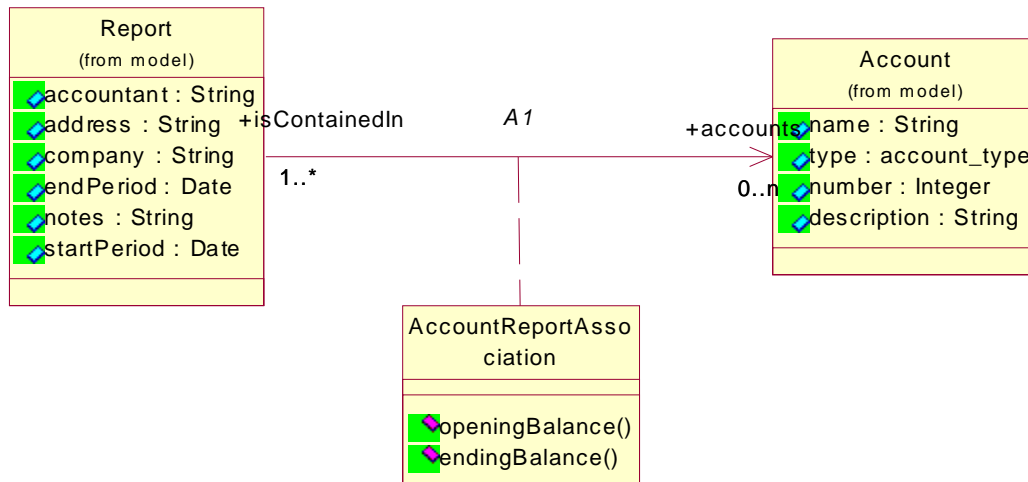


**Picture 7-2 Class diagram with a Currency class**

This way the definition of a currency could be re-used across reports. Each report would be linked to the definition of one currency, which could be linked to another report. We did not opt for this solution for simplicity reasons. Crebit originally does not have support for re-using currency definitions, and we believe that that is for a good reason.

There are two convenience methods in the Account class for calculating the ending and opening balances of a given account object relative to one report. The relationship from the Account class to the Report class is one to many, meaning that the same account can be linked to many reports, this is specially important when copying account definitions from one report to the other. The ending and opening balances of a given account make sense only in the context of a given report. Entries are made to a report and not to accounts directly. From these considerations, it's logical that the methods for calculating the ending and opening balance of an account relative to a report should not

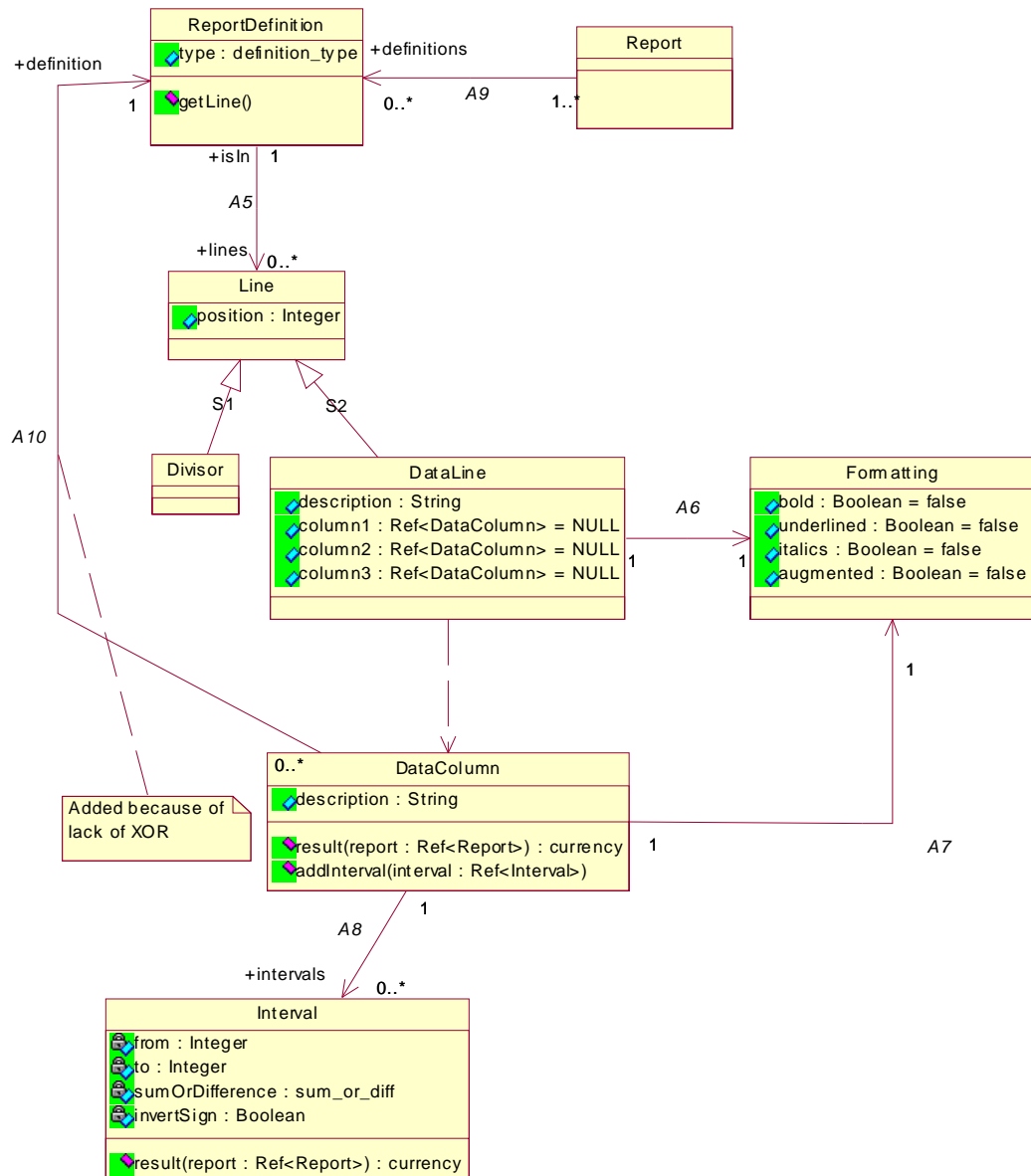
ideally from an object-oriented perspective belong to the account class itself but rather to the association between account and report, as in the picture below:



**Picture 7-3 Balance methods in an association class**

The tool we used did have support for association classes, but it apparently did not make possible to access data of the associated classes from the association class, only the opposite could be done (get data of the association class from the associated classes). We needed, for instance, to know the entries added to the associated report in order to calculate the ending balance of an account relative to the report. Because this was not possible, we could not implement an association class for the purpose of calculating the balances, the methods had to be added to the Account class instead, with the report object being given as an argument, as the picture 7-1 shows. As mentioned before, this is not the ideal solution from an object-oriented point of view, but it was the best feasible one given the limitations of tool.

Another important aspect of the application that is worth modeling is that of report definitions. The contents of three of Crebit’s four reports are defined by the user, who selects the values to be shown and added according to certain criteria. Modeling the definition of reports is also an interesting part of this application. The diagram below shows how we finally decided to model the definition of reports according to the Executable UML methodology.



**Picture 7-4 Report definition model**

Each report has multiple (usually three) report definitions. Actually, the multiplicity could well be defined as three or zero to three, but the Executable UML methodology does not enforce this kind of exact multiplicity for associations. In the Executable UML, only four types of multiplicity are allowed: one (1), one to many (1..\*), zero or one(0..1), zero to many (0..\*). As an attribute of the ReportDefinition, class there is the type of the definition, it can assume one of three values: income, assets or equity. A report definition contains a number of lines, whose main attribute is the line's position. The position of a given line cannot collide with the position of another line. Lines can be of two types, they can be a simple divisor or data line, containing data to be displayed.

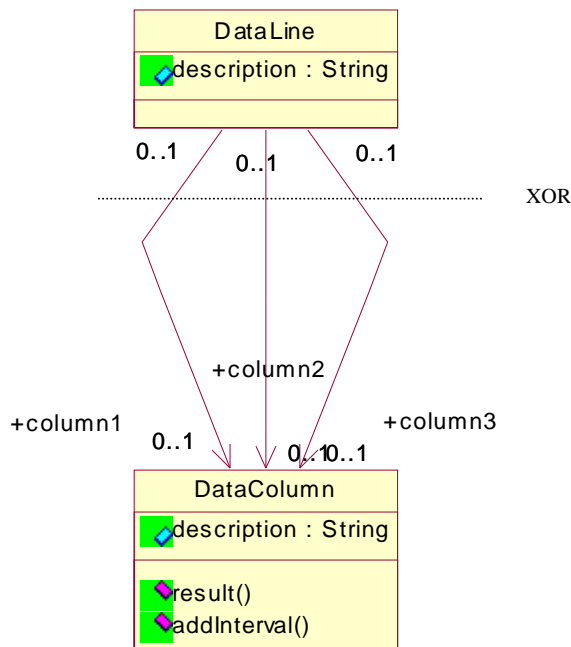
While divisors add no attribute to the line's position, data lines are more intricate, they contain a description of the line and three data columns as attributes. Data columns use opening and ending values of accounts under a number of intervals for calculating the value to be displayed. The description of a data column holds a string value which is to be parsed in order to reveal a number of intervals to be added. The rules for parsing the description are the same as for Crebit. They are explained below.

- Intervals are separated by a plus (+) or minus(-) sign. The plus sign means that sign of the sum of accounts in the interval will be kept, the minus sign indicates that it will be inverted. By convention, debit balances will have a positive sign and credit balances a negative one. Specifying a minus sign at the beginning of the interval signifies that those signs shall be inverted.
- After the plus or minus sign a code letter specifying whether it is the ending balance or the difference between the ending and opening balance of the accounts in the interval that shall be taken into account. A letter 'D' signifies that the difference will be taken into account, a letter 'S' signifies that the sum will be used.
- Following the code letter mentioned above, there is finally a number that specifies the ranges of the account number that shall belong to the interval. One to five numeric digits might be given, in each case shall belong to the interval all accounts whose first digits in the account number are similar to the digits specified. For instance, if two digits are given, such as 05, all accounts whose account numbers begin with those digits will belong to the interval, in this case all accounts whose account number are between 05000 and 05999

Intervals are then obtained from the results of parsing. If, for instance, the following code is given +S1-D24, two intervals shall be created and linked to the data column, the first will consist of the sum of ending balances of the accounts whose account number fall between the interval 10000 and 19999 inclusive, the second interval will include all accounts whose number are in the interval 24000 and 24999, the difference between the ending balance and opening will be summed and the sign inverted.

Each data column as well as the description of the data lines are linked to a formatting object that contain data that will be used for formatting the font when displaying the actual reports to the user.

The class diagram presented above is not the optimal from an object-oriented perspective. We believe that having data columns as attributes of the data line is a secondary option. Ideally, there would be three associations from the data column class to the data line as in the picture below:



**Picture 7-5 Best solution for lines and data columns**

Ideally, there would be three relationships from a data line to a data column representing each of the three allowed columns. The problem is that having three associations, each with a multiplicity of zero to one at the data line end, could potentially signify that a single data column could be linked to more than one data line through more than one association. That's why it's necessary to add the XOR constraint at the data line end of the association, to specify that only one of these associations can be true at a time from the data column perspective. In other words, a data column could have only one data line at the other hand through one and only one of the three associations. The tool we used did not have support for XOR constraints, so we had to work this around by having data columns as attributes of the data line class, a dependency relationship was added between the data column and data line classes, this dependency relationship has no meaning in Executable UML and was added for the purpose of illustration only.

