Round-trip Engineering

Pasi Kellokoski

University of Tampere
Department of Computer
and Information Sciences
Master's thesis
May 2000

University of Tampere
Department of Computer and Information Sciences
Pasi Kellokoski: Round-trip Engineering
Master's thesis, 74+6 pages

May 2000

The basic problem of any software design is how to derive executable software components from requirement specification, and how this process could be supported by a computer. The object-oriented approach allows smooth shift from one process phase to another. However, problems tend to occur particularly in the design and implementation phases. It is typical that these phases aren't seamlessly integrated. A lot of manual work has to be done to transform the design artifacts into implementation artifacts. Even more work has to be done if the design artifacts are kept up-to-date with the implementation artifacts. The bottom line is that models today are expenses. They need to be assets. All the work spent on modeling doesn't pay off unless we can translate our models into executable deliverables.

With round-trip engineering and UML the programmer generates code from a UML model, changes that code in the development environment and recreates the adapted UML model back from the source code. This approach promotes better time-to-market and better quality. This thesis studies the whole problem domain of round-trip engineering, consisting of processes, methodologies and tools.

It can be concluded that full round-trip engineering with UML still has a lot of problems. This is mainly because UML is not a visual programming language but a visual modeling language. However, there are some promising methodologies to support round-trip engineering with UML. Hopefully these methodologies gain more support from tool vendors and other organizations in the future.

Keywords: object-oriented design, UML, round-trip engineering, forward engineering, reverse engineering.

Contents

Figures

Acknowledgements

I wish to thank Tellabs Oy for providing me with all the necessary resources for carrying out my thesis. Particularly my group managers Pirjo Hannikainen and Markus Mikkola deserve to be acknowledged for their support and comments.

I would also like to thank my supervisor, professor Jyrki Nummenmaa at the University of Tampere.

Tampere, May 31st, 2000.

Pasi Kellokoski

1. Introduction

The basic problem of any software design is how to derive executable software components from requirement specification, and how this process could be supported by a computer. The object-oriented approach provides a common paradigm throughout the software development process from analysis to implementation. This allows a smooth transition from one phase to another and makes it possible to use wide-spectrum tools for software development (Koskimies *et al.*, 1996). But the object-oriented approach doesn't address all the problems. Sophisticated methodologies and CASE tools could be utilized better to enable a smooth transition from design to implementation, and vice-versa especially in the iterative development process.

In this thesis I discuss round-trip engineering with the Unified Modeling Language (UML) as a way to improve an object-oriented software engineering process. Round-trip engineering consists of forward engineering and reverse engineering. With forward engineering the UML models can be transformed into source code. With reverse engineering the source code can be transformed back into UML models. UML models are designed with a CASE tool that also executes the forward engineering and reverse engineering processes. This kind of environment – round-trip engineering method supported by a powerful CASE tool – could enable seamless integration between design and implementation phases.

The need for this thesis arises from real-life software projects. Too much time is spent on modeling, transforming the models into source code and then keeping the models consistent with the implementation. A lot of errors are introduced when these activities are performed manually by the developer. Also, if the documentation is not kept consistent with the implementation, it pretty soon becomes useless. Round-trip engineering could be possible solution to these problems.

This thesis studies round-trip engineering in several aspects, as round-trip engineering has to be supported by tools, methodologies, and also utilized in a process. The practices presented in this thesis are applicable in all object-oriented programming languages although the examples are presented with the Java™ programming language. The major contributions of this thesis are related to the practices for the development of object-oriented systems using UML modeling and CASE tools.

The research problems in this thesis are:

1. How are the transitions between design and implementation phases implemented in state-of-the-art software processes?

2. Can round-trip engineering with UML solve these problems?
3. How can round-trip engineering with UML be utilized effectively?

This thesis is divided into 10 chapters. Chapter 2 introduces the essential terminology in the thesis. Chapter 3 gives a brief introduction to UML. Chapter 4 introduces round-trip engineering in general and studies how the UML constructs are mapped to the implementation language. Chapter 5 studies how round-trip engineering is utilized in a state-of-the-art software engineering process. Chapter 6 covers methodologies that support round-trip engineering. In Chapter 7, round-trip engineering tools are presented. Chapter 8 gives an example to demonstrate development with a round-trip engineering CASE tool. Chapter 9 discusses possible future research topics. Chapter 10 presents conclusions on the contents and the results of the thesis.

2. Terminology

In this chapter I clarify the essential terminology in my thesis: *software engineering process*, *software engineering methodology*, *modeling language* and *model*. A lot of controversy exists in the usage of this terminology, especially in the case of process and methodology. Some people think that *Object Modeling Technique*++, OMT++ for short (Jaaksi *et al.*, 1999), is a process, some think it's a methodology. *The Unified Modeling Language*, UML for short (UML 1.3, 1999), is actually more often referred to as a methodology than as a modeling language. What about model, view, diagram and abstraction; what is the difference between them?

Every term is discussed according the same pattern. First I briefly present the term generally, then definitions from respected software engineering publications are presented and discussed. Typical misunderstandings are clarified and then examples of correct usage of the term are given. UML class diagrams are used to represent relationships between terms. UML statechart diagrams are used to describe transitions between the different states of a term.

2.1. Software engineering process

2.1.1. Definition

Process

Etymology: **Middle English proces, from Middle French, from Latin processus, from procedere.**

Date: 14th century.

2 b: a series of actions or operations conducing to an end; especially: a continuous operation or treatment especially in manufacture.

Figure 1. Software engineering process as defined by Webster (1999).

Booch (1998) thinks that *the goal of software engineering process is to build a software product or to enhance an existing one*. Process ensures the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. The input for the process are the new or changed requirements. The output of the process is a new or changed system. Hence, the software engineering process can be considered as a machine that builds the system from requirements.

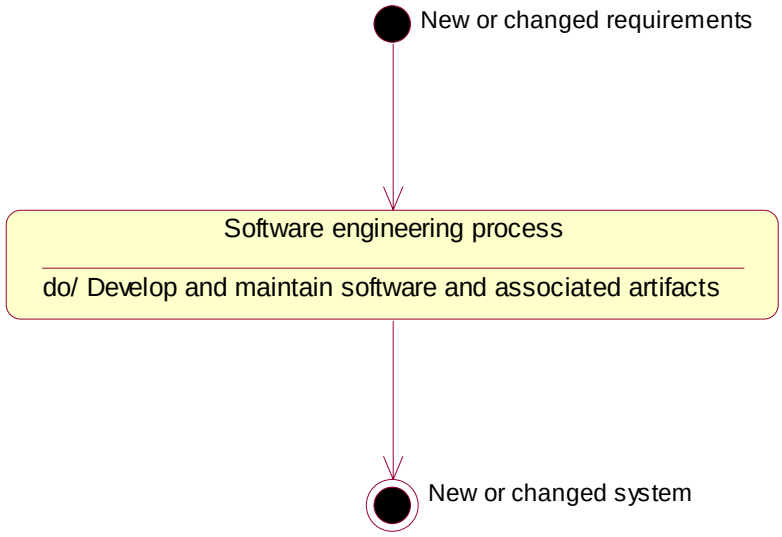

Figure 2. Software engineering process.

A good definition of software engineering process is presented by Ambler (1997): *a set of project phases, stages, methods, techniques, and practices that people employ to develop and maintain software and its associated artifacts (plans, documents, models, code, test cases, manuals, etc.).*

The key concepts in the software engineering process are (Booch, 1998):

- **Phases**. This answers the question "*when does it happen*". Typical phases are inception, elaboration, construction and transition.
- **Process workflows**. This answers the question "*what does happen*". This concept includes activities and steps. Si Alhir (1998) thinks that activities are efforts or collections of tasks directed at producing or developing artifacts. Activities establish or recommend tasks and techniques in order to accomplish objectives. Tasks specify thinking, performing, and reviewing actions to meet the goals of an activity. The workflow is divided into steps to reduce complexity. Typical process workflows are requirements, analysis, design, implementation and test.
- **Actifacts**. This answers the question "*what is produced*". Si Alhir (1998) thinks that artifacts are work products or deliverables resulting from efforts, they are expressed in languages such as the UML. Typical artifacts are models, reports and documents.
- **Workers**. This answers the question "*who does it*". Typical workers in a software engineering process are project managers, analysts, architects, designers and testers.

Figure 3 illustrates phases and workflows in *The Rational Unified Process*, RUP for short (Kruchten, 2000).

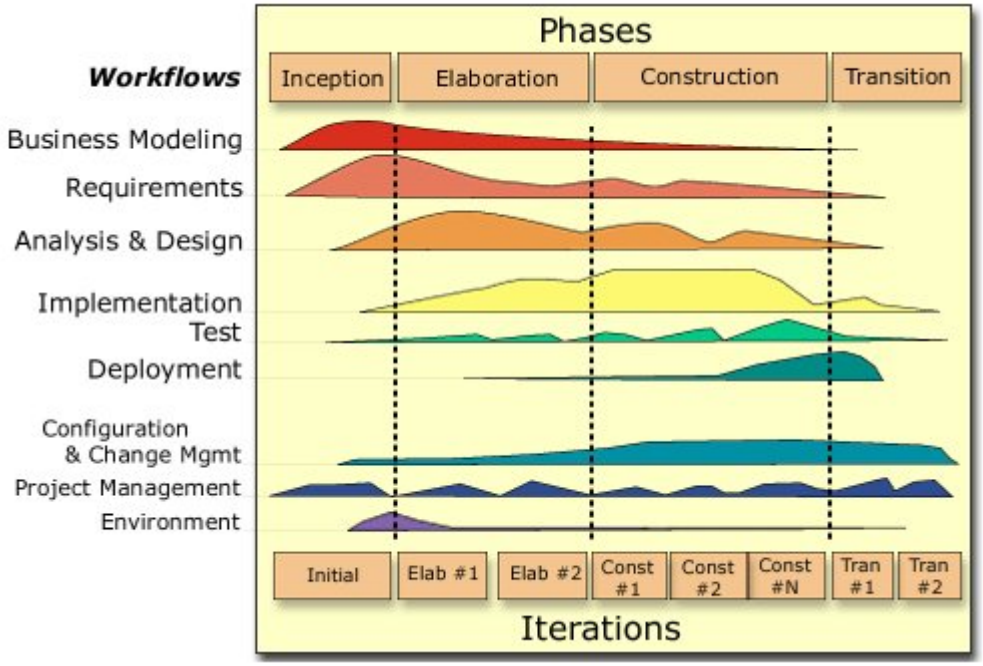

Figure 3. Phases and workflows in RUP (Kruchten, 2000).

Booch's definition of software engineering process doesn't clearly answer the question "how is it produced". Si Alhir (1998) emphasizes *methods, rules and guidelines* as a way to facilitate problem solving. He thinks that methods should be used to provide an infrastructure for the problem-solving process. Si Alhir (1998) thinks that process provides guidance regarding:

- *the order of activities,*
- *specification of artifacts to be developed,*
- *direction of individual developer and team tasks (or activities), and*
- *monitoring and measuring criteria of project artifacts and activities.*

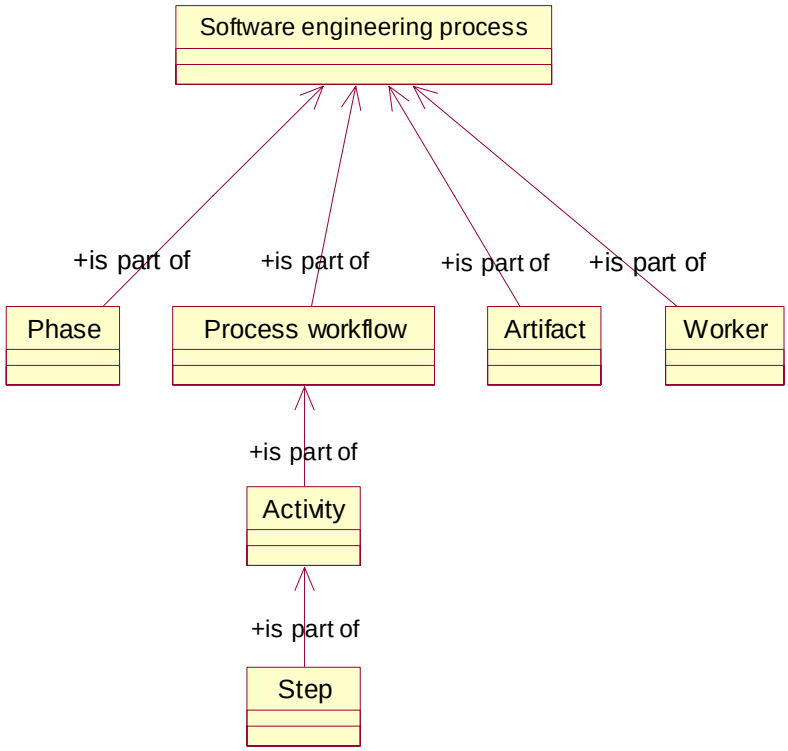

Figure 4. Software engineering process.

2.1.2. Discussion

The term software engineering process is often confused with terms methodology and notation, especially in the case of OMT++ and UML. Another typical misunderstanding is that processes can be taken off-the-shelf to be deployed in an organization.

OMT++ is a software engineering process although it is more often referred to as a methodology. According to Jaaksi *et al.* (1999), OMT++ is a collection of practices, selected notations, and phase-product templates that assist software developers in their everyday work. It also forms the backbone of software project management and quality assurance. All the software development work is planned, managed, and performed according to its phases.

UML is not a software engineering process, it's a modeling language. UML is process independent, but it enables and promotes a use-case driven, architecture-centric, iterative and incremental process that is object-oriented and component based (Si Alhir, 1998). However, Fowler and Scott (1999) emphasize the meaning of process in the usage of UML. They think that diagrams don't make much sense without a process to give them context. Many software engineering processes (OMT++ and The Rational Unified Process, for example) use the notation of UML.

It is clear that there is no universal, one-size-fits-all software engineering process. Processes by their very nature must be tailored to the organization, culture, and problem domain at hand. What works in one context (shrink-wrapped software development, for example) would be a disaster in another (hard-real-time, human-rated systems, for example). The selection of a particular process will vary greatly, depending on such things as problem domain, implementation technology, and skills of the team. (UML 1.3, 1999)

There has been some convergence on development process practices, but there is not yet consensus for standardization. According to UML 1.3 (1999) the result is likely to be a general agreement on best practices and potentially the embracing of a process framework, within which individual processes can be instantiated. Fowler and Scott (1999) don't see a need for a standard process, but they think that some harmonization on vocabulary would be useful. Process frameworks allow a variety of lifecycle strategies, select which artifacts to produce, define activities and workers, model concepts (Jacobson, 1999). Some processes consider themselves more as process frameworks (The Rational Unified Process for example) just because the process is designed for flexibility and extensibility.

The leading object-oriented software engineering processes include RUP (Rational Unified Process) (Kruchten, 2000), OMT++ (Object Modeling Technique++) (Aalto and Jaaksi, 1994) and OPEN (Object-Oriented Process, Environment and Notation) (Graham *et al*., 1997).

2.2. Software engineering methodology

2.2.1. Definition

Methodology

Etymology: New Latin methodologia, from Latin methodus + -logia –logy.

Date: 1800.

1: a body of methods, rules, and postulates employed by a discipline: a particular procedure or set of procedures.

2: the analysis of the principles or procedures of inquiry in a particular field.

Figure 5. Software engineering methodology as defined by Webster (1999).

Software development is about solving problems. Si Alhir (1999) suggests that since projects can be very complex, methods should be used to provide an infrastructure for the problem-solving process. They should be considered as suggestions and recommendations that organize and facilitate the problem-solving process rather than being considered rigid and inflexible rules that restrict the art of problem solving.