The lack of support for XOR constraints in the tool and our resulting choice to store the relationship between data lines and columns as attributes of the data class had yet

another implication to our diagram. Because we could no longer traverse the relationship from the data column to the data line directly – the columns have no knowledge that they are used as attributes of another class – we had no indirect access to the report definition from the data column, this motivated the inclusion of association A10 linking the data column to the report definition. This relationship wouldn't be necessary had the column access to its associated data lines.

## 7.5 The dynamic part of the application

The original executable UML places too much emphasis on state-charts for modeling the dynamic behavior of the modeling elements. There are many situations when creating such a model of a system reactive to events and changing in discrete states could be overly complex. Quite often, we will find out that modeling the dynamic aspect of the modeling entity in such a way that it will execute a set of actions under the receipt of an external synchronous signal and then return to a previous idle state is not only simple but also adequate. This kind of behavior corresponds to modeling the dynamic aspects of an object with procedures and methods, and dispenses the use of state-chart diagrams.

We decided to model the dynamic aspects of our prototype as methods written in Action Language. We believe that that would be the simplest solution for the task at hand. Picture 7-1 and Picture 7-4 show the methods that were added to modeling elements in the UML notation for class methods. Some are trivial and were added for mere convenience, such as the method *addEntryLine* of the Entry class, the method *setAccount* of the EntryLine class, the method *getLine* of the ReportDefinition class and the method *addInterval* of the DataColumn class. No further details need to be added about those methods. We believe that their names fully explain their purposes. Other methods are more connected to the essential functionalities of the system, such as the methods *openingBalance* and *endingBalance* of the Account class and the *result* methods of the DataColumn and Interval classes. Those methods deserve some attention that will be devoted in the section.

As mentioned in sub-section 6.4.2, calculating balances right was among Crebit's main objectives. The two methods *openingBalance* and *endingBalance* are of crucial importance. As it is discussed in the previous section and shown in Picture 7-3, those two methods would be best added to a class modeling the association between the Account and Report classes. This was not possible because Pathfinder MDA's

PathMATE's action language apparently does not allow access for the attributes of the associated classes from the association class. As a result, these methods had to be added to the Account class, which we consider as a sub-optimal solution. The same account object can be added to many different reports, because of that, the report in question has to be passed as an argument to the *openingBalance* and *endingBalance* methods, as we are interested in the balances of an account relative to a certain reports, calculating the overall balance of the account taking into account the entries added to all reports to which the account is associated would make little sense. Below is the *openingBalance* method written in the action language of Pathfinder MDA's PathMATE. The action language in the piece of code below is a copyright of Pathfinder Solutions LLC. It's out of the scope of this work to describe any concrete syntax of UML's action semantics, we will rather provide descriptions of the operations. We believe that the personnel of Pathfinder MDA will be helpful in providing answers to queries about their action language and may even be able to provide a copy of the manual. From our experience, the action language can be learned with no difficulty and with no great investment in time.

```
1: Ref<Report> previous = FIND report->A12->@previous Report ;
2: IF (previous == NULL)
3: {
4:     RETURN 0 ;
5: }
6: ELSE
7: {
8:     RETURN this.endingBalance(previous) ;
9: }
```

From the diagram in Picture 7-1, one can see that the Report class is associated to self in order to provide a link to a previous report. The multiplicity of the association is zero to one, which means that a report object might be linked to a previous report, or it might not have a precedent report. The code in line 1 navigates from the report passed as an argument to its previous report through the association named A12. If the navigation returns a null reference, in which case the previous report isn't known, then the opening balance of the account in question will be zero (as it can be seen from line 4). If, on the other hand, a previous report is known then the opening balance of the account will equal the closing (or ending) balance of the same account as related to the previous report (as one can see in line 8).

The action semantics procedure for the *endingBalance* method is shown below:

```

1: Ref<EntryLine> line;
2: currency sum = this.openingBalance(report);
3: FOREACH line = this->A4
4: {
5:     Ref<Report> lineReport = FIND line->A3->A2;
6:     IF (lineReport == report)
7:     {
8:         IF (line.type == debit)
9:         {
10:            sum = sum + line.value;
11:        }
12:        ELSE
13:        {
14:            sum = sum - line.value;
15:        }
16:    }
17:}
18:RETURN sum;

```

In line 3, a currency variable `sum` is set to the opening balance of the account as relative to the report in question. The method then iterates through all the entry lines associated to the account through the association `A4` (see Picture 7-1). The method will then have to check whether each entry line belongs to an entry that was added to the report in question or if it is a part of an entry that belongs to another report. This is done by navigating from the entry line to its entry through association `A3` and from the entry to the report it belongs to through association `A2` as shown in line 5. The entry line will be used to update the sum only if it is a part of an entry to the report in question, as one can see from line 8. Following the conventions, the value of the entry line will be added to the sum if it is of debit type or it will be subtracted from the sum if it is of credit type (by convention credit transactions have negative value). The sum is finally returned as the result of the method in line 18.

Below is the piece of code for the *result* method of the Interval class.

```

1: Ref<Account> account;
2: currency sum = 0;
3: FOREACH account = report->A1 WHERE (account.number >= from &&
account.number<to)
4: {
5:     currency accountValue;
6:     IF (sumOrDifference == isSum)
7:     {
8:         accountValue = account.endingBalance(report);
9:     }
10:    ELSE
11:    {
12:        accountValue = account.endingBalance(report) -
account.openingBalance(report);
13:    }
14:    IF (invertSign)
15:    {
16:        accountValue = -accountValue;
17:    }
18:    sum = sum + accountValue;

```

```
19: }
20: RETURN sum;
```

As with the *openingBalance* and *endingBalance* methods, the reference to a report has to be passed as an argument to this method. This is because a single report definition can be linked to many reports, as is the case when report definitions are imported from one report to another. An Interval object stores a range of account numbers, the account objects whose account numbers fall into the interval range are included in the final result. Depending on the value of the *sumOrDifference* attribute, the final result will consist of the sum of the ending balances of the selected accounts or the sum of differences (ending balance minus opening balance) of those accounts. The sign of the final result might also be inverted depending on the value of the Boolean attribute *invertSign*. The code fragment above follows this logic. Firstly, all the accounts of the report in question (passed as an argument) whose account numbers are within the range of the interval are iterated through (line 3). A value is taken from each account that is iterated through, which could be the account's ending balance or the difference between its ending and opening balances (lines 6 to 13) depending on the value of the attribute *sumOrDifference*. The sign of this value might be inverted depending on the attribute *invertSign*. The sum is updated and finally returned at line 20. As an observation, the ranges and attributes of Interval objects are obtained from parsing text strings, as explained on section 7.4.