Software engineering methodology is *a process for organized production of software, using a collection of predefined techniques and notational conventions.* A methodology is usually presented as a series of steps, with techniques and notation associated with each step. The steps of software production are usually organized into a life cycle consisting of several phases of development. The complete software life cycle spans from initial formulation of the problem, through analysis, design, implementation, and testing of the software, followed by an operational phase during which maintenance and enhancement are performed. (Rumbaugh *et al.*, 1991)

Software engineering methodology can also be defined as *a taxonomy, or well-organized collection, of related methods.* Methods specify how to conduct problem-solving efforts. They specify an overall problem-solving approach and its components. They specify how problems and solutions are viewed in relation to the problem-solving approach; this is known as a method's descriptive aspect since it describes how knowledge is captured and communicated regarding a problem and solution. Methods also specify a problem-solving approach to be used to solve the problem and derive a solution; this is known as a method's prescriptive aspect since it prescribes how knowledge is leveraged to solve a problem. Methods specify descriptively how problems and solutions are viewed, and prescriptively how the problem-solving effort may be actualized. (Si Alhir, 1998)

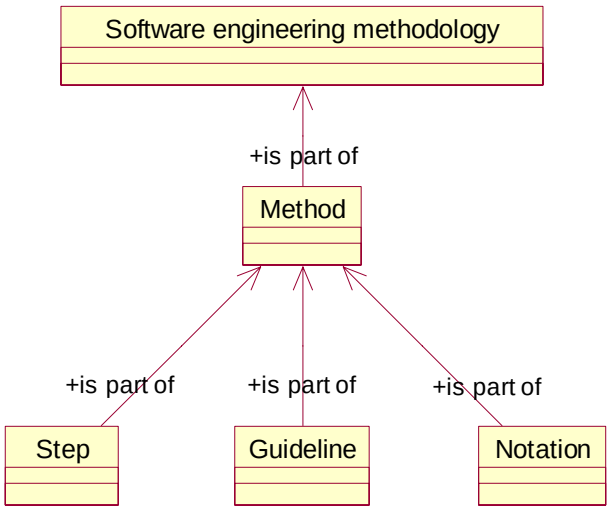

Figure 6. Software engineering methodology.

2.2.2. Discussion

It is difficult to draw a line between the terms software engineering process and software engineering methodology. Even the authors of OMT++ are uncertain

whether OMT++ is a methodology or a process. Fowler and Scott (1999) also think that the terms methodology and process are often confusing. They find that most people, when they say they are using a method, use the modeling language, but rarely follow the process.

According to Rumbaugh *et al.* (1991), software engineering methodology includes many of the same concepts as process. *The Object Modeling Technique*, OMT for short (Rumbaugh *et al.,* 1991), includes same concepts as in process definitions: phases, steps, guidelines (techniques and notational conventions).

Most methods consist of both a modeling language and a process. The modeling language is the (mainly graphical) notation that methods use to express designs. The process is their advice on what steps to take in doing a design. Process is an important part of a method. (Fowler and Scott, 1999)

What makes a difference between software engineering process and software engineering methodology? The methodology definition from Rumbaugh *et al.* (1991) emphasizes the collection of predefined techniques and notational conventions. Si Alhir (1998) also defines a methodology from this point of view; as a collection of related methods. Problem-solving methods aren't defined strictly in software engineering process definition. Process definitions emphasize the workers concept ("who does it"), but methodology definitions don't address this concept at all. I think that the main difference here is that software engineering methodologies are concentrated on techniques and notational conventions whereas software engineering processes are more focused on phases and workers.

Popular object-oriented software engineering methodologies include Object-Oriented Analysis and Design (OOA/OOD) (Coad and Yourdon, 1991a, 1991b), OMT (Rumbaugh *et al.*, 1991), and Fusion (Coleman *et al.*, 1993).

2.3. Modeling language

2.3.1. Definition

Language

Etymology: Middle English, from Old French, from langue tongue, language, from Latin lingua

Date: 14th century

1 a: the words, their pronunciation, and the methods of combining them used and understood by a community

1 b (2): a systematic means of communicating ideas or feelings by the use of conventionalized signs, sounds, gestures, or marks having understood meanings

1 b (3): the suggestion by objects, actions, or conditions of associated ideas or feelings

1 b (5): a formal system of signs and symbols (as FORTRAN or a calculus in logic) including rules for the formation and transformation of admissible expressions

2 a: form or manner of verbal expression; specifically: STYLE

2 b: the vocabulary and phraseology belonging to an art or a department of knowledge

Figure 7. Language as defined by Webster (1999).

Good communication along with good understanding is the key to developing good software. Languages are used for communication, they are means for expressing and communicating content or information. Fowler and Scott (1999) think that natural language is too imprecise when it comes to more complex concepts. Code is precise but too detailed. A modeling language is an ideal solution for communicating concepts in object-oriented software development.

UML 1.3 (1999) suggests that modeling language includes *model elements, notation and guidelines.* According to Si Alhir (1998), a modeling language consists of:

- *concepts and semantics* that are communicated,
- *a notation or syntax* used to render semantics for communication,
- *rules or guidelines* expressing idioms of usage and how syntactical constructs are combined in forming the content of communication.

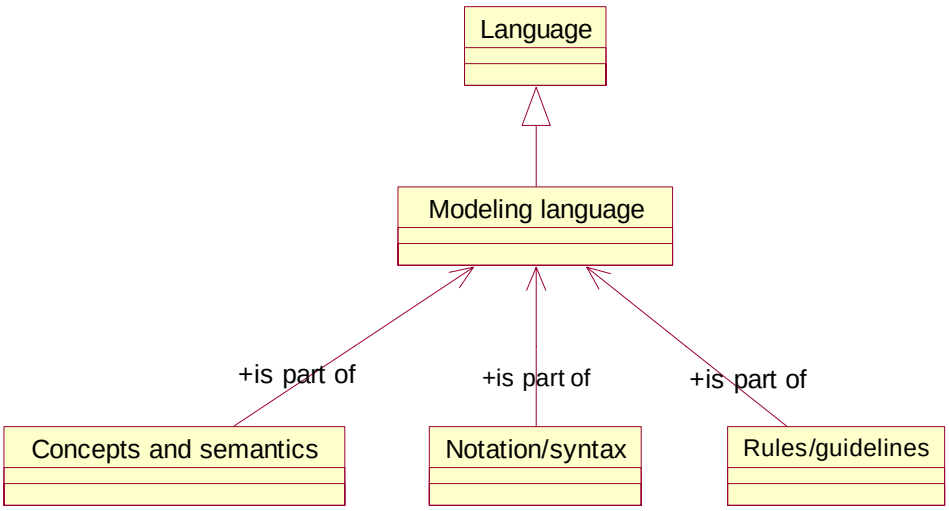

Figure 8. Modeling language.

2.3.2. Discussion

Modeling language and notation are two terms that have almost the same meaning.

Ambler (1997) defines notation as *the set of symbols that are used to document the analysis/design of a system*. UML 1.3 (1999) defines notation *as visual rendering*

of model elements. Jaaksi *et al.* (1999) think that notation is the key tool for software developers. It enables visual appearance for abstract concepts and enables people to communicate and document creative work. Object-oriented languages, such as C++, Java, and Smalltalk, make the implementation of the reality-based structures and behavioral patterns possible. Notations help in illustrating these structures and patterns in different phases of software development. Therefore, notation is mainly a means or a tool that enables people to communicate and document creative work. The relationship between modeling language and notation is the following: *notation is a part of modeling language.*

UML is often considered as a notation although it is a modeling language. It is more than just the notation and syntax; it also contains model elements, various guidelines or idioms of usage and best practices (Si Alhir, 1998).

UML has become the standard object-oriented modeling language. It is often said that the English language is the world's universal language; now it is virtually certain that the UML will be the information systems and technology world's universal language (Si Alhir, 1998). Other popular object-oriented modeling languages include the modeling languages of previously leading methodologies.

2.4. Model

2.4.1. Definition

Model

Etymology: Middle French modelle, from Old Italian modello, from (assumed) Vulgar Latin modellus, from Latin modulus small measure, from modus.

Date: 1575.

1 obsolete: a set of plans for a building.

3: structural design <a home on the model of an old farmhouse>.

4: a usually miniature representation of something; also: a pattern of something to be made.

5: an example for imitation or emulation.

11: a description or analogy used to help visualize something (as an atom) that cannot be directly observed.

12: a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs.

Figure 9. Model as defined by Webster (1999).

Models are used for a number of reasons: testing a physical entity before building it, communication with customers, visualization and reduction of complexity. Because a model omits nonessential details, it is easier to

manipulate than the original entity. Engineers, artists, and craftsmen have built models for thousands of years to try out designs before executing them. (Rumbaugh *et al.*, 1991; UML 1.3, 1999)

A couple of easy-to-understand definitions for model are as follows:

- *Blueprints of systems used for systems construction and renovation.* (Si Alhir, 1998)
- *An abstraction of something for the purpose of understanding it before building it.* (Rumbaugh *et al.*, 1991)
- *The language of the designer. Representations of the system to-be-built or as-built. Captures knowledge regarding a system or context. Represents knowledge of problems, solutions, and the context in which they exist and are addressed.* (Booch, 1998)

2.4.2. Discussion

Confusing terms with model are *abstraction*, *diagram*, and *view*.

UML 1.3 (1999) defines *abstraction* as the essential characteristics of an entity that distinguish it from all other kinds of entities. Abstraction is the focus on relevant details while ignoring others. Rumbaugh *et al.* (1991) think that abstraction is a fundamental human capability that permits us to deal with complexity. Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant. All abstractions are subsets of reality selected for a particular purpose.

UML 1.3 (1999) defines *diagram* as a graphical presentation of a collection of model elements, most often rendered as a connected graph of arcs (relationships) and vertices (other model elements). Booch (1998) defines diagram as a view into a model. Diagram is presented from the aspect of a particular stakeholder, it provides a partial representation of the system.

Views are perspectives through which models may be represented or projected as diagrams. Views organize perspectives or views of models around specific sets of concerns particular to different stakeholders involved in the problem-solving effort. (Si Alhir, 1998)

Diagrams are sometimes called models because they render models or are the visual representation of models. Views are often referred to as models because they ultimately map back to a subset of a whole diagram. (Si Alhir, 1998)

3. UML

UML is considered familiar to the reader, so only a brief introduction is presented here. In this thesis the main UML reference is *The Unified Model Specification Version 1.3* (UML 1.3, 1999). Other UML references used here are *The Unified Modeling Language User Guide* (Booch *et al.*, 1998), *UML Distilled* (Fowler and Scott, 1999), and *UML in a nutshell* (Si Alhir, 1998).

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML gives you a standard way to write system's blueprints, covering conceptual things, such as business processes and system functions, as well as concrete things, such as classes written in a specific programming language, database schemas, and reusable software components. (Booch *et al.*, 1998)

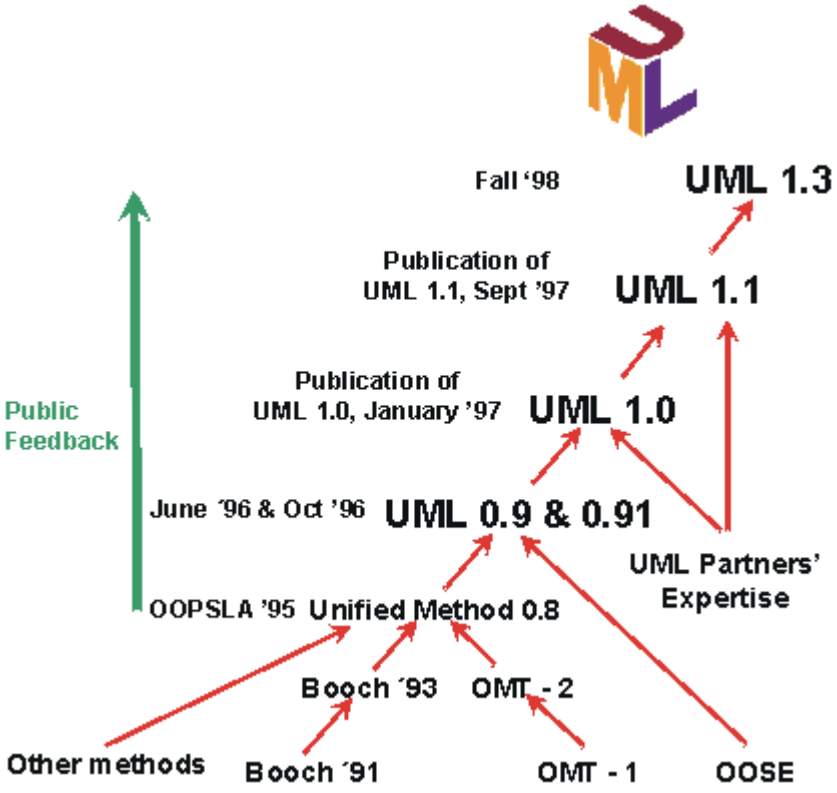

Figure 10. The history of UML (Booch, 1998).

UML is the successor to the wave of object-oriented analysis and design (OOA/OOD) methods that appeared in the late '80s and early '90s. It most directly unifies the methodologies of Booch, Rumbaugh, and Jacobson, but its reach is wider than that. The UML went through a standardization process with the Object Management Group (OMG) and is now an official standard. The history of UML is illustrated in Figure 10. (Fowler and Scott, 1999)

The benefits of visual modeling and UML are indisputable. Booch *et al.* (1998) think that we build models so that we can better understand the system we are developing. According to Quattrani (1999), modeling promotes better understanding of requirements, cleaner designs, and more maintainable systems. Through modeling, we achieve four aims (Booch *et al.*, 1998):

1. Models help us to visualize a system as it is or as we want it to be.
2. Models permit us to specify the structure or behavior of a system.
3. Models give us template that guides us in constructing a system.
4. Models document the decisions we have made.

The value of UML is the following (Booch, 1998):

- UML is an open standard,
- UML supports the entire software development life-cycle,
- UML supports diverse application areas,
- UML is based on experience and needs of the user community, and
- UML is supported by many tools.

The UML provides a very robust notation, which grows from analysis into design. The UML includes nine diagrams: class, object, use case, sequence diagram, collaboration diagram, statechart, activity, component and deployment diagrams.

4. Round-trip engineering

4.1. General

In this chapter I introduce round-trip engineering, mappings between UML and the source code, and generally agreed advantages and disadvantages of round-trip engineering. So far, there has been very little research on round-trip engineering. The major references in this chapter are *The Unified Modeling Language User Guide* (Booch *et al.*, 1998), *UML shortcoming for coping with round-trip engineering (*Demeyer *et al.*, 1999), and *Universal Object Language 1.2. specification* (UOL 1.2, 1998).

According to Booch *et al.* (1998) modeling is important, but it is important to remember that the primary product of a development team is software, not diagrams. Of course, the reason you create models is to predictably deliver at the right time the right software that satisfies the evolving goals of its users and the business. The bottom line is that models today are expenses. They need to be assets. All the modeling doesn't pay off unless we can translate our UML models into executable deliverables. The usefulness of UML really starts to play into the life of a developer when tied to another technology, such as round-trip engineering.

Although UML is not a visual programming language, its models can be directly connected to a variety of programming languages. It is possible to map from a model in the UML to a programming language such as Java, C++, or Visual Basic, or even to tables in a relational database. This mapping permits *forward engineering*: the generation of code from a UML model into a programming language. *Reverse engineering* is also possible: you can reconstruct a model from an implementation back into the UML. Combining the two paths yields *round-trip engineering,* meaning the ability to work in either a graphical or a textual view, while tools keep the two views consistent. (Booch *et al.*, 1998)

Demeyer *et al.* (1999) defines round-trip engineering as follows: *the seamless integration between design diagrams and source code, between modeling and implementation.* With round-trip engineering and UML, the programmer generates code from a UML model, changes that code in the development environment and recreates the adapted UML model back from the source code. The object-oriented development processes with their emphasis on iterative development make round-trip engineering a relevant issue.

Round-trip engineering (and especially forward engineering) is often referred to as code generation. Code generation methods are described as push (this means the same as forward engineering), pull (reverse engineering), and

push-pull (round-trip engineering). The term round-trip engineering (later also RTE) is usually used only with UML although round-trip engineering is used with other technologies as well, for example in graphical user interface (GUI) design, database design and with other modeling languages. In this thesis, round-trip engineering studied only in the context of UML.