A DataColumn object contains a number of Interval objects. The result that is returned from a data column is nothing more than the sum of the results of its associated intervals, as evidenced from the piece of code below, which we believe is quite self-explanatory (the returned value consists of the sum of values returned from the *result* method of each associated Interval object).

```
1: currency sum = 0;
2: Ref<Interval> interval;
3: FOREACH interval = this->A8
4: {
5:     sum = sum + interval.result(report);
6: }
```

## 7.6 Generating the target code

Code generation is made very easy from Pathfinder MDA's PathMATE. By just clicking on a menu command that is added to Rational Rose's main interface, one is able to seamlessly generate code for the target language. The model is checked and any error is reported.

We did experience some problems though, there seems to be some problems in the standard templates, which cause models containing data types consisting of groups of references (kinds of collections of reference objects) to generate code that can't be compiled correctly. We tried changing the generated code and we were then able to compile it correctly. Because we wanted that the java code from the model be totally generated from the tool, with zero manual intervention, those data types were avoided in the final executable model, which did not cause us any great disturbance. This problem is not as serious as it may seem to many, we believe that simple changes to the templates would be able to fix this problem.

The greatest problem we faced with the generated code has to do with the principle of strict separation of domains that underlies the executable UML methodology. The executable UML approach is heavily based on the separation of domains, which means very loose coupling among domains. Following this approach, the tool does not allow that classes in a given domain be aware of classes in another domain. This presents us with some challenges, as usually there must be a heavy flow of data between a modeled domain and the user interface. We must then find a way to structure this flow of data. Some sort of import mechanism in which classes on a certain domain are added to the namespace of another through an "import" dependency would be a useful addition to the Executable UML methodology or tools, specially when data must be transferred between a modeled domains and a UI domain.

Adding domain-specific service methods to retrieve data from the model domain to the UI domain in effect reduces decoupling between the domains. If new attributes are added to the classes, the interface would have to be changed in order to transfer data related to the new attribute from and to the domains. This was the previously thought solution: in other words, to add a range a domain-specific services for transferring data from the model to the UI, while classes, their attributes and methods remained invisible outside the model. This approach proved difficult as a plethora of domain-specific methods would have to be created and would result in a great duplication of efforts.

Our solution to this problem was to effectuate changes to the model-compilation templates so that classes could be visible outside their packages. Besides, their attributes and operations were made accessible. Constructors remained package restricted and domain-specific services (methods) were created so that the creation of the domain classes could be signaled from the UI domain.

Our experience changing templates is mostly positive. Although we did not thoroughly studied the template language used in Pathfinder MDA's PathMATE we managed to make changes to the templates quite easily by just following the intuitive naming conventions. The fact that we had to make small modifications to the templates is not negative per se, the executable UML methodology acknowledges that fine-tuning of off-the-shelf model translators would in many cases be necessary. For some projects, developing a custom model compiler would be the best solution.

We believe that showing snippets of generated is not necessary for the layout and appearance of the generated code can be changed by just altering templates.

### **7.7 The user interface**

The issue of developing a user interface for an executable UML program is a tricky one. Actually, the UML says little about user interfaces. The UML is largely considered inadequate for depicting let alone developing user interfaces. Speaking of the limitations of UML Rumbaugh, Jacobson and Booch (1998, pg. 4) contended that:

For specialized domains, such as GUI layout, VLSI circuit design or rule-based artificial intelligence, a more specialized tool with a special language should be appropriate.

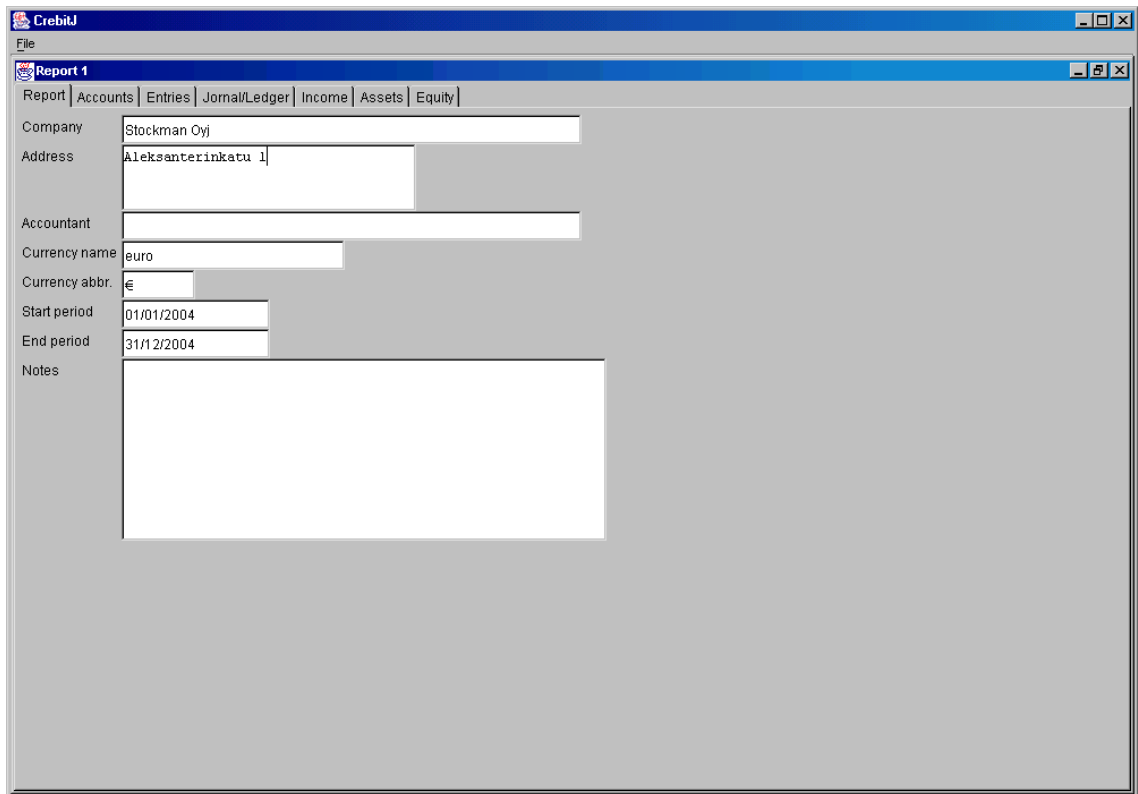
If the UML is not generally concerned with the design of user interfaces, could Executable UML aid in any way in such an endeavor? We set out to answer this question by asking the many Executable UML specialists at the executable UML electronic group. Some of the respondents were enthusiastic though a little doubtful about the development of a re-usable GUI widgets Executable UML domain in the future. Others contended that the satisfaction of the functional requirements would be independent of the communication mechanism with the user, thus the issue would be somewhat secondary, or too low-level, and better relegated to specialized tools for GUI development. Another respondent argued that the best way would be to bridge through data agents from the model to a pre-implemented GUI domain.