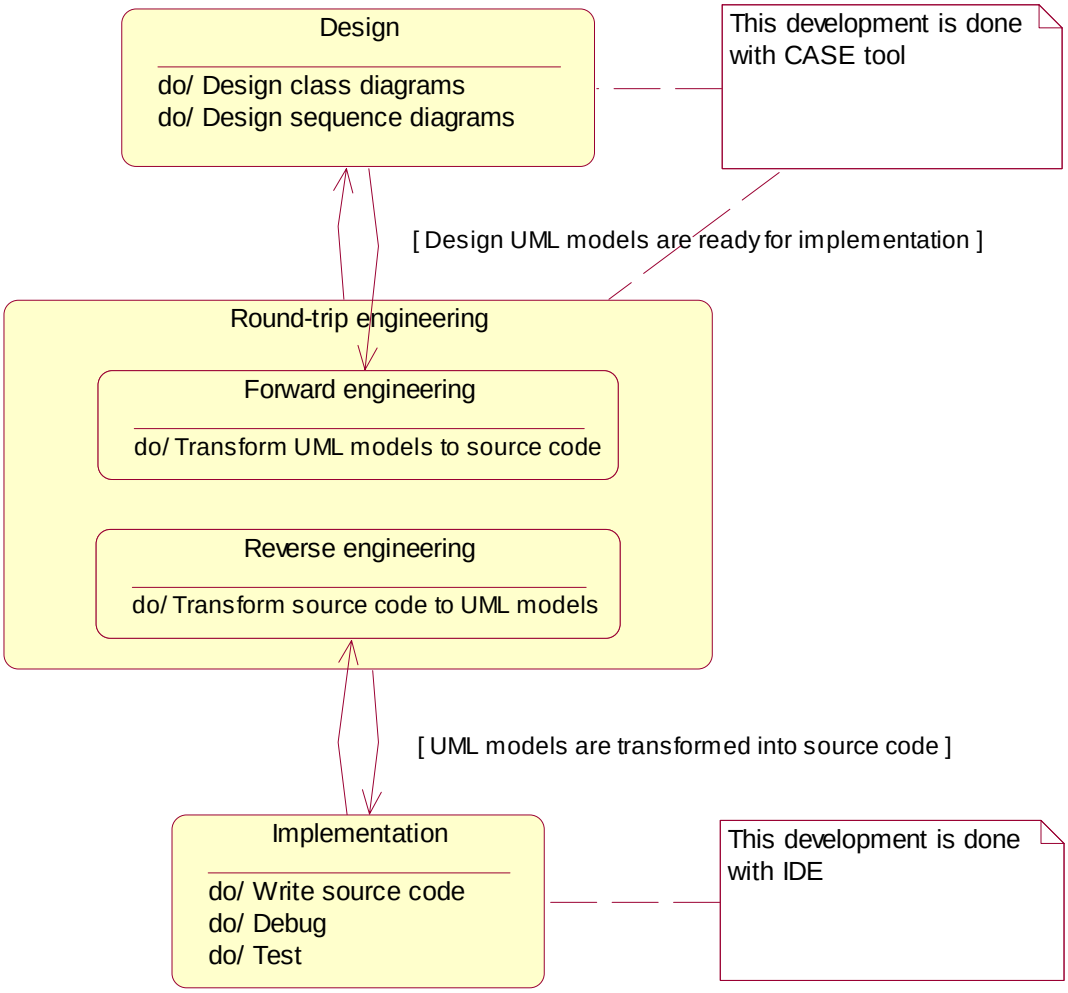

Figure 11. Round-trip engineering.

UOL 1.2 (1998) suggests that round-trip engineering is used in two different situations:

1. When developing a program, software engineers start by working with Computer Aided Software Engineering (CASE) tools to describe graphically and document the analysis and design. With a full life-cycle CASE tool they then proceed to generate code from their design. At this point they start doing testing and debugging with the language compilers. Usually they will introduce changes that must be reflected in the designs stored in the CASE tool repository. Therefore, if they want to have both model and source

synchronized, they must either re-import the source code restructuring the model or, at least, reflect in the model the changes that have been produced during the testing phase.

2. Although the first situation is the one most discussed in the industry, one should also consider the inverse, especially in an iterative life-cycle as we usually apply in OO development. When we start a new iteration of the system being developed, our main efforts will start again at analysis and design. Naturally, when we augment our system with new aspects, we will introduce changes to the previous model. Since this model has been translated to source code and debugged, it will be necessary to reflect in the source code the changes in the model. That is the precise inverse situation to that we have previously described and it is mandatory that the model reflects exactly what is implemented in the program to be able to generate the necessary changes to the program or a new version of the program.

4.2. The mappings between UML and the source code

4.2.1. General

The UML does not specify a particular mapping to any object-oriented programming language, but the UML was designed with such mappings in mind. This is especially true for class diagrams, whose contents have a clear mapping to all the industrial-strength object-oriented languages, such as Java, C++, Smalltalk, Eiffel, Ada, ObjectPascal, and Forte. The UML was also designed to map to a variety of commercial object-based languages, such as Visual Basic. For some uses of the UML, the models you create will never map to code. For example, if you are modeling a business process using activity diagrams, many of the activities you model will involve people, not computers. (Booch *et al.*, 1998)

Use case diagrams reflect rather than specify the implementation of a system, subsystem, or class. Use cases describe how an element behaves, not how that behavior is implemented. Therefore, use case diagrams cannot be directly forward or reverse engineered. (Booch *et al.*, 1998)

Forward engineering is possible for sequence, collaboration and activity diagrams, especially if the context of the diagram is an operation. Reverse engineering is also possible for these diagrams, especially if the context of the code is the body of an operation. (Booch *et al.*, 1998)

Forward engineering is possible for statechart, especially if the context of the diagram is a class. This requires a little cleverness from the tool. The forward engineering tool must generate the necessary private attributes and final static constants. Reverse engineering is theoretically possible, but

practically not very useful. The choice of what constitutes a meaningful state is in the eye of the designer. Reverse engineering tools have no capacity for abstraction and therefore cannot automatically produce meaningful statechart diagrams.(Booch *et al.*, 1998)

Forward engineering and reverse engineering for component diagrams is pretty direct because components are themselves physical things (executables, libraries, tables, files, and documents) that are therefore close to the running system. When you forward engineer a class or a collaboration, you really forward engineer to a component that represents the source code, binary library, or executable for that class or collaboration. Similarly, when you reverse engineer source code, binary libraries, or executables, you really reverse engineer to a component or set of components that, in turn, trace to classes or collaborations. (Booch *et al.*, 1998)

There's only a modest amount of forward engineering that you can do with deployment diagrams. Reverse engineering from the real world back to deployment diagrams is of tremendous value, especially for fully distributed systems that are under constant change. (Booch *et al.*, 1998)

4.2.2. Example

As there is no standard mapping between UML and implementation languages, each CASE tool has to define its own proprietary mapping. This subsection presents how Rational Rose 98i Enterprise Edition with Rose J add-in models Java™ programming language constructs. This subsection is based on Rose 98i, (1998).

Rational Rose's round-trip engineering is limited to class diagrams only. The mapping between class diagrams in UML and the Java™ programming language constructs is quite straightforward. The definitions in the UML models are mapped to corresponding Java constructs in Rational Rose's forward engineering. Reverse engineering is executed the same way (but vice versa). In some constructs (especially in associations) the developer has to define extra properties because the mapping between UML and Java isn't unambiguous.

Rational Rose models a Java variable either as an attribute of a class or as a role in an association between two classes. The difference between the two is based on the type you want to declare for the variable, either a primitive type or a reference type.

The semantics Rational Rose supports for variables are:

- **Modifiers**: `public` (default), `abstract`, `final`, and `static`.
- **Generators**: finalizer, static initializer, instance initializer, default constructor.

- **Constructor Visibility**. You can set the visibility to public (default), private, protected, or package.
- **Extends**. You can indicate if a class extends another class.
- **Implements**. You can indicate if a class implements one or more interfaces.
- **DocComment**. You can annotate the class declaration. The comments you add can be used by the JavaDoc feature to create HTML documentation.

If the variable's type in a class diagram is a primitive type (`int`, `byte`, `short`, `long`, `float`, `double`, `char`, or `boolean`), the variable is modeled as an attribute of the class. If the variable's type is a reference to another entity (such as another class, interface, or array), the construct is modeled by creating an association between the class whose variable is defined and the class that models the entity that is referenced. In Figure 12, `timeOfYear` is a variable of the `HelloWorldApp` class. The variable's type is the class `Season`.

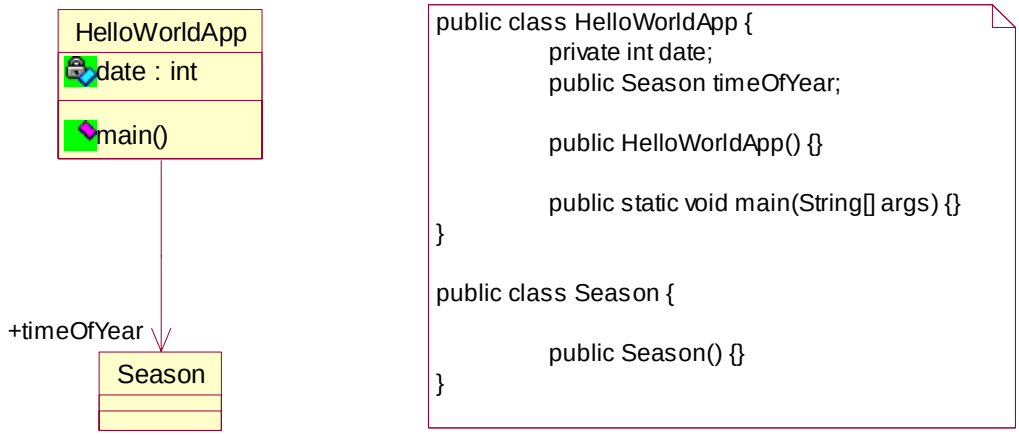

Figure 12. Java relationships and the mapping to UML.

To create an array of objects, you would create a variable exactly as described for a variable with a referenced type. Namely, you would draw the association between supplier and client, name a role as the array, and disable the other role's navigability. To generate the variable as an array vs. a variable, you set array role's cardinality to a value greater than 1 (for example, n). The cardinality is the setting that triggers Rose to generate the array. See Figure 13 for an example.

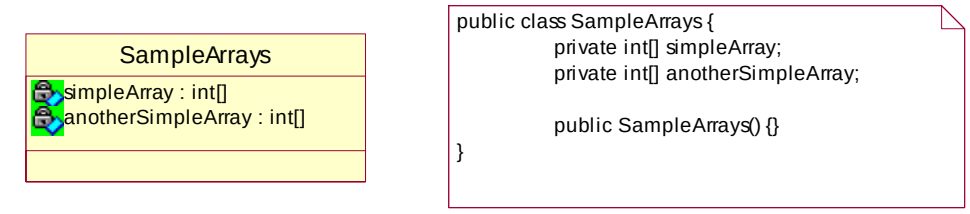

Figure 13. Java arrays and the mapping to UML.

You can create a vector by creating the same association between classes as for an array, but identifying a container class, such the `Vector` Java class from `java.util`. See Figure 14 for an example.

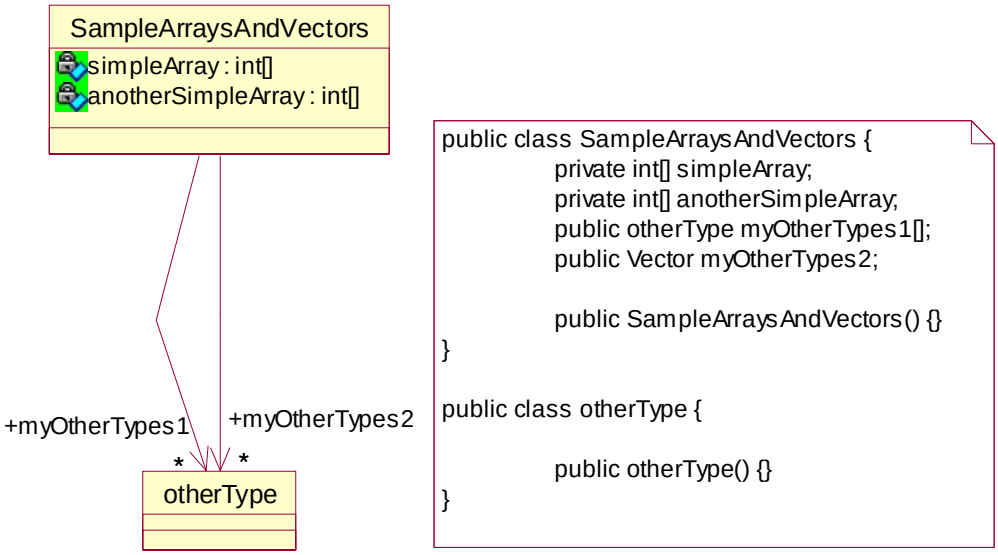

Figure 14. Java vectors and the mapping to UML.

Rational Rose models Java methods as operations (see Figure 15).

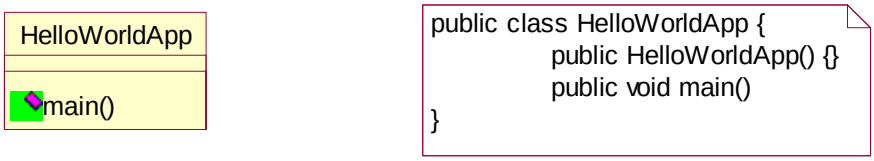

Figure 15. Java methods and the mapping to UML.

The semantics Rational Rose supports for methods are:

- **Return Type**. You can opt for the default (void), or select from the types defined in your model.
- **Modifiers**: `Abstract`, `final`, `static`, `synchronized`, `native`.
- **Visibility**: `Public`, `private`, `protected`, `package`.
- **Arguments.**

- **Throws**. You can identify the exception classes to associate with the method.
- **DocComment**. You can annotate the class declaration. The comments you add can be used by the JavaDoc feature to create HTML documentation.

Rational Rose models the Java implements relationship as a UML realization relationship between a subclass and a superclass that has a stereotype of `Interface` (see Figure 16).

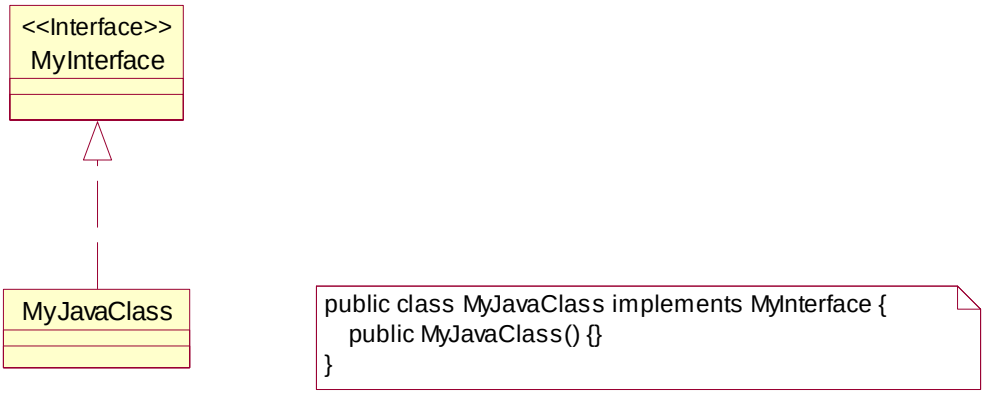

Figure 16. The implements relationship in Java and the mapping to UML.

Rational Rose models the Java extends relationships as a UML generalization relationships (see Figure 17).

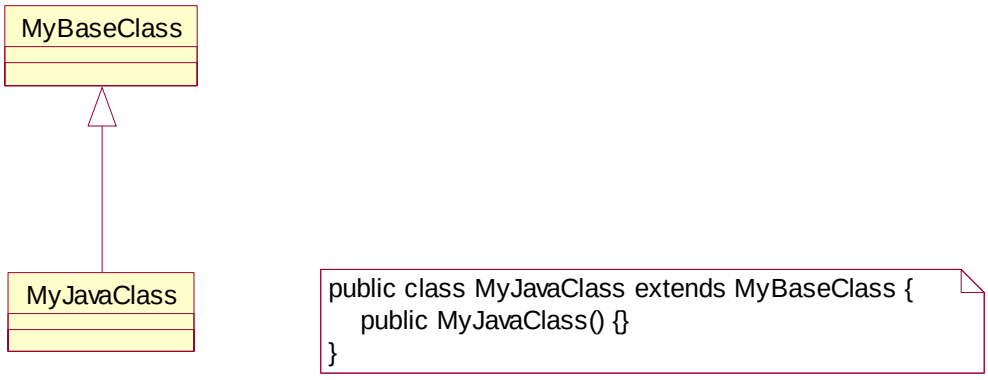

Figure 17. The extends relationship in Java and the mapping to UML.

4.3. Discussion

4.3.1. General

According to Boyd (1999), software developers translate ideas into code. On one side of this gap is the natural language used to describe customer problems and solution requirements. On the other side of the gap are the formal

languages used in software development for analysis, design and implementation. With round-trip engineering this gap can be bridged, at least between design and implementation.

Software development invariably begins with some human need or desire - to explore or solve a problem, to automate a business process, to entertain, to educate, to communicate, to transact and track commerce, to explore and share information and knowledge, etc. The human needs that have been touched by computers are truly varied and numerous. However, we often have difficulty describing precisely what we want and need computers to do, and why. (Boyd, 1999)

We use natural language to describe our needs and problems, but it's often complex, vague and ambiguous. Sentences are complex when they contain clauses and phrases that describe and relate several objects, conditions, events and / or actions. Sentences are vague when they contain generalizations, or they are missing important information, especially the subject or objects needed by a verb for completeness. Sentences are ambiguous when they are open to multiple interpretations. All these troubles arise (naturally) when we discuss our needs and problems using natural language. On the other hand, software requires more precision, formality and simplicity than what is commonly found in natural language. Computer programming languages typically limit expressions to a few simple computational primitives and their combinations. Computational primitives are often hardwired into the syntax of a programming language. Software components and classes can be built from these primitives, but their construction is typically tedious and error prone. Also, the translation from requirements to code often reduces, eliminates or distorts much of the original meaning intended by the requirements. (Boyd, 1999)

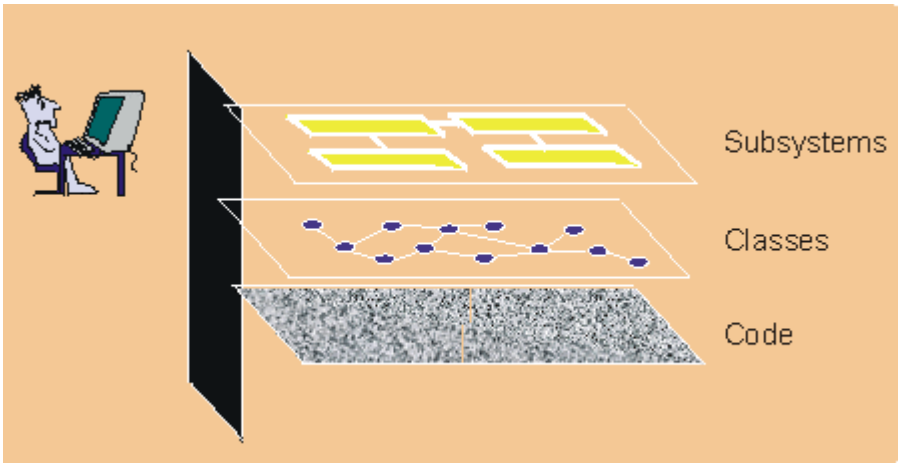

Figure 18. Visual modeling raises the level of abstraction. (Kruchten, 2000)

The translation of natural language descriptions into usable software poses quite a challenge. We need ways to reduce ambiguity and complexity without sacrificing the richness and meaning of natural language. We need to be able to clearly and consistently express and share our intentions: our purposes, objectives and goals, and the obstacles and problems that provide opportunities for software-based solutions. Ideally, programming languages will eventually increase our ability to approximate the expressive power of natural language. Meanwhile, we need ways to systematically map our rich and meaningful ideas into the limited forms of expression supported by programming languages. (Boyd, 1999)

4.3.2. Advantages of round-trip engineering

The main advantage of round-trip engineering is that the design and implementation artifacts are automatically synchronized all the time. This feature is especially important in an iterative development process where the software is prototyped, tested, measured, and analyzed, and then refined in subsequent iterations. Other important advantages include automatic generation of source code from UML models (forward engineering), automatic generation of UML models from source code (reverse engineering), improved design-led development and better traceability of design. All of these advantages promote quicker time-to-market and better quality of the product.

Automation is the traditional industrial means for improving productivity and product quality. With round-trip engineering the design and implementation can be automatically kept up-to-date all the time. What usually happens (without round-trip engineering) is that there is no time to keep the design UML models and documents up-to-date in the implementation phase of the software engineering process. Thus, models pretty soon become untrustworthy or even useless. Haikala and Märijärvi (1998) think that documentation is one of the weakest links in software projects. Documentation is as essential to the application as the source code itself. It’s very important to keep the UML models up-to-date because of communication purposes. It is also quite difficult to build upon previously constructed code if the documentation is not up-to-date.

With forward engineering the developer can automatically derive the implementation from the design. By transforming the UML models automatically to source code, the developer can prevent errors and save a lot of time. Errors are likely to occur if the UML models are manually transformed into source code. The implementation can be a skeleton implementation or even the complete application, depending on the level of accuracy in the UML diagrams.

With reverse engineering the developer can automatically derive the design from the implementation. This is typically used in the case of legacy systems. To develop software systems from scratch in most cases remains an impossible dream: legacy systems tend to live longer. Thus, reverse engineering is getting increasingly important in software engineering. The UML models aid greatly in understanding and learning the legacy system. The developer can start by reverse engineering the system. Then he can make changes to UML models, reflect these changes to the implementation and continue this development iteratively. It is also possible to derive other information besides UML models from the implementation, for example metrics and design patterns. Metrics can give valuable information about the program, thus helping to steer and control the software process. Design patterns help understand large designs, usually requiring collaborations of several types of objects, that occur frequently in software

Round-trip engineering promotes design-led development. There are some tremendous benefits from using a model-based approach over a code-based approach to designing and maintaining systems and these benefits increase with the complexity of the systems under development. Over the last 10 years or so the programming paradigm has significantly changed to this direction: design is getting increasingly important. Instead of procedural programming the task of the developer has shifted to the analysis and design of the application. It is more or less up to the code generators (forward engineering) to produce the production system. Emphasis on design rather than implementation brings better quality and saves time in the long run. Too often organizations just crank out the code, realizing afterward that careful design would have been a good idea to save time on rework later.

Round-trip engineering improves design traceability. When the implementation is seamlessly integrated into the design, it is possible to trace the design all the way back to the higher level designs and finally to system requirements. This enables enterprise-wide understanding and perspective to the whole development, also for management and business people. Without round-trip engineering the implementation of a system is often not consistent with the design and traceability from source code back to higher level abstractions is difficult, if not impossible. Without round-trip engineering the implementation will easily begin to deviate from the specification, and the final system doesn’t reflect the original requirements. According to TogetherSoft (2000), the ability to trace requirements into the software artifacts is usually done to achieve the following goals:

- Understanding why parts of the model are being developed.

- Help group related portions of the development effort back to requirements.
- Determine the dependencies associated with a requirement. Should that requirement change, you can then perform an impact analysis.
- Help drive test cases based on up-front requirements / features.

As requirements change, you may want to be able to track things like affected items (use cases, classes, operations, etc.) and impact on design. This way you can provide information like implementation effort, affect of release schedule, impact on maintenance and how change will be built into the system and released to end-users. (TogetherSoft, 2000)

4.3.3. Disadvantages/problems

The main problem in round-trip engineering is that UML is not a visual programming language. One-to-one mapping between UML and the implementation language is not possible. Also, a lot of precision is required in the UML models to enable round-trip engineering.

UML is not a sufficient modeling language for round-trip engineering. UML is visual modeling language, not a visual programming language. UML contains more than the implementation language and vice versa, the implementation contains more than the UML. What is described by the modeling language should be identical to what is described in the implementation language. Problems arise when the model contains important information to understanding of the objects that the language can’t support. According to Booch *et al.* (1998), forward engineering results in a loss of information, because models written in the UML are semantically richer than any current object-oriented programming language. Conversely, there are things that the code describes that have no place in model. Booch *et al.* (1998) suggest that it's easy to get too much information from simple reverse engineering, and so the hard part is being clever about what details to keep. It can be concluded a complete conversion between UML and implementation language cannot be achieved because there is no one-to-one mapping between the UML and the implementation language. And as I stated in Section 4.3., the lack of standard mapping between UML and the implementation language forces each CASE tool to define its own proprietary mapping.

Round-trip engineering is actually only possible in the case of class diagrams. The mapping between the class diagrams and the source code is quite straightforward, if complex associations are not used. In the case of more complex associations, the mapping between UML models and source code isn't unambiguous.

Round-trip engineering isn't possible for UML models that are based on run-time information. For example, there is no correlation between an interaction diagram and code other than the one inside the designer's head. Because of polymorphism, not all method invocations can be resolved at compile time. Other UML diagrams not mentioned in this context (use case, component and deployment diagrams) cannot be used in round-trip engineering either. This is one of the major problems in round-trip engineering. Basing the model solely on static information eliminates some interesting facts about a software system. It is possible to instrument the source code and derive dynamic UML models by running the program. However, this approach contains a lot of problems and has not been used commonly.

Booch *et al.* (1998) think that round-trip engineering, at least to some extent, can be used with other UML diagrams besides class diagrams. However, they cover this only briefly. They justify their approach with the fact that the mapping of UML to specific implementation languages for forward and reverse engineering is beyond the scope of UML. The only advice Booch *et al*. (1998) give for round-trip engineering is to use stereotypes and tagged values tuned to the programming language. I think that it's clear that in practice, round-trip engineering can only be used with class diagrams. The CASE tools only support round-trip engineering for class diagrams (except for some that support also sequence diagrams). I also think that UML should be more targeted towards implementation. As Booch *et al.* (1998) suggest, the primary product of a development team is software, not diagrams.

To be able to transform UML models into implementation language a lot rigor has to be invested in the models. This is not feasible in practice, it just moves the programming from the development environment into the CASE tool environment. CASE tools are meant for designing the applications, the actual programming should be done in a more suitable environment, for example in a modern Integrated Development Environment (IDE). In most cases, round-trip engineering is only suitable for generating a skeleton implementation and then keeping the UML models synchronized with the implementation. According to Booch *et al.*, (1998), things that are best expressed graphically are done so graphically in the UML, whereas thing that are best expressed textually are done so in the programming language.

5. Round-trip engineering in software process

5.1. General

All the software engineering activities, like round-trip engineering, are realized in a software process. A defined software process is needed to provide organizations with a consistent framework for performing their work and improving the way they do it. In this chapter I introduce how round-trip engineering is used in a state-of-the-art software engineering process, OMT++ (Aalti and Jaaksi, 1997). Another very popular software engineering process is RUP (Kruchten, 2000). OMT++ was selected as the process to study because it's almost an unofficial de-facto process for the development of interactive systems with user interfaces. This process is used as a reference throughout this thesis.

5.2. OMT++

5.2.1. General

OMT++ is a collection of practices, selected notation and phase-product templates that assist software developers in their everyday work. It covers software development from the requirements to the tested systems and provides means to monitor and manage this development. OMT++ uses the notation of UML, but only a subset of large UML is needed in OMT++.

The predecessor of OMT++, OMT+ (Kuusela and Aalto, 1993), is an object-oriented software development methodology developed by Nokia Research Center and Nokia Telecommunications, Cellular Systems (NCS). OMT+ is based on OMT (Rumbaugh *et al.*, 1991), Fusion (Coleman *et al.* ,1993) and OOSE (Jacobson *et al.*, 1992), and it provides a solid software process using the modeling facilities of OMT and Fusion. OMT++ is a superset of OMT+, extended with practices for the development of user interfaces, database management, and communication solutions. (Aalto and Jaaksi, 1994)

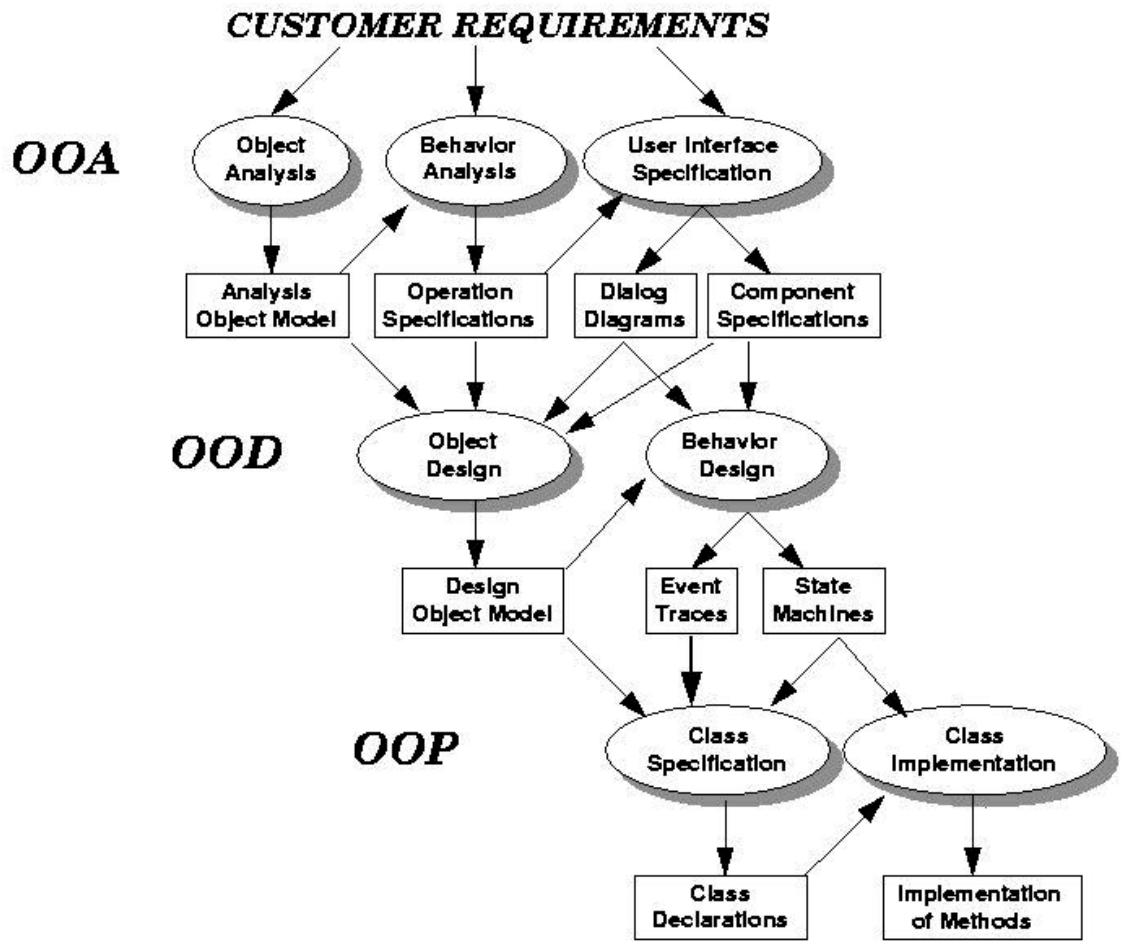

Figure 19. OMT++ process (Aalto and Jaaksi, 1994).

The OMT++ process transforms customer requirements to program code (see Figure 19). The process model specifies the artifacts, and includes guidelines for all phases from analysis through design to testing. Unlike most object-oriented development methods, OMT++ includes detailed procedures for the specification and design of user interfaces. Unnecessary diagrams, which do not add value to the customer, have been eliminated from the process. (Aalto and Jaaksi, 1994)

OMT++ consists of four main phases, object-oriented analysis, object-oriented design, object-oriented programming and testing. Each phase builds on the previous phases, and each sets requirements for the following phases. The phases are listed sequentially, but iterative approach is recommended. A clear distinction is made between analysis and design. Analysis deals with the concepts relevant to the end users. It analyzes the requirements and produces the initial solution descriptions. After successful analysis, both the users and the developers should have a similar vision about the future system. Design deals with the concepts relevant to the programmers. Design specifies how the

outlined and analyzed solution will be implemented. Design aims at programming, and the programming phase produces the software modules consisting of code. The programming phase makes the most detailed design decisions. Typically, testing the most elementary units of code, such as classes and functions, is considered to be programming. The testing phase integrates the modules together, builds the final software packages, and tests it against the requirements. (Jaaksi *et al.*, 1999)

OMT++ uses the notation of UML. In particular, it uses class, sequence and implementation diagrams, including component and deployment diagrams. Natural language is also an essential modeling tool in OMT++, especially when there is a need to communicate with end users or if there is a need to emphasize something related to diagrams. (Jaaksi *et al.*, 1999)

5.2.2. Round-trip engineering issues in OMT++

OMT++ doesn't utilize round-trip engineering. The transitions between phases in the process are done manually by the developer according to specified guidelines in OMT++. Some forward engineering practices are suggested (for example in GUI and database development), but object-oriented CASE tools are ignored. Jaaksi *et al.* (1999) think that it's more beneficial to draw design models with less details and not use forward engineering. They don't use reverse engineering either. They justify this with the fact that they do not require that design and analysis documents are kept up-to-date.