We found in the work of Horrocks (1998) perhaps one of the most conclusive hints as to how Executable UML might contribute in the future to the issue of GUI development. Horrocks basically showed how state-chart diagrams, especially those containing nested states, could be used in constructive ways to design the controller part of applications involving a user interface. As explained in previous chapters, state-chart diagrams are

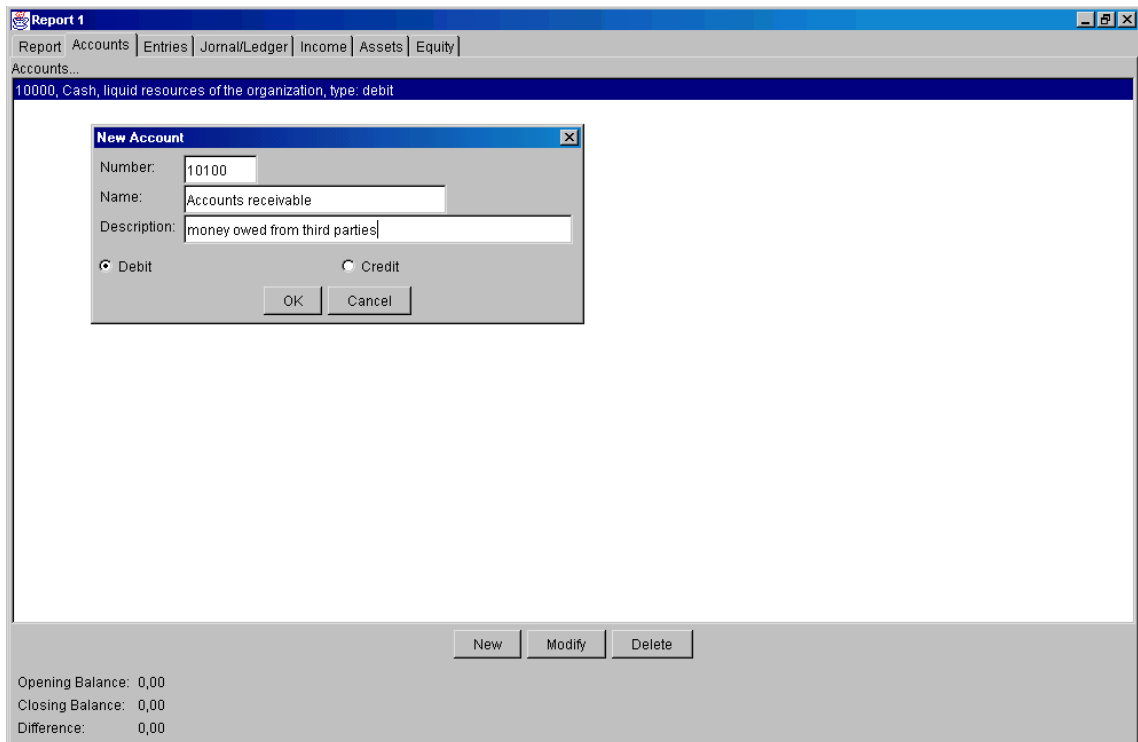
one of the main diagrams in Executable UML. The author started by demonstrating how flawed the traditional way of coding event handlers was, which in the end had to resort to some sort of global state and went on to show that state-chart diagrams provided fruitful ways to design the states and transitions of a user interface with practical examples. The work of the author preceded Executable UML and of course did not have it in mind. The executable UML community seems to be slow in adopting similar ideas. After reading the book, we felt convinced that those ideas have a good potential in the future of Executable UML.

In spite of what was said in the previous paragraph, we decided not to use the ideas presented in Horrocks's book for the development of the user interface of our prototype. Though potentially fruitful, we feared that those ideas were still too unproven and could potentially retard the development of the prototype. We opted instead for developing a user interface in the traditional way by coding it in the target programming language (java). We did not use any GUI design tool, for we believed they would not substantially speed up the development. Implementing a user interface is not obviously the main topic of this work, but our intention was to develop a working prototype and that could not be accomplished without a user interface. Besides, developing a user interface provided valuable insights. Many of those resulted in changes or adaptations to the model. In the previous section some of those insights were discussed along with solutions taken.