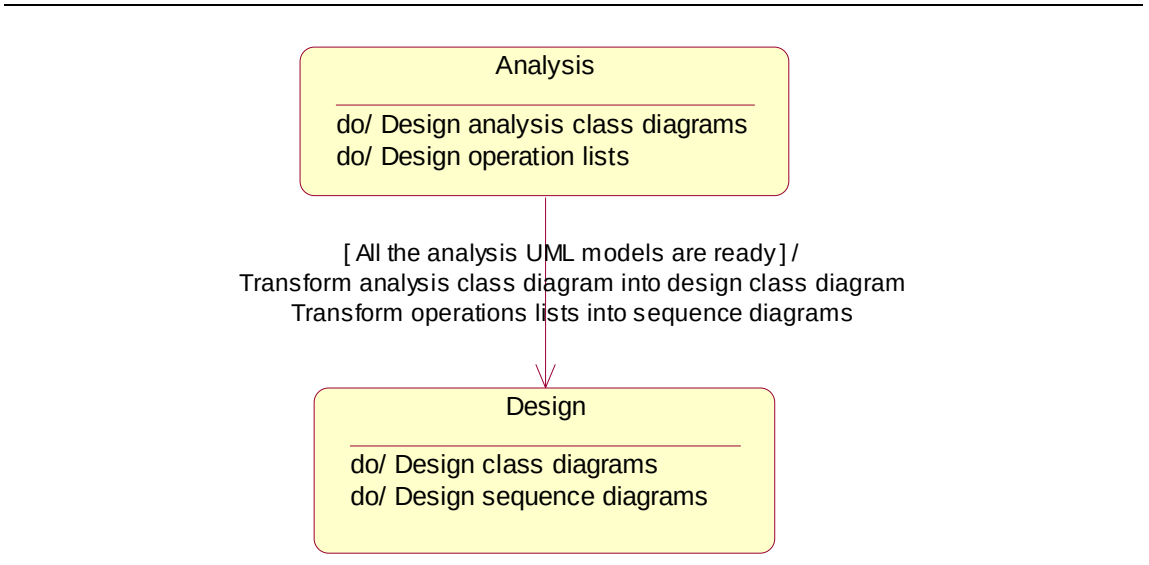

Figure 20. From analysis to design in OMT++.

When moving from the analysis phase to the design phase in OMT++, the design model is obtained directly from the analysis model using the guidelines of OMT++. The purpose of the design phase is to transform the products of the

analysis phase into a form that can be implemented in a programming language. The analysis class diagram is transformed into a design class diagram, and the operations in the operation list are modeled as sequence diagrams. (Jaaksi *et al.*, 1999)

When moving from the design phase to the implementation phase, the design class diagrams and sequence diagrams are used as a basis for programming. The implementation phase transforms the design class diagram and the sequence diagrams into a programming language. The class declarations are based on the design class diagrams, and the code of individual member functions is based on the sequence diagrams. This transformation is not trivial though. The graphical notation does not depict all the details of the programming language. Therefore, designers need to make important decisions on how to implement the software based on the graphical design specifications. It is important that classes, their interfaces, and the co-operation between objects have been modeled during design. Based on such a design, a programmer can concentrate on one class and one member function at a time. (Jaaksi *et al.* 1999)

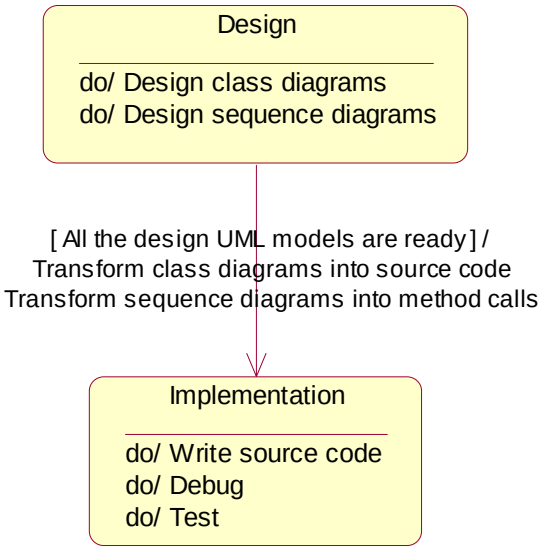

Figure 21. From design to programming in OMT++.

Jaaksi *et al.* (1999) think that in most cases there is no need to assist the implementation of a single class with any additional graphical notations. The programming language itself is the most powerful tool for this purpose. Figure 21 illustrates the transition from the design phase to the implementation phase.

OMT++ don't utilize forward engineering. Jaaksi *et al.*, (1999) think that GUI builders, database tools, and application wizards are very useful in limited portions of the system, but CASE tools with forward engineering are ignored

the process. In OMT++ design is done at a higher level of abstraction. This means that the UML models represent only the most important classes, methods and attributes. The most detailed design is performed during programming. Thus, the details of design can only be seen from code and the code itself is a document.

OMT++ doesn't utilize reverse engineering. Jaaksi *et al.*, (1999) do not think that it's beneficial to maintain analysis documents after a software project. Thus, reverse engineering is not necessarily needed. In OMT++, analysis documents are stored just as they are, and they are not maintained in future projects. Jaaksi *et al.*, (1999) suggest that future projects that build new features will only study the stored documents and will produce new analysis documents from scratch. Design documents are not kept up-to-date either. OMT++ emphasizes the approach that the ultimate purpose of a software development project is not to produce documents but to produce quality software.

5.3. Discussion

Jaaksi *et al.* (1999) suggest that the OMT++ process transforms customer requirements to program code. They also think that OMT++ provides a seamless transition from analysis to implementation. However, they specify the techniques of doing this only vaguely.

In OMT++ the design is done at a higher level with less detail. The design documents and UML models are not kept up-to-date when the programming has started. This kind of approach doesn't necessarily require round-trip engineering. However, I think that several aspects of OMT++ could be improved by utilizing round-trip engineering. For example, the transition from design to implementation could be accelerated a lot with forward engineering (see Section 4.3. for more advantages). With reverse engineering the documentation could be kept up-to-date with no manual effort (see Section 4.3. for more advantages). Good practitioners of the art consider that documentation is as essential to the application as the code itself. If the documents are not kept up-to-date, they pretty soon become useless.

Aalto and Jaaksi (1994) suggest that the object-oriented CASE tool technology is not mature enough for round-trip engineering. However, their aim is to convert the OMT++ design rules to a format understood by the computer as the technology matures. Any such development in OMT++ is yet to be discovered.

Keeping documentation up-to-date is especially important in iterative development where the software is prototyped, tested, measured, and analyzed, and then refined in subsequent iterations. OMT++ promotes iterative

development, but it doesn’t require that documents are kept up-to-date. I think this presents an inconsistency in OMT++.

Some people believe that whatever is modeled and designed is always right, and that nothing significant could happen during the implementation phase. This kind of approach promotes the traditional waterfall process model. But usually nothing is ever right the first time. What we build is constantly evolving, and requirements are changing all the time. Round-trip engineering supports the way that developers really work; do some design, then coding, testing, debugging and then some more design. Seamless integration between design and implementation is especially important in iterative development. The developer can go back to the design phase at any stage, make modifications to the UML models, and proceed with the implementation. The iterative process model is more feasible and more commonly used in real life software projects than the traditional waterfall process model. According to TogetherSoft (2000), round-trip engineering is very effective in an iterative process because the actual source code is generated while the class diagramming is done. This way it is easy to adopt a style that has software being built up, compiled, executed, and tested "continuously" as part of the normal development process.

One very important aspect in round-trip engineering is *when* the UML models and source code are synchronized. Round-trip engineering can be simultaneous or batch-mode. In simultaneous round-trip engineering the models and source code are always in-sync. This means that any changes made to the design are reflected in the source code immediately and, in turn, any changes made to the source code are immediately reflected in the design. In batch-mode round-trip engineering the synchronization of models and code is deferred to the end of the iterative cycle.

The greatest advantage of simultaneous round-trip engineering is that the model is always up-to-date with the as-built code. There is very little chance of the code getting out of sync with the model. The source code becomes just another view of the model, instead of being a model itself. Another major advantage of this approach is that it does not require the use of any separate reverse engineering tool. Although advances in automated reverse engineering have been significant over the years, there is still a lot of information that these tools miss, or confuse. Additionally these tools require a significant degree of skill to operate. Even after modified code has been analyzed there is usually a significant amount of manual massaging needed. (Conallen, 1997)

The major advantage of batch-mode round-trip engineering is that it lets developers be developers, allowing them to concentrate on getting code to work, and not on the subtleties of higher level object modeling. Intermediate

states of the code do not have to be incorporated into the model. It is easier for a developer to experiment with a technique before deciding to adopt it. The greatest disadvantages about deferring the changes to the model are mostly in the limitations of the tools. Careful review of all reverse engineered code must be made before incorporation into the main model. (Conallen, 1997)

When should one approach be adopted over the other? The answer to this question depends very much on the make up and experience of the analysts and developers involved, the desired degree to which the model matches the actual code, and the organizational structure of the team. The first approach requires developers with skill in object modeling, as they must select the right level of abstraction to present in the model. It requires a strong self-discipline and dedication to the integrity of the model. In those organizations where analysts and developers share hats, this approach makes the best match. When the iterative loop is small and quick turnaround is important, it is wiser to use the first approach, which updates the model as development progresses. When deliverables are expected every week, a separate reverse engineering stage might take too long. For larger organizations, with separate analysis and development teams the projects tend to be larger, and hence the length of the iterative loops longer. Developers are specialized in programming and analysts in modeling. A separate reverse engineering stage is more appropriate, during which the developers and analysts take part in the activity together. It is important that both skill sets are involved in the process, since the intentions made in code might be confused by non-programmers, and the importance of certain modeling constructs might be lost by one specializing in the code. (Conallen, 1997)

6. Round-trip engineering methodologies

6.1. General

In this chapter I introduce methodologies that support round-trip engineering. These methodologies improve the mapping between UML and the implementation languages, and promote better interoperability between tools involved in software development.

UML is not considered sufficient for a round-trip engineering language. As I stated earlier, one-to-one mapping between the UML and the implementation language isn't possible. One possible solution to this problem is to extend UML in such a way that it is 100% consistent with the implementation language. There are two methodologies intended for this purpose: J-UML (Kaitanen, 1999) and *The Universal Object Language*, UOL for short (UOL 1.2, 1998). J-UML specifies an implementation oriented UML extension for modeling Java implementations. An implementation-oriented model is constructed between UML and the implementation language in J-UML. UOL is a formal method. It enables round-trip engineering by defining the UML models in formal textual constraints.

Apart from methodologies, tool interoperability needs to be improved, otherwise the various tools in round-trip engineering cannot fully work together. XML is the de-facto data format in the industry at the moment. There are already XML based data formats for UML, namely *UML eXchange Format*, uXF for short (Suzuki and Yamamoto, 1998), and *XML Metadata Interchange*, XMI for short (XMI, 1998). XMI is considered better of these because it promotes a repository approach to UML modeling and it is also better supported by the industry. The basic idea in the XMI approach is that all the tools involved in the round-trip engineering act as front ends to the shared XMI repository. When all the software development information is stored in this repository, design and implementation are always up-to-date and consistent.

6.2. J-UML

6.2.1. General

Currently there is no common modeling language describing simply and un-ambiguously pure source code, for example the Java™ programming language code. Thus, there exists a clear gap between design and the actual programming. Some quite simple and common two-dimensional view for describing the Java source code - i.e. similar to existing modeling environments - would be very useful for Java developers. This could also encourage the actual

developers into modeling and design. There is a clear need to describe the actual Java code – final decisions in the implementation – unambiguously. (Kaitanen, 1999)

J-UML is a graphical UML extension for Java. It is developed by Kari Kaitanen at the Technical Research of Finland. According to Kaitanen (1999), J-UML enables straightforward transformation between design and implementation for UML and Java by defining implementation oriented concepts to UML. J-UML can be understood as a subset of UML with Java implementation concepts. It's totally Java oriented supporting only and fully the graphical object-oriented design of the actual Java source code.

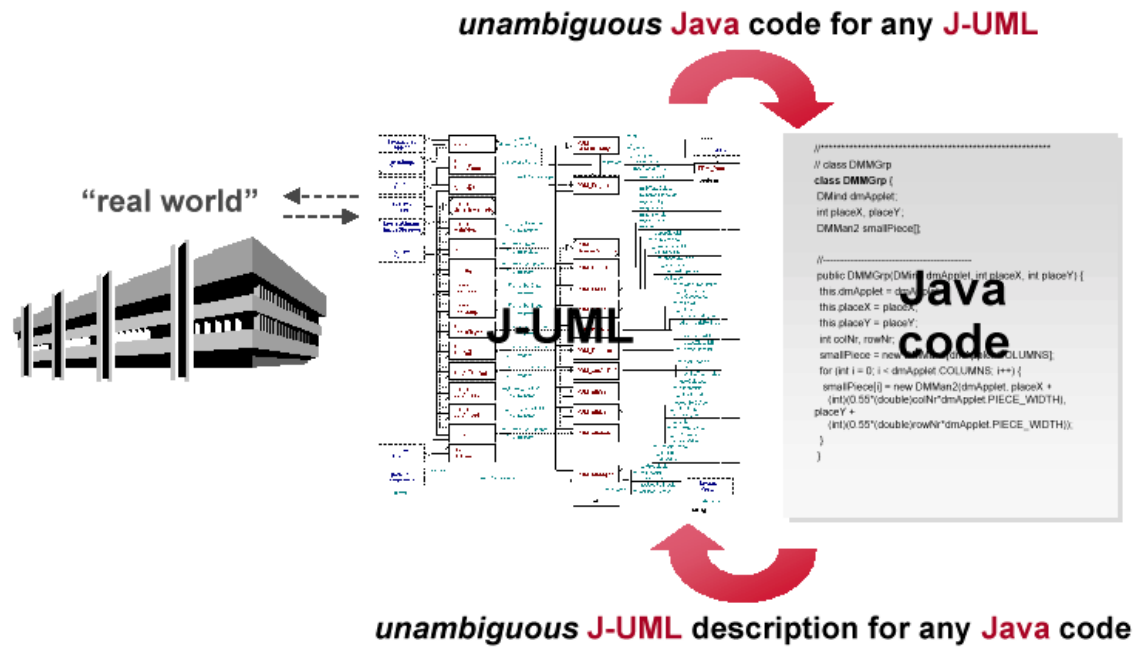

Figure 22. J-UML (Kaitanen, 1999).

J-UML is not trying to diminish the important value of language independent modeling. Object-oriented modeling languages, like UML, are language independent, which is excellent. In a nutshell, modeling environments are created for domain experts whereas programming languages are for software experts. Most of the problems of finding the best object-oriented solution for any application field can be found by using a well thought OOA / OOD methodology. But eventually the created model has to be mapped into some implementation language. When the developer gets closer to the actual implementation, the level of details becomes smaller. Besides the constraints of the designed data model, the available system with chosen language environment starts to make various restrictions to the model. Also, this new implementation has to be integrated with the existing environment, for example to component libraries etc. that might have been produced earlier.

J-UML is used as an adapter between UML and Java, it defines how to map UML models into actual Java implementations. The implementation model has to meet all the requirements of the system: resources, speed, limitations of memory, security issues of network and for example all the existing libraries or components you might reuse in the project. (Kaitanen, 1999)

According to Kaitanen (1999), J-UML is a subset of UML. In addition to UML, J-UML contains the following concepts:

- direct references to Java resources; Java APIs,
- `interface` entity (better reflection of Java syntax),
- `implements` relationship (better reflection of Java syntax) ;
- Java (primitive) data types,
- supporting also "wrapped" Java primitive types,
- definition of the actual reference ("pointer") location in associations (i.e. defining the class having the association),
- two-way, "strong", relationships (supporting navigation,
- optional relationships (supporting navigation),
- Java arrays,
- full support to Java dynamic data structures (`java.util.Vector`, `java.util.Hashtable`, etc.),
- page references (for unambiguous AP / schema definitions),
- full support to Java component interface; JavaBeans, and
- package coloring.