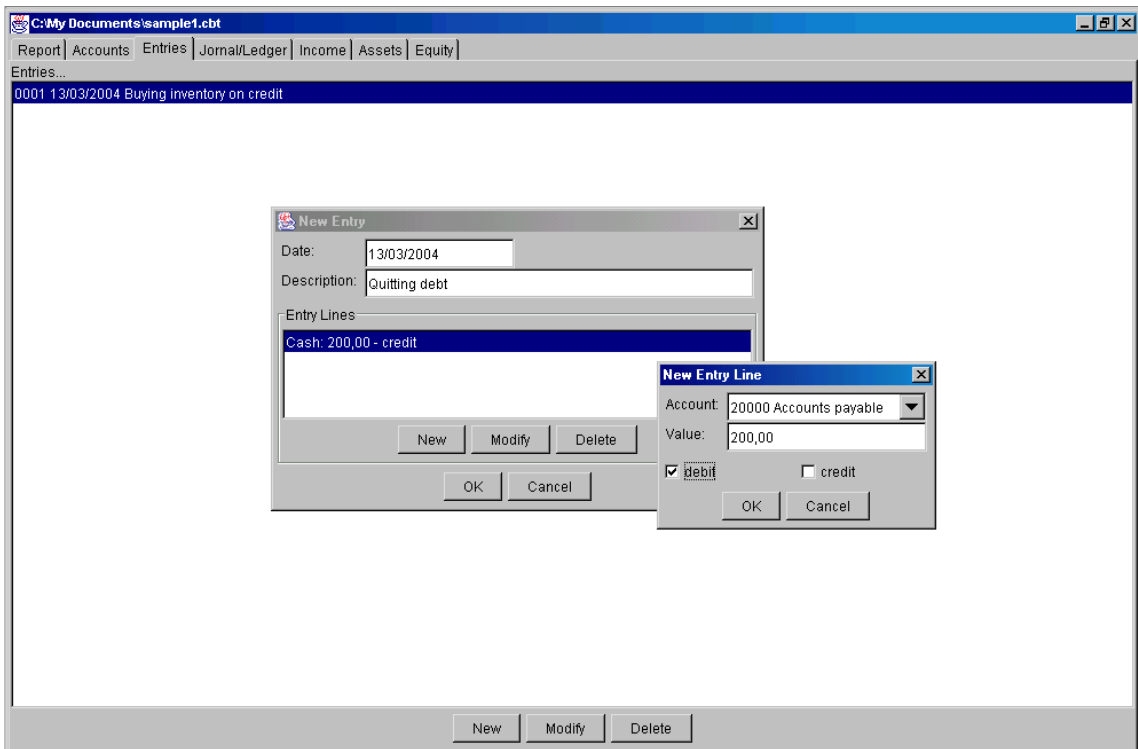
Below, one can see some snapshots of the user interface. It tries to be as much faithful to Crebit's original user interface as possible with some minor improvements or adaptations added.



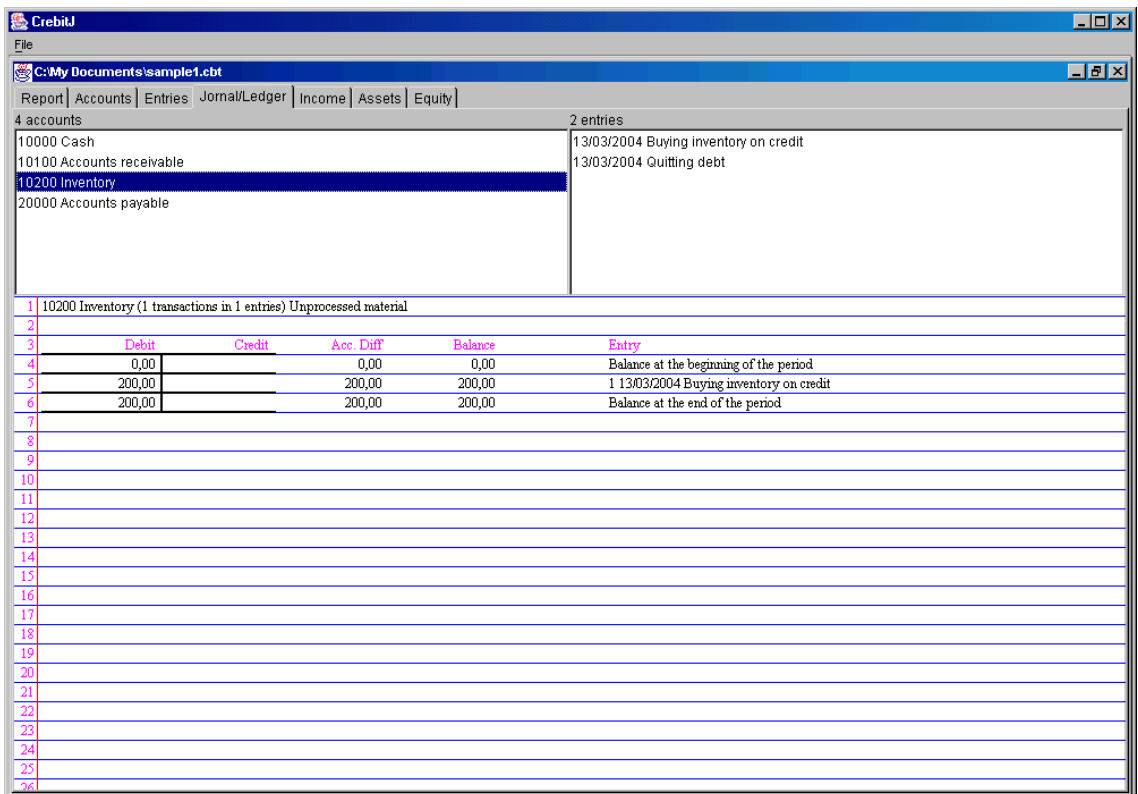
**Picture 7-6 Entering basic report data**



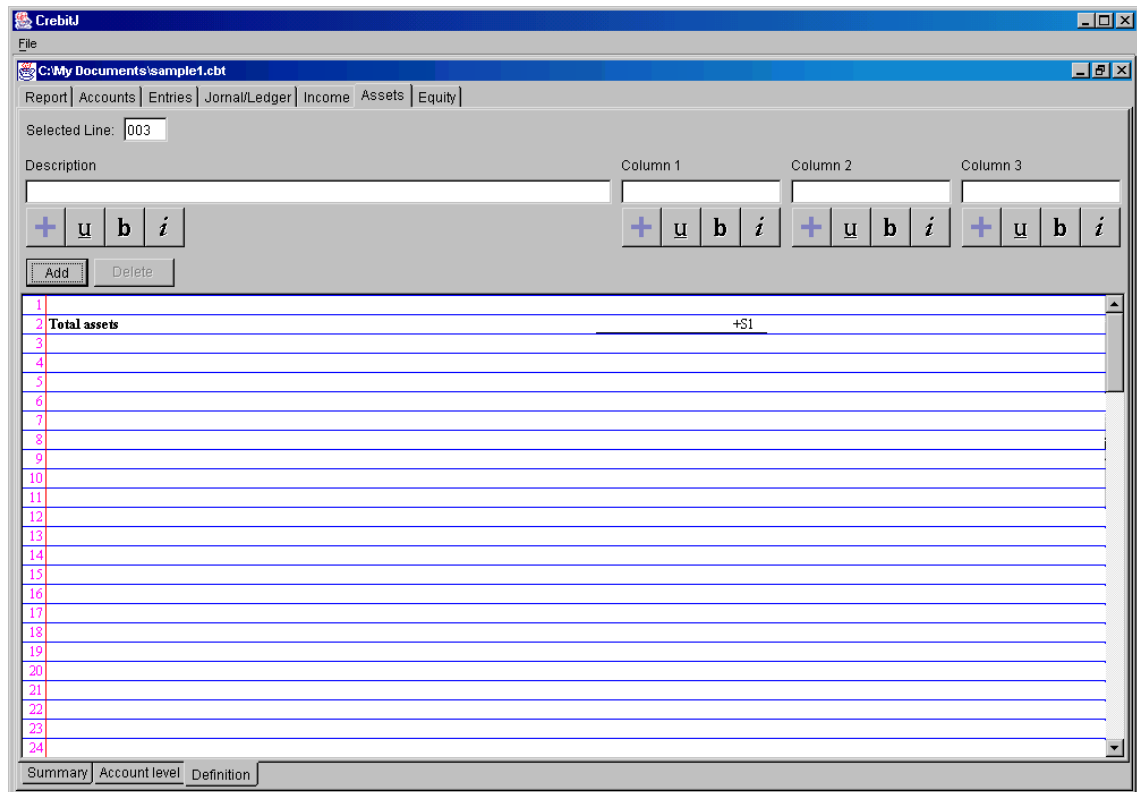
**Picture 7-7 Managing accounts**



Picture 7-8 Managing entries



Picture 7-9 Viewing journal and ledger



**Picture 7-10 Defining assets report**

The end result is a fully working user interface that connects to code generated from the model in order to create a prototype of a bookkeeping program. Nonetheless, we decided not to implement all the features of Crebit. We just feared that implementing some of the features, while adding to our time requirements, would not help us gain any further understanding of the subject being studied. For instance, our prototype currently is not able to print reports. This feature is an extension of the panel-drawing feature. Though it has not been implemented to date, it could be added quickly without any considerable time effort.

Actually, developing a user interface is not a trivial task at all as many have been led to believe. Developing a fully working, interactive and usable user interface for a certain platform is often a time-consuming and somewhat complex task. The fact that the UML says little about and can aid little in this task is one of the drawbacks of the technology. This has led some to press for this drawback to be solved alongside with others. Scott Ambler (Ambler, 2004), a senior consultant, and an expert in agile technologies said:

Why is the release of UML 2.0 a non-event?  
 Because it doesn't address any of UML 1.x's  
 gapping holes. Every single system that I've ever  
 built implements business rules and a user interface,  
 and uses a database to persist objects – yet these

three issues haven't been addressed (...). However, UML still doesn't address user interface development, a serious problem when you consider that the user interface *is* the system for most users.

Does the fact that the Executable UML methodology aided little in the development of our user interface invalidate the methodology? We believe that no. In most cases, the development of user interfaces is a specialized domain. Details about the platform in most cases prevail over general abstractions. It seems very logical to us that a team of experts developing a model for a certain domain area would delegate or outsource the development of the user interface of their systems to subcontractors specialized in the development of user interface under a certain platform. This team would better proceed in such a way than to develop the user interface by themselves. This would let them concentrate on their domain area, as a result we could expect better models than if the team had also spent considerable effort designing a user interface by itself.

## 8. CONCLUSIONS

Based on the research conducted, in this last chapter we set out to answer the questions imposed in chapter 6.

Perhaps the most important of all the questions is whether executable UML is able to raise the level of abstraction. This lies at the core of Executable UML. Mellor and Balcer expressed very clearly their vision that Executable UML would become the next level of abstraction. The following excerpt was also cited in the first chapter of this book: "Executable UML is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software so that a specification built in Executable UML can be deployed in various software environments without changes" (Mellor and Balcer, 2002, pg. 5).

Other authors, on the other hand, think otherwise. Kleppe, Warmer and Bast (2003, pg. 34) speaking of the ways UML could be used as language for platform independent model in a model-driven architecture context contented that:

The AS (Action Semantics) is not a very high-level language. In fact the concepts used are at the same abstraction level as a PSM (platform specific model). Therefore, using Executable UML has little advantage over writing the dynamics of the system in the PSM directly. You will have to write the same amount of code, at the same abstraction level.

Kleppe, Warmer and Bast were more enthusiastic about using a combination of UML and OCL. This seems natural as the authors were involved in the development of the Object Constraint Language and have authored books on the issue.