Having the implementation concepts included in the UML, it is possible to transform J-UML to the implementation language and vice versa, with no loss of information. Kaitanen (1999) states the idea J-UML in a nutshell:

"*You can't design anything that can't be straightforwardly transferred to Java. Also, you can't make any Java code that doesn't have an unambiguous J-UML counterpart.*"

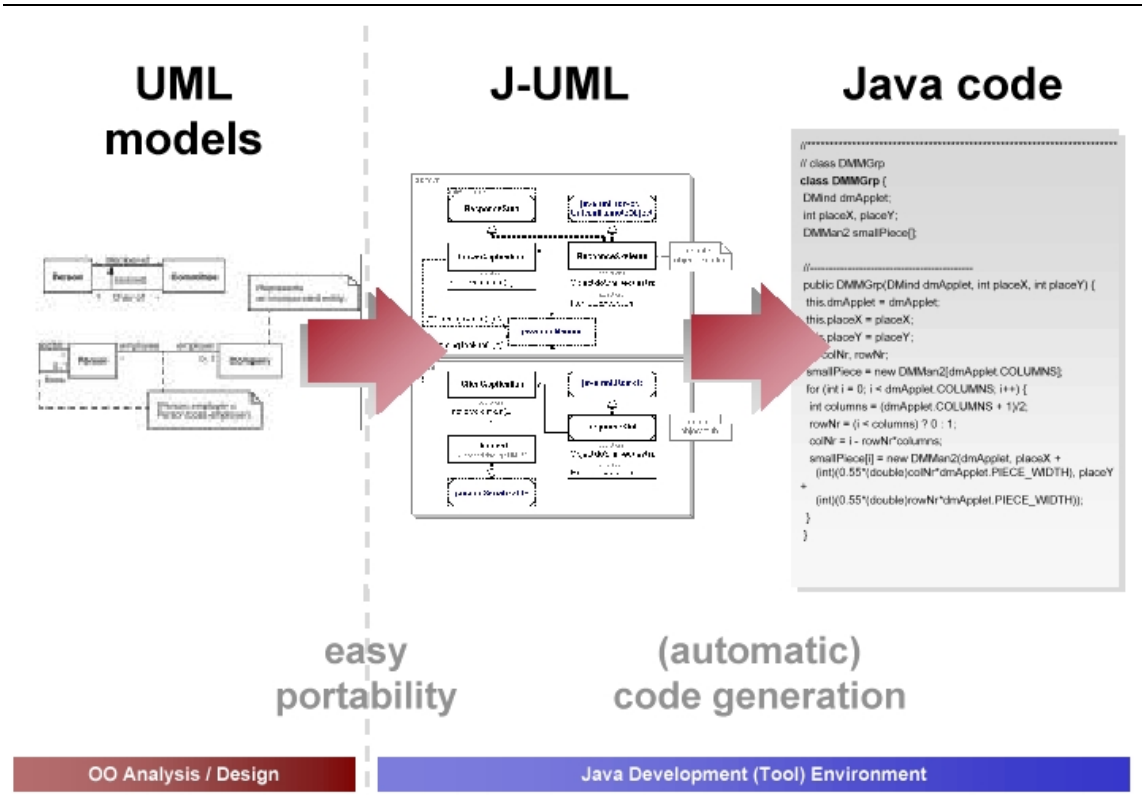

Figure 23. Transforming UML models into Java applications (Kaitanen, 1999).

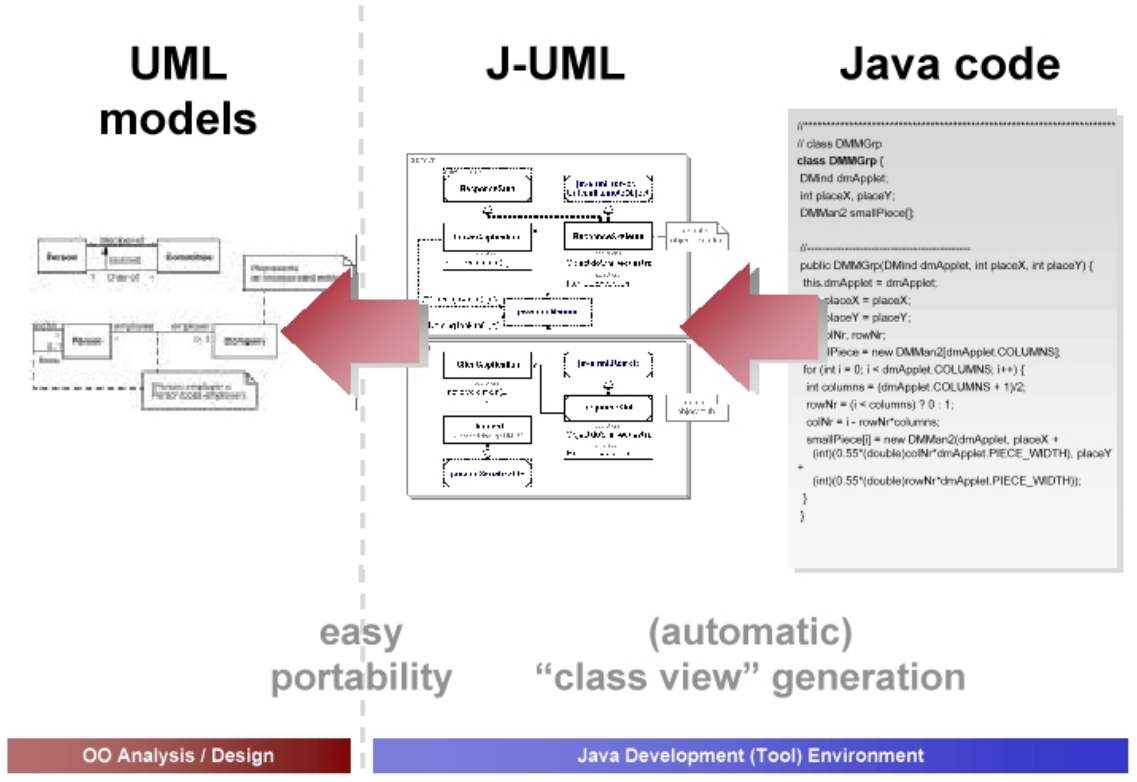

Figure 24. Describing the implemented Java code (Kaitanen, 1999).

6.2.2. Example

In Figure 25 there is one simple UML structure - of only three classes –and some (7) variations in Java implementation for this UML definition. According to Kaitanen (1999), the (senior) programmers usually make these final decisions about the implementation. In many cases it's handy to be able to easily describe these actual application structures for the other developers as well - for the team or for project management.

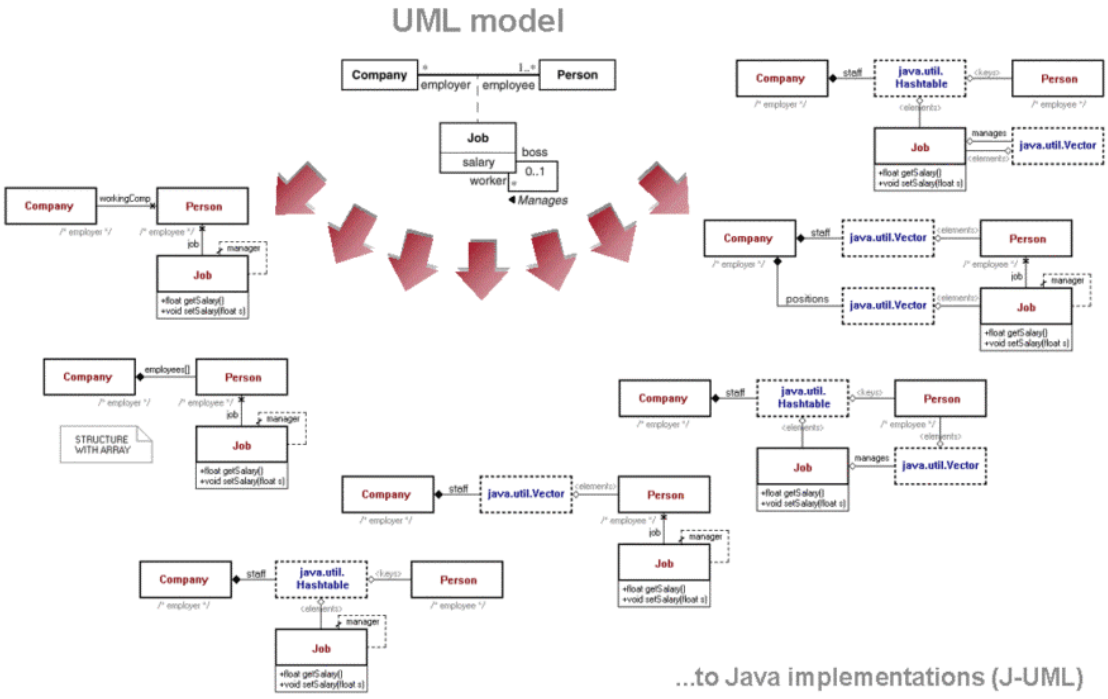

Figure 25. Implementation variations from UML (Kaitanen, 1999).

Figure 26 illustrates one possible Java implementation for the UML model in Figure 25.

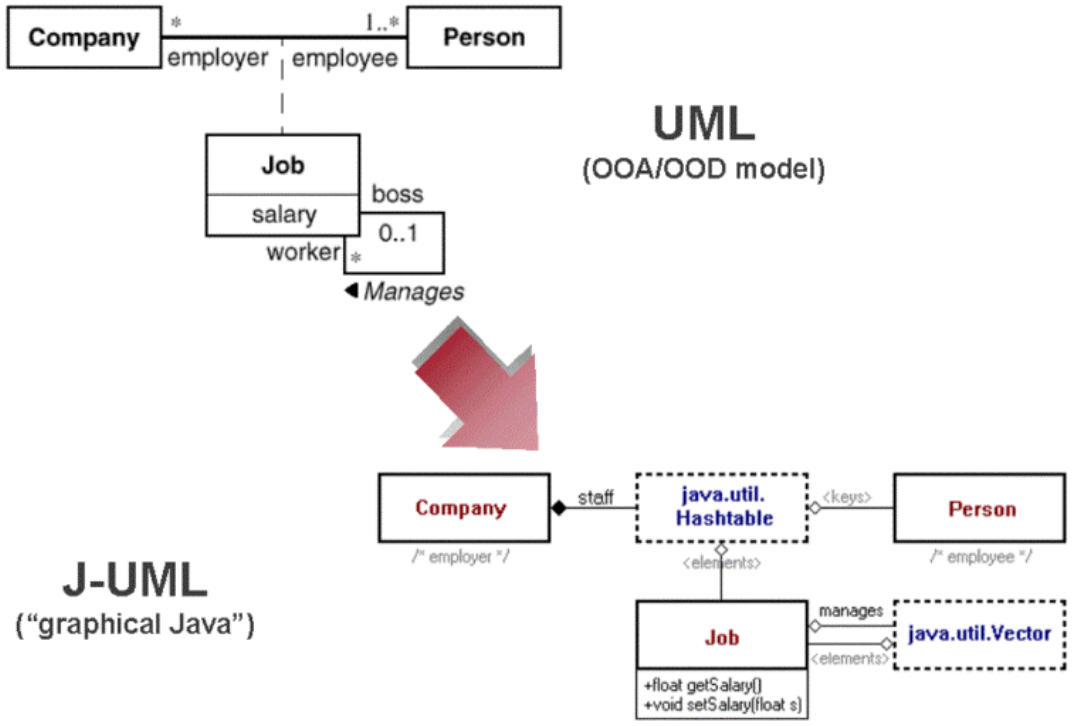

Figure 26. UML to Java (J-UML) conversion (Kaitanen, 1999).

6.3. Universal Object Language (UOL)

6.3.1. General

Object-oriented modeling languages are richer than OO programming languages. They are capable of describing more concepts and in more detail than any OO programming language. Transforming an OOAD model to code implies, therefore, a loss of information in all cases. The only way to solve these problems is with a language (a "round-trip engineering language") that can be used to enrich the source code in any language. This means that the language is able to describe all OOAD constructs to facilitate round-trip engineering and it's also sufficiently simple that any programmer can learn in a few days. (UOL 1.2, 1998)

The Universal Object Language (UOL) is a textual, object-oriented full life-cycle language developed by Recerca Informàtica, Universitat Politecnica de Catalunya and Daimler-Benz Research and Technology. A full life-cycle language can be defined as *a language that allows us to use and communicate always with the same OO constructs at any stage and with any tool or between any pair of tools.* UOL is intended to support a wide range of usage patterns and applications. One of these applications is round-trip engineering. UOL is an object-oriented formal language that can be embedded into any OO

programming language. With UOL the developer can use the same object-oriented concepts throughout the whole life cycle of software development. This enables seamless transition between the stages in software development and thus facilitates round-trip engineering. (UOL 1.2, 1998)

Round-trip engineering can be considered as tool-to-tool communication: it is communication between design tools and programming tools. In forward engineering the programming tool uses input (UML models) from a design tool. In reverse engineering the design tool uses input (source code) from a programming tool. In order to obtain seamless transition between the stages in software development, it is not only necessary to use the same concepts at every stage, but also to use at every stage tools that support the same concepts. The problem arises because at some stages we are forced to use, to communicate with, tools that do not support OO concepts. (UOL 1.2, 1998)

UOL 1.2 (1998) justifies UOL as a round-trip engineering language with the following facts:

- **Standard language for OO CASE tools.** CASE tool builders can use it as a substitute for their proprietary incomplete (it is not well adapted to all OO languages) “mark-up language” that they are presently using. They usually have different versions of the mark-up code for different languages.
- **Standard language for round-trip engineering tools.** Using only one mark-up language for all programming languages reduces over 80% of the cost of developing round-trip tools, if round-trip engineering tools are split in two parsers: a front-end (common to all source languages) and a back-end (specific for each language) using, what we call, collaborative compilation.
- **Standard language for component developers.** It may, also, be used by companies developing GUI builders, component libraries, etc. to allow the source code, and its corresponding OOAD model embedded in the code, generated by their products to be easily imported into any model by CASE tools supporting UML.
- **Standard round-trip engineering language for developers.** Due to the simplicity of the language and the easiness of learning it, the programmer can easily change the code and the model during testing and debugging. This avoids necessary guessing by the round-trip tool.
- **Reverse engineering of non-CASE-tool-generated OO code.** It allows reverse engineering of non-CASE-tool-generated OO code to be imported correctly if it is enriched with UOL code. This can be done manually or with a software product that acts as a Wizard, that can learn

from previous experience asking the programmer questions like "Is this declaration an attribute or does it represent an association?" "Is this pointer a shared aggregation or simply an association?" etc.

- **Better interoperability between tools.** Makes interoperability, both for the model and the code, between CASE tools immediate and with more semantics than interchange formats allowing for more control during transfer between tools.

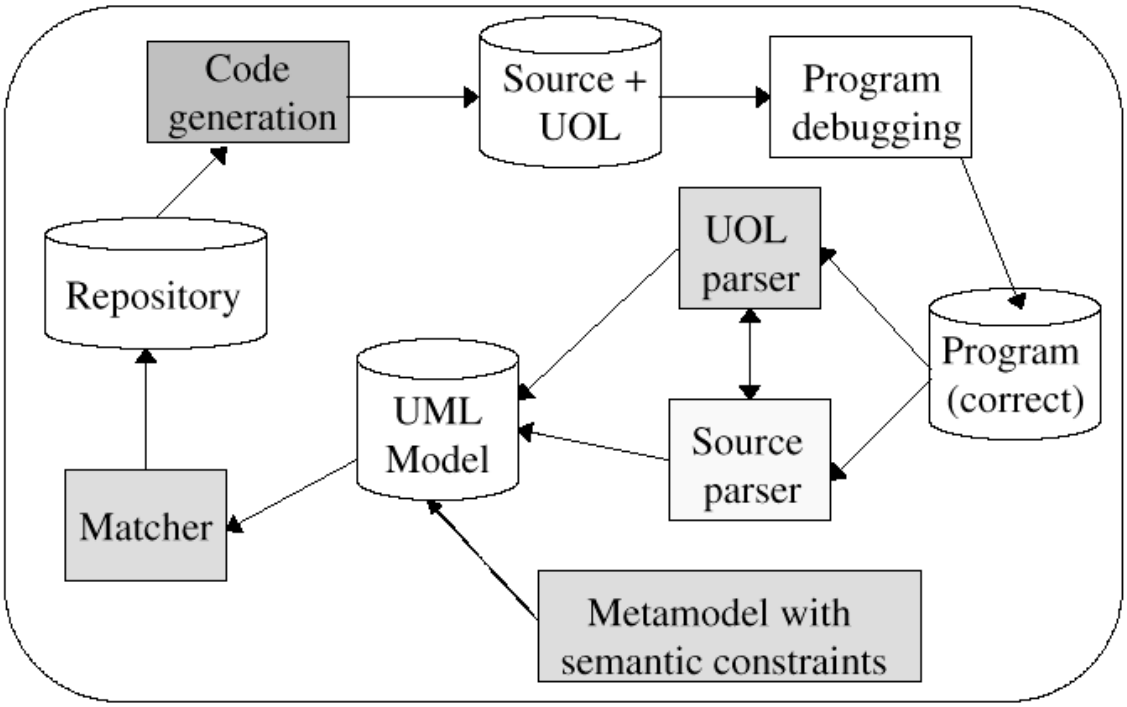

Figure 27. The modular structure of a RTE tool (UOL 1.2, 1998).

6.3.2. Example

The examples here illustrate the syntax and usage of UOL.

```
Class_declaration     -> Class_header ( Formal_generics )?
                      ( viewed with View_element_name_list )?
                      Extension_declaration_list
                      Extension_use
                      Class_body end
Class_body            -> Inheritance Rest ( Use_of_constraint )?
                      | Features State_machine ( Use_of_constraint )?
                      | Features feature '{' Visibility '}'
                      Feature_rest end ( Use_of_constraint )?
Rest                  -> Features ( State_machine )?
```

```
Class_header          -> ( deferred )? class Class_name
Class_name            -> identifier
```

Figure 28. UOL class syntax expressed in Extended BNF grammar.

```
class Person viewed with MainD
     feature {any}
          isMarried, isUnemployed:Boolean;
          birthDate:Date;
          age:Integer;
          firstName,lastName:String;
          sex: unique { male,female };
          deferred income(d:Date):Integer is text"Incoming operation"
     end
     -- State machine for the class Person
     -- declaration of an invariant
     constrained by
     { self.age>=0 }
     -- rest of constraints ommited
end -- Class Person
```

Figure 29. Class Person defined with UOL.

6.4. XML Metadata Interchange (XMI)

6.4.1. General

The XMI specification integrates XML, UML and MOF specifications adopted by OMG. *The Extended Markup Language*, XML for short (XML 1.0, 1998), is a standard for defining, validating and sharing document formats on the Web. It is a specification set forth by the World Wide Web Consortium (W3C). *The Meta Object Facility*, MOF for short (MOF 1.3, 1999), is designed as an OMG repository standard for distributed repositories and meta data management and is fully integrated with the UML and XML. XMI was originally sponsored by IBM and Unisys as a result of their joint work on the OMG UML and MOF standards.

Brodsky (1999) suggests that application development tools can interchange their information using XMI as the standard. These tools include:

- **Design tools**, including example object-oriented UML tools such as Rational Rose and Select Enterprise.
- **Development tools**, including Integrated Development Environments like VisualAge for Java and Symantec Café.

- **Databases**, Data Warehouses and Business Intelligence tools, including IBM DB/2, Visual Warehouse, Intelligent Miner for Data, and Oracle/8i.
- **Software assets**, including program source code (C, C++, and Java) and CASE tools such as TakeFive's SniFF+.
- **Repositories**, including IBM VisualAge TeamConnection and Unisys Universal Repository.
- **Other tools**, for example report generation tools, documentation tools, and web browsers.

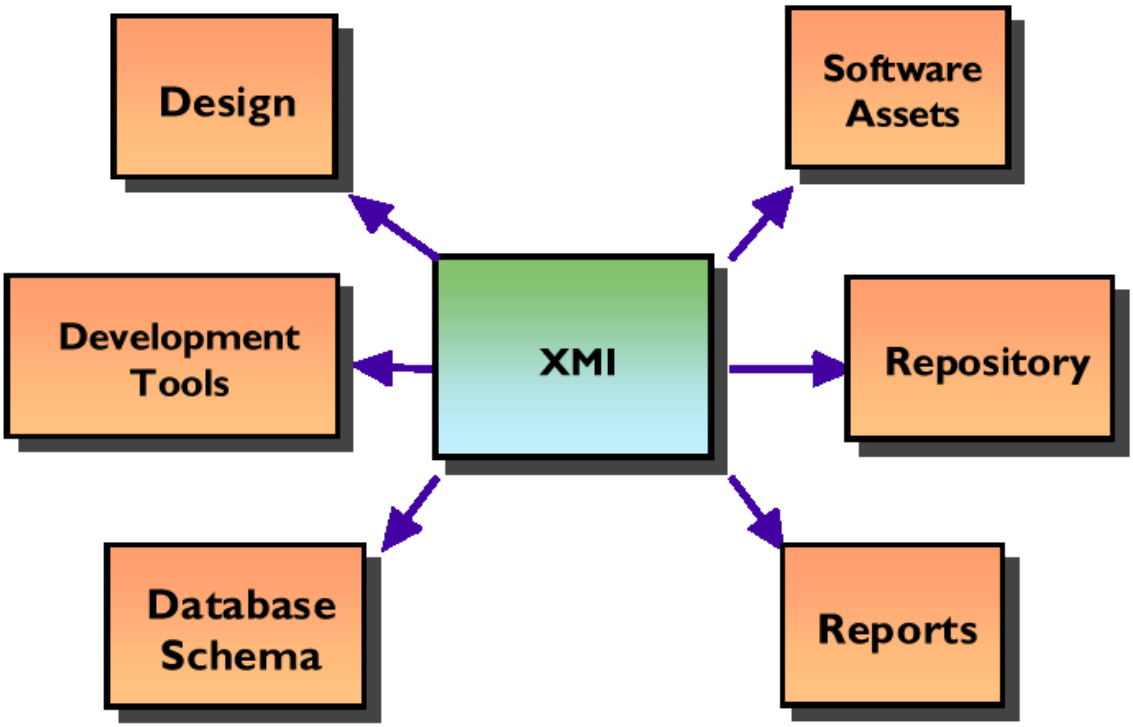

Figure 30. Open interchange with XMI. Brodsky (1999)

In order to participate in this architecture, each vendor only needs to add XMI support to their product to leverage access to all the other tools. The system is open and everyone can participate immediately with XMI-enablement. Participants also gain a web-enabled collaborative environment by the close relationship between XMI and XML, a standard technology of the Internet being added to the major web browsers. (Brodsky, 1999)

6.4.2. Example

XMI defines two sets of rules that provide open interchange and leverage the capabilities of XML. The two sets of rules in XMI are DTD generation and document generation. The DTD generation is used to specify an interchange format, and the document generation creates documents that use a given XMI DTD. (Brodsky, 1999)

Suppose that one constructs a model for a car. Figure 31 shows the car modeled with UML. The car is a UML class containing several class-attributes: `make`, `model`, `year`, `color`, and `price`. Figure 32 shows an example of application exchange using this DTD. The applications are exchanging an XMI document containing a specific car definition. The document uses the elements from the generated XMI DTD generated from the UML car model. The last example shows the car example as an XMI document using elements from the generated car XMI DTD.

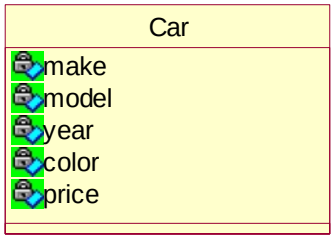

Figure 31. Car as a class in UML.

```
<!ELEMENT Car (Car.make, Car.model, Car.year,
Car.color, Car.price,
XMI.extension*)? >
<!ATTLIST Car %XMI.element.att; %XMI.link.att;>
<!ELEMENT Car.make (#PCDATA | XMI.reference)* >
<!ELEMENT Car.model (#PCDATA | XMI.reference)* >
<!ELEMENT Car.year (#PCDATA | XMI.reference)* >
<!ELEMENT Car.color (#PCDATA | XMI.reference)* >
<!ELEMENT Car.price (#PCDATA | XMI.reference)* >
```

Figure 32. Car as an XMI DTD.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XMI SYSTEM "car.dtd">
<XMI xmi.version="1.0" >
     <XMI.header>
          <XMI.documentation>
               An example of a car.
          </XMI.documentation>
     </XMI.header>
     <XMI.content>
          <Car>
               <Car.make>Ford</Car.make>
```