From our research, we take a view that is opposed to that of Kleppe, Warmer and Bast. We have come across manuals for action languages that are about 20 pages long and could be easily read and understood in one or two hours. It is not difficult to find guides for 3<sup>rd</sup> generation programming that have close to 1000 pages. We believe that this is evidence that supports our view that the action languages of popular Executable UML tools are in a higher level of abstraction than popular 3<sup>rd</sup> generation programming language. We also believe that it is substantially easier to learn to use one of such action languages than it is to learn and master a 3<sup>rd</sup> generation programming language.

We conclude that the answer to this question should thus be a yes: we believe that the Executable UML is able to raise the level of abstraction and as such can assist in the development of models to be translated into code to different platforms, or at least serve as basis for this vision.

Another of the research question concerned whether Executable UML could bring productivity gains. The answer to this is trickier. In fact, adopting the Executable UML methodology would in most organization require a change in their software development mentality. For those organizations whose development processes are closely aligned with the concepts of Executable UML, for instance many in the hardware and electronics industry, we believe that adopting the Executable UML methodology would result in productivity gains. Gustavsson and Lidén (2002) on their Master's thesis about Executable UML interviewed an engineer from Ericsson organization was upbeat about his experience with Executable UML. For those organizations whose current software or hardware development bears no resemblance to the concepts of Executable UML, the answer as to whether adopting Executable UML would bring gains in productivity is unclear. In our view it would depend in such factors as: the degree of expertise in UML currently found in the organization, the extent to which modeling is performed and model-experts are found in the organization and others. In the long term, we dare to say that most organizations would benefit from a greater modeling expertise, as this in the end is key to bringing and retaining more knowledge about the subject area, and Executable UML could be a valuable tool in that direction. It is my opinion unclear whether our prototype could have been developed faster had Executable UML not been used, though we believe that had we not used Executable UML there would be advantages in resorting to some other sort of modeling technique, which supports our view that not only software developers but also organizations would benefit from a greater modeling expertise in the long haul and using Executable UML could help towards developing such expertise in the organization or individual.

About the availability of tools for Executable UML, we take a negative view. It's clear so far that the availability of tools for Executable UML is rather limited to date. Besides, the available tools are often hard to get. Organizations producing Executable UML tools often impose requirements in order to provide evaluation versions that are not commonly found in the other areas of software development. The availability of model compilers to various platforms is also rather scarce. Most vendors concentrate on

developing model compilers for embedded systems platforms, usually translating to languages such C or C++. This might changes in the future as the technology matures.

Many of the shortcomings in the Executable UML methodology stem from its relative immaturity, ranging from the limited availability of tools and model compilers to the lack of a concrete standard syntax for action languages. Action languages are not difficult to learn, at least not from the standpoint of someone reasonably acquainted with the discipline information technology, but in their attempt for simplicity they seem to leave some questions unanswered. We believe that these shortcomings will be solved as the technology matures.

To finalize, we believe that the UML is a very interesting and comprehensive piece of craftwork that has been made available for the software community. Though it has gained wide acceptance and is believed to be a de facto standard when it comes to the visual modeling of software artifacts, we believe that it has not been used at a level compatible with its full potential. It is regrettable that in the current state of the art UML models are commonly used either to create rough representations of a software system to be constructed that quickly run out-of-synch with the actual system or to document a software system that has already been built. Given the investment that has been made in UML tools and expertise and the fact that UML is a comprehensive and potentially powerful technology we believe that such endeavors aiming at providing greater value from it should be welcome. Executable UML is one of such endeavors, and as such, we believe that it is a step in the right direction and brings with it the promise of greater value from UML to the software community worldwide.

## References

- AMBLER, Scott, *What's new in UML 2*, Software Development Magazine (2004).
- AMBLER, Scott, *Be Realistic About the UML*, Agile Modeling (2002), <http://www.agilemodeling.com/essays/realisticUML.htm> accessed on 15/06/2003.
- BOOCH, Grady, RUMBAUGH, James, JACOBSON, *The Unified Modeling Language Reference Manual*, Addison Wesley (1999).
- BOOCH, Grady, RUMBAUGH, James, JACOBSON, *The Unified Modeling Language User Guide*, Addison Wesley (1998).
- CANTOR, Murray, *Object Oriented Project Management with UML*, Wiley Computer Publishing (1998).
- ENGERS, Tom M. Van, GERRITS, Rik, BOEKENOOGEN, Margherita, GLASSÉÉ, Erwin, KODELAAR, Patries, *Using UML/OCL for Modeling Legislation – an application report*, ACM (2001).
- ERIKSSON, Hans-Erik, PENKER, Magnus, *Business Modeling with UML – Business Patterns at Work*, Wiley Computer Publishing (2000).
- FOWLER, Martin, *Analysis Patterns – Reusable Object Models*, Addison Wesley (1997).
- GUSTAVSSON, Jessica, LINDÉN, Simon, *Executable UML - En introduktion till morgondagens systemutveckling?* Ms. Thesis at Växjö University, Sweden (2002).
- HORROCKS, Ian, *Constructing the User Interface with Statecharts*, Addison Wesley (1998).
- Kennedy Carter Company, *Executable UML – an Online Tutorial*.
- KLEPPE, Anneke, WARMER, Jos, BAST, Wim, *MDA Explained – The Model Driven Architecture: Practice and Promise*, Addison Wesley (2002).

- KRUCHTEN, Phillipe, *A Rational Development Process*. Rational Software (2001). Rational corp.
- MELLOR, Stephen, BALCER, Marc, *Executable UML*, Addison Wesley (2002).
- NUNAMAKER, Jay, CHEN, Minder, PURDIN, Titus, *Systems Development in Information System Research*, Journal of Management Information Systems (1991).
- STARR, Leon, *Executable UML How to Build Class Models*, Prentice Hall (2002).
- TALLBERG, Anders, *Crebit 1.2 Bruksanvisning*, Swedish School of Economics and Business Administration (2003), <http://crebit.shh.fi/manual/br12.pdf> accessed on 01/02/2004.
- WARMER, Jos, KLEPPE, Anneke, *The Object Constraint Language*, Addison Wesley (1998).