```
            <Car.model>Mustang</Car.model>
            <Car.year>99</Car.year>
            <Car.color>blue</Car.color>
            <Car.price>25000</Car.price>
        </Car>
    </XMI.content>
</XMI>
```

Figure 33. Car as an XMI Document using the Car XMI DTD.

6.5. Discussion

The methodologies presented here can also classified as formal methods, at least in the case of XMI and UOL. Formal methods haven't yet proven successful is software development because software developers consider them too complicated to use in practice. XMI and UOL also promote the repository approach as a best way to implement round-trip engineering. In this approach, UML models are stored is a shared repository that is used by different tools.

Marciniak (1994) defines formal methods as mathematically based techniques for describing system properties. They typically include a precise notation for constructing mathematical models of a (software) system. The notation is used for specifying (rather than designing or implementing) the required system, and is concerned with what is done by a system rather than how it is done. Two defining characteristics of formal methods are precision and abstraction. According to this definition, UOL and XMI can be classified as formal methods. With XMI, however, the developer doesn't write any XMI code. The tools transform UML models into XMI (and vice versa).

Long experience tells us that normal software developers will not, in the foreseeable future, be willing to use abstract formal languages and notations to design software, regardless of how theoretically desirable it might be to do so. Software engineers are not trained in using the discrete mathematics that formal specification is based on. Also, relatively little effort has been devoted to tool support yet such support is essential if large-scale specifications are to be developed. The only way to get end users to use formal techniques is to provide tools that make it very simple and obvious to do so. (Evans *et al.*, 1999)

Given the factors, I don't think that formal methods like UOL could prove successful in large-scale software development. UOL is too complicated to use because of its formalism. On the other hand, formalism is required to express UML models unambiguously. J-UML is a graphical UML extension, yet defined to unambiguously model Java source code. I think this is a much better approach for round-trip engineering. J-UML is easy to use and it is also formal

enough formal enough. With J-UML, the modeling is divided into two sections: logical modeling and implementation modeling. The implementation modeling is where round-trip engineering is used. It is possible to transform J-UML models to implementation language and vice versa, with no loss of information, because the implementation concepts of Java are included in the J-UML.

XMI and UOL promote the repository approach to round-trip engineering. TogetherSoft (2000) criticizes the repository approach with the fact that using a separate repository (binary or otherwise), separates the source from the model. However, I agree with Evans *et al.* (1998) in that the best way to implement round-trip engineering is to use the repository approach. UML models are stored in a shared repository that is used by different tools, for example modeling tools, development tools, and metrics tools. Instead of using proprietary formats, all the repository vendors use a common format for storing the UML modeling artifact. XMI is clearly the best choice for the repository format because it is based on XML that is the de-facto data format in the industry at the moment. With XML the syntax and semantics of the repository format can easily be modified and extended. The designs can be expressed in www-pages, which is important in today's networked world. Thus, XMI provides true interoperability between tools.

7. Round-trip engineering tools

7.1. General

In this chapter I present two round-trip engineering CASE tools for the Java™ programming language: Rational Rose from Rational Software and Together from TogetherSoft. The specific versions covered here are Rational Rose 98i Enterprise Edition with Rose J add-in and Together/J 3.2. In this thesis, CASE tools are covered only from practical point of view with the purpose to introduce the support for round-trip engineering.

CASE tools offer many benefits for developers building large-scale systems. As spiraling user requirements continue to drive system complexity to new levels, the CASE tools enable us to abstract away from the entanglement of source code, to a level where architecture and design become more apparent and easier to understand and modify. The larger a project, the more important it is to use CASE technology. As developers interact with portions of a system designed by their colleagues, they must quickly seek a subset of classes and methods and assimilate an understanding of how to interface with them. In a similar sense, the management must be able, in a timely fashion and from a high level, to look at a representation of a design and understand what's going on. For these reasons, CASE tools coupled with methodologies give us a way of representing systems too complex to comprehend in their underlying source code or schema-based form. (Hebbel, 1997)

Object-oriented CASE tools create diagrams that represent an object model using the notational elements of specific methodologies. These notations depict classes (including attributes, methods, and events), various object relationships (such as inheritance, aggregation, and friendship), and cardinality. In addition, most if not all of the tools available offer the ability to generate skeletal application code from models. This skeletal code typically includes class definitions and function prototypes. Developers then add meaningful underlying application logic using C++, Java, or Smalltalk, to name just a few supported languages. (Hebbel, 1997)

The object-oriented CASE tool technology is finally maturing. Earlier the CASE tools were only specialized drawing tools. At the moment, the most advanced CASE tools feature round-trip engineering in team environment. Currently, there are many CASE tools that feature forward, reverse or even full round-trip engineering. UOL 1.2 (1999) defines the following complementary functionality for CASE tools in order to fulfil the obvious need of facilitating synchronicity between model and code:

- generate source code,
- export source code changes from a modified model,
- import source code generated from other tools and the model it represents,
- import modified source code reconstructing the model.

7.2. Rational Rose

Rational Rose dominates the market of UML case tools. It provides a wide range of features for visual modeling, component-based development and round-trip engineering.

The major features in Rational Rose are the following (Rational, 2000):

- *Controlled iterative development results in shorter development cycles.*
- *Model-driven development results in increased developer productivity.*
- *Use-case and business-focused development results in improved software quality.*
- *Focus on software architecture and components results in significant software reuse.*
- *Common, standard language results in improved team communication.*
- *Reverse engineering capabilities allow you to integrate with legacy systems.*

The round-trip engineering features of Rational Rose are as follows (Rational, 2000):

- *Rational Rose allows you to move easily from analysis to design to implementation and back to analysis again, and thus supports all phases of a project's lifecycle.*
- *Rational Rose supports a dynamic change-management process with forward engineering, reverse engineering, and model updating features that allow you to alter your implementation, assess your changes, and automatically incorporate them in your design.*
- *Rational Rose's support for round-trip engineering ensures that the iterative development cycle is controlled and productive.*

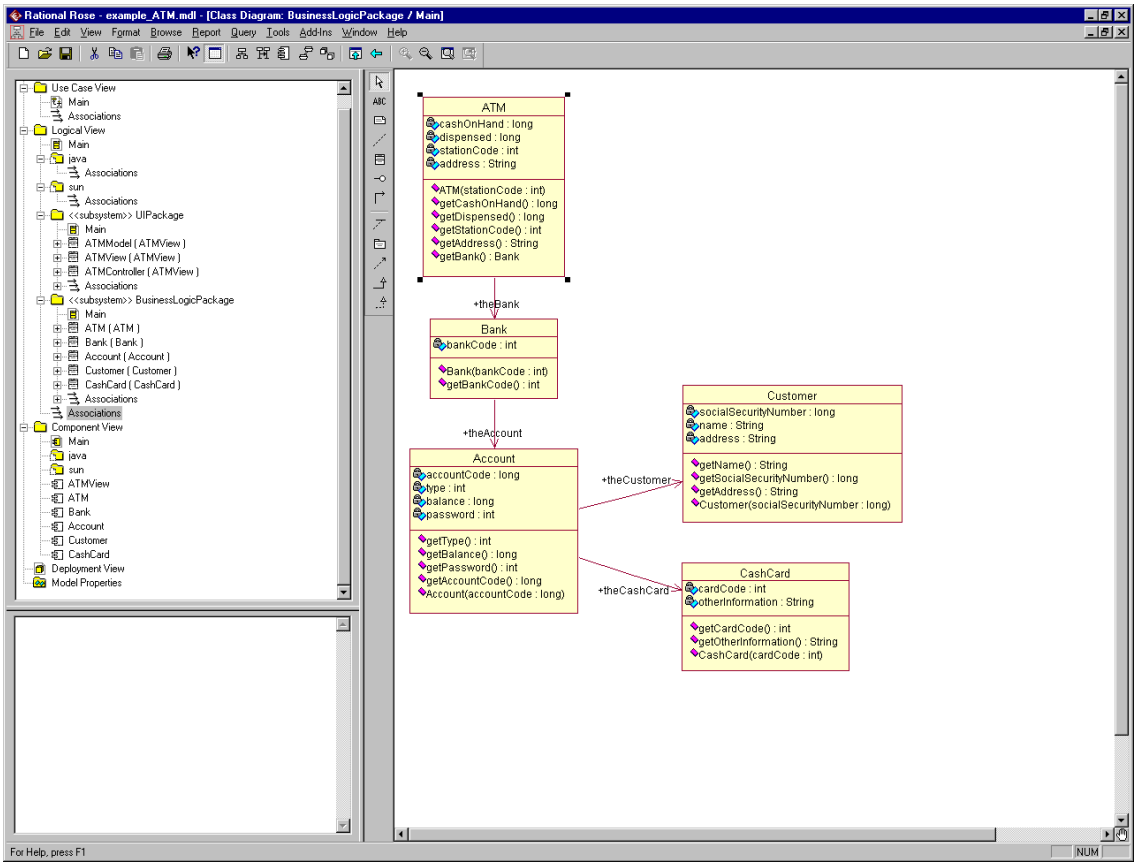

Figure 34. Rational Rose.

7.3. Together

Together is currently one of the most interesting design tools, especially for the Java™ programming language (Together/J). The unique feature in Together is the simultaneous round-trip engineering, which means that changes in the design are reflected in the source code *immediately* (and vice versa). Together also features end-to-end team support, multi-level documentation generation and much more.

The main features in Together are the following (TogetherSoft, 2000):

- *OneSource™ simultaneous round-trip engineering for Java.*
- *True multi-user team support built right in.*
- *Reusable integrated patterns and components.*
- *Customizable QA tracking for requirements, metrics and audits.*
- *Immediate integration with existing development tools like IDEs and version control systems.*
- *Extensive customizability to meet your standards and adjust to the way you work.*

The round-trip engineering features of Together can be described as follows (TogetherSoft, 2000):

"At the heart of Together is the concept of simultaneous round-trip engineering. It means that any changes made to the design are reflected in the source code immediately and, in turn, any changes made to the source code--in Together or an IDE [or your favorite editor]--are immediately reflected in the design."

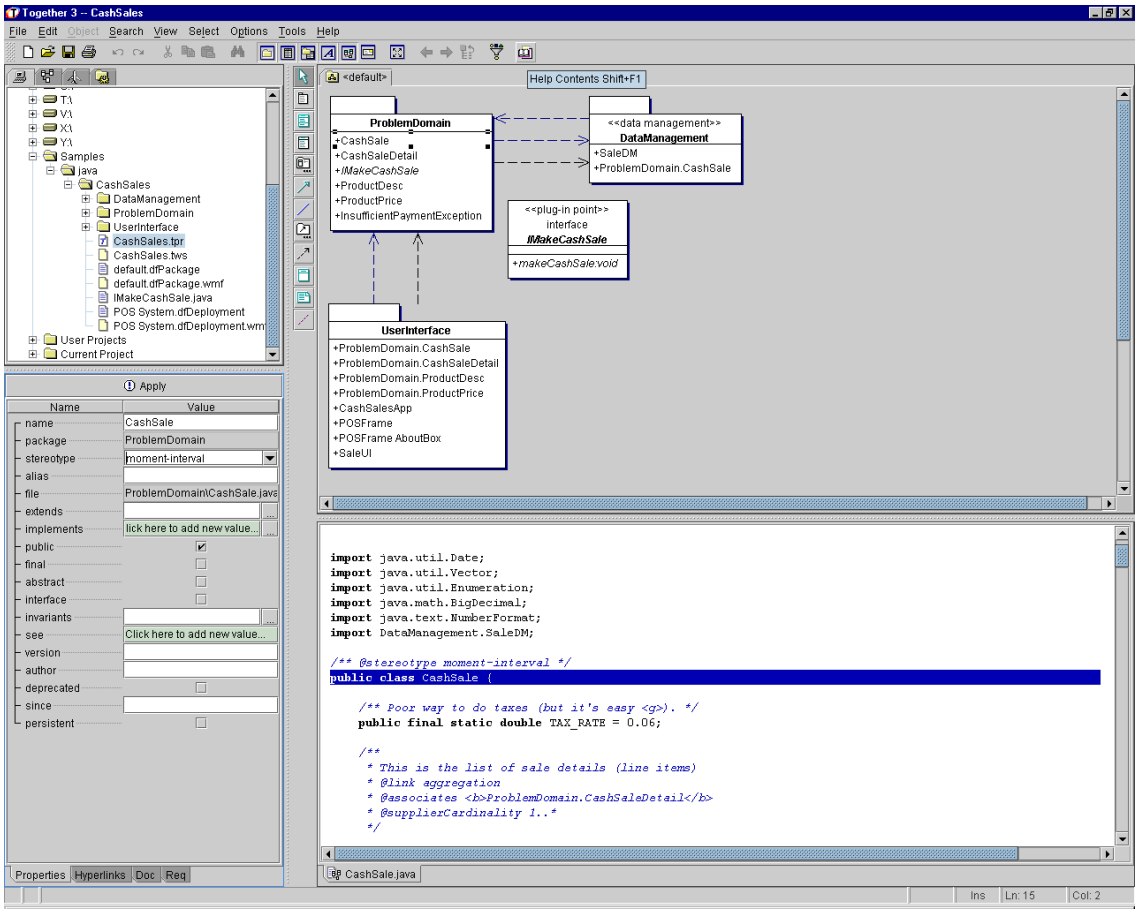

Figure 35. Simultaneous round-trip engineering in Together.

7.4. Comparison

The comparison of features relevant to round-trip engineering in Rational Rose and Together is illustrated in Table 1. TogetherSoft (2000) defines the following requirements for CASE tools' round-trip engineering features:

- Provide a full suite of UML diagramming capability
- Diagramming should support "changing one's mind," making it simple to try out different modeling constructs.
- Allow corporate- and project-specific standards to be imposed on source code generation.
- Provide complete customization from the top of the source code file to the very last line, and everything in between.

- During class diagramming, provide implementation details for attributes, operations, constructors, and associations—giving the developer a head start.
- The tool should allow invocation of other favorite tools (e.g., editor, compiler, IDE, lint checker, unit test).
- Tool should not add in large sections of auto-generated, tool-specific artifacts to the source code. This obscures the source code and is a penalty every time a developer has to go into the code to edit the source. (The penalty is in the effort required to "see through" the awkward and bulky code markers to try to glean the true source code-- separating the wheat from the chaff as it were.
- Tool should employ "natural" looking artifacts-- making it easier to decode the "meta language". No wading through non-native, proprietary code markers.
- Tool should use natural, text-based model artifacts to make it simpler to put these items under source control.

The comparison of features is presented in Table 1. In the table, EJB stands for *Enterprise Java Beans*, IDL for *Interface Definition Language*, DDL for *Data Definition Language*, JDBC *for Java Database Connectivity*, and DCOM for *Distributed Component Object Model*. CMVC, CVS, VSS, PVCS, and SCC are version-control systems.

Feature	**Rational Rose**	**Together**
UML Diagrams (v1.3)	x	x
Use Case	x	x
Class	x	x
Sequence	x	x
Collaboration	x	x
Activity	x	x
State	x	x
Business Process	-	x
Package	x	x
Component	x	x
Deployment	x	x
Data model	x	x
User-defined	-	Simplify or extend any UML diagram. Create new diagram types.
Round-trip engineering		

Class diagrams RTE	x	x
Sequence diagrams RTE	-	x Does not generate initial code.
Statechart diagrams RTE	-	-
Simultaneous RTE (always up-to-date)	Manual synchronization required.	x
Other	IDL generation for CORBA. DDL generation. Oracle 8 forward and reverse engineering for object models<> relational schemas.	Simultaneous RTE for class diagram <> EJB source code, class diagrams <> IDL code. RTE for data models <> JDBC. IDL generation for CORBA and DCOM. DDL generation for leading database management systems.
Document generation		
HTML	x	x
RTF	-	x
ASCII	-	-
Metrics	-	Metrics checking and likely-error audits.
Customizability		
customizable forward engineering	-	“Fill in the blank” source code construction templates.
customizable document generation	Some features, more customization with scripts.	Visual template designer. Custom reports in any format.
XMI support		
export XMI	x	x
import XMI	x	x
Java features		
Java Beans	Limited features.	RTE class diagrams / Java Bean source code.
Enterprise Java Beans	Limited features.	Simultaneous RTE for class diagrams <> EJB source code. View and edit EJB as one component.
Repository support	Integration with MS Repository.	-
Other		
Design pattern support (generation,	-	Gang-of-Four patterns, custom

extract from code)		patterns.
Team support	Rational Rose supports teams of analysts, architects, and engineers by enabling each to operate in a private workspace that contains an individual view of the entire model.	BigPlay™ multi-user team support
Version control integration	Integrates with all the major version control systems (Rational ClearCase, Microsoft SourceSafe, also open to other CMVC systems.	Integrates with all the major version control systems (CVS, VSS, PVCS etc., also any SCC-compatible version-control system
Integration with other tools	Good integration with other Rational tools. Many add-ins to other tools.	Integration with other IDEs and editors.
Support for different language (Java, C++)	x	x

Table 1. Comparison of Rational Rose and Together.

The major difference between the two tools is that Rational Rose supports only batch-mode round-trip engineering whereas Together supports only simultaneous round-trip engineering. In Together it is even possible to use separate environments for designing UML models and writing code. Still, simultaneous round-trip engineering is possible. UML models and source code are automatically synchronized when the developer switches between design and implementation environments. Rational Rose supports repositories, but Together uses only text-based artifacts. Together beats Rational Rose in that there is none of the burden of code markers or limited regions for editing the source code as used by Rational Rose.

7.5. Discussion

Round-trip engineering promotes better utilization of CASE tools. The basic CASE tools are only specialized drawing tools. But a truly useful tool does more than paint pretty pictures. According to TogetherSoft (2000), the more advanced CASE tools ensure that all UML diagram types are drawn "to spec" and that the details and behavior behind the diagram types should be highly automated so that the user need not be concerned with diagram notation. When doing object modeling, for example, knowing that actual source code (syntactically correct) is being generated is a comforting feeling. The most advanced CASE tools feature round-trip engineering in team environment. In round-trip engineering the design models are just as important artifacts as the source code. If the design models are not used in the real programming, the

developers will not use the CASE tool and the implementation will begin to deviate from the one specified in the analysis and design phases. According to my experiences, this is such a difficult issue that a large number of developers have simply given up modeling with CASE tools and use either a paper and pencil or a simple but versatile design tool such as VISIO to visualize and communicate their initial designs. According to Kaitanen (1999), UML has not been commonly used by many programmers yet. One reason for this is that there is not enough time to spend on modeling. Programmers also tend to think that the UML models are somewhat useless because there is no connection between design and implementation. There is a huge conversion, but also a mental based gap between the OOA/OOD environment and the actual programming. This gap could be considerably narrowed with forward engineering and round-trip engineering and thus, it could encourage the actual developer to modeling and design.

To support forward engineering there must be a way to ensure that the CASE tool doesn't overwrite method bodies (and other code or comments) entered by the developer. The safest and most frequent way of doing this is to embed information about the model into the generated code. This can make it more difficult for programmer to understand the source code. It may also mean that large expanses of the code are declared "do-not-touch" and must not be modified manually. Many CASE tools can hopelessly mangle manually modified source code in the round-trip engineering process.

According to Hebbel (1997), embedded information in code files often leads to complaints by developers when they first begin using CASE tools. These complaints are understandable because the readability of the source code is significantly reduced by the CASE tool's information placed into comment blocks. Hebbel (1997) suggests that two things happen to nullify this complaint:

- First, developers get used to seeing the extra comments and begin to mentally filter them out. This might seem a bit odd to point out, but developers do in fact learn to selectively attenuate the noise level introduced by the tool into the code.
- The second thing that tends to offset the "code pollution problem" comes about through the use of sophisticated integrated development environments such as Microsoft's Visual C++ Symantec's Visual Café. These IDEs offer class and method browsers that make navigation to specific declarations and definitions very easy, so developers can bypass the noise level in the native code files.

I think that CASE tools can be restrictive because they dictate a user of the CASE tool to support a vendor specific source code style and format. Code

generators are usually not configurable and they are targeted to a restricted set of languages. How is the need for creativity and innovation combined with the requirements for a more controlled management practice?

It is important to allow users to work "their way". Tools that are rigid and force users into single modes of doing their work are very unproductive. The tool must allow users to work naturally, which may not even be the same way from day to day or context to context. For example, you may wish to create use cases and then sequence diagrams, still, no "real" classes. Or you may go from use cases to object modeling. Or simply a list of requirement "bullets" and then on to class and sequence diagrams. No matter which style, the tool should never get in your way. (TogetherSoft, 2000)

Collaboration between tools causes problems in round-trip engineering. Tools involved in the round-trip engineering process include CASE tools, development tools, report and documentation tools, version control systems, databases, repositories etc. All these tools have their own proprietary formats. Round-trip engineering in a team environment with several tools isn’t possible without a complete interoperability between tools. Too often the development has to be based on disparate chunks of applications desperately trying to work together

Demeyer *et al.* (1999) have found that proprietary formats cause problems. Each tool is forced to extend UML to express the insufficiencies. This may cause problems because the protection of the standard is abandoned and with that the reliability necessary to achieve true interoperability. Worse, each tool will define its own extensions that are not understood by others.

8. Example of development with round-trip engineering and UML

8.1. General

In this example I created an automatic teller machine (ATM) simulation system using UML and round-trip engineering. The UML modeling was done with Rational Rose 98i Enterprise Edition with Rose J add-in and the final implementation with Symantec Visual Café. The UML models and Java source were kept in-sync with batch-mode round-trip engineering.

8.2. The development process

The example system was developed with an iterative and incremental approach. I concentrated only on phases relevant to round-trip engineering. First I designed the high level architecture with a UML package diagram. Then I proceeded to design each package in more detail. In the first increment I designed only the most essential classes. When the class diagrams were modeled, I forward engineered these class diagrams into Java source code. At this stage, the development was continued in Symantec Visual Café IDE. After some coding, debugging and testing in the IDE, the first increment of the software was ready. The second increment started out more UML modeling. First I had to reverse engineer my Java source code back into Rational Rose, so that the design models would reflect the modified source code. In this second increment I added more classes and refined the classes further. Finally I forward engineered the class diagram into Java source code. I didn't use UML to model the user interface package because it is much easier to just design the user interface graphically in the IDE and then reverse engineer the UI classes back into UML models.

Figure 36 illustrates the high level view of the ATM system and the dependencies between subsystems. The system consists of two subsystems, UIPackage and BusinessLogicPackage. Both subsystems utilize platform services in sun and java packages. The UIPackage contains only the user interface, BusinessLogicPackage contains the services that the user interface uses.

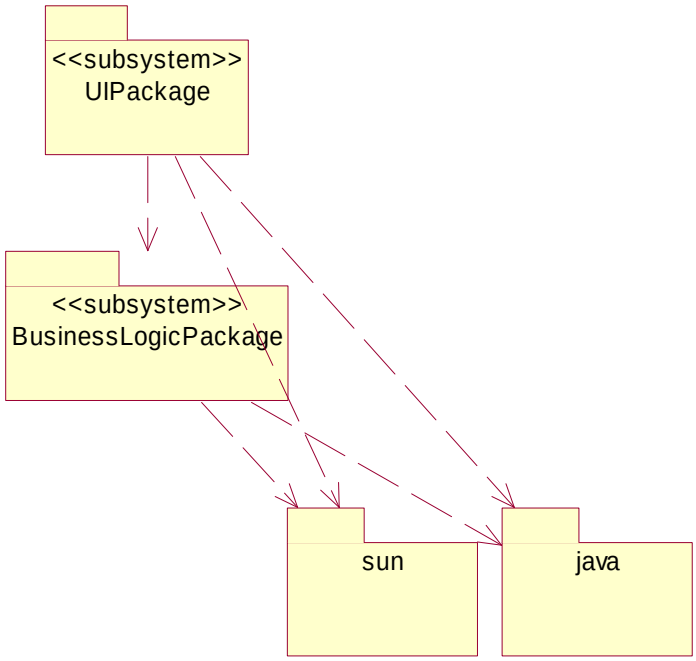

Figure 36. High level view of ATM system.

In the first increment the BusinessLogicPackage contains only two classes: ATM and Bank (see Figure 37). The source code generated in the forward engineering can be found in the Appendix.

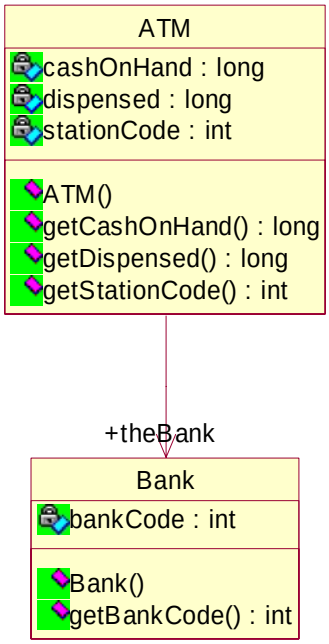

Figure 37. BusinessLogicPackage, first version.

In the second increment more classes were added to the BusinessLogicPackage (see Figure 38). The source code generated in the forward engineering can be found in the Appendix.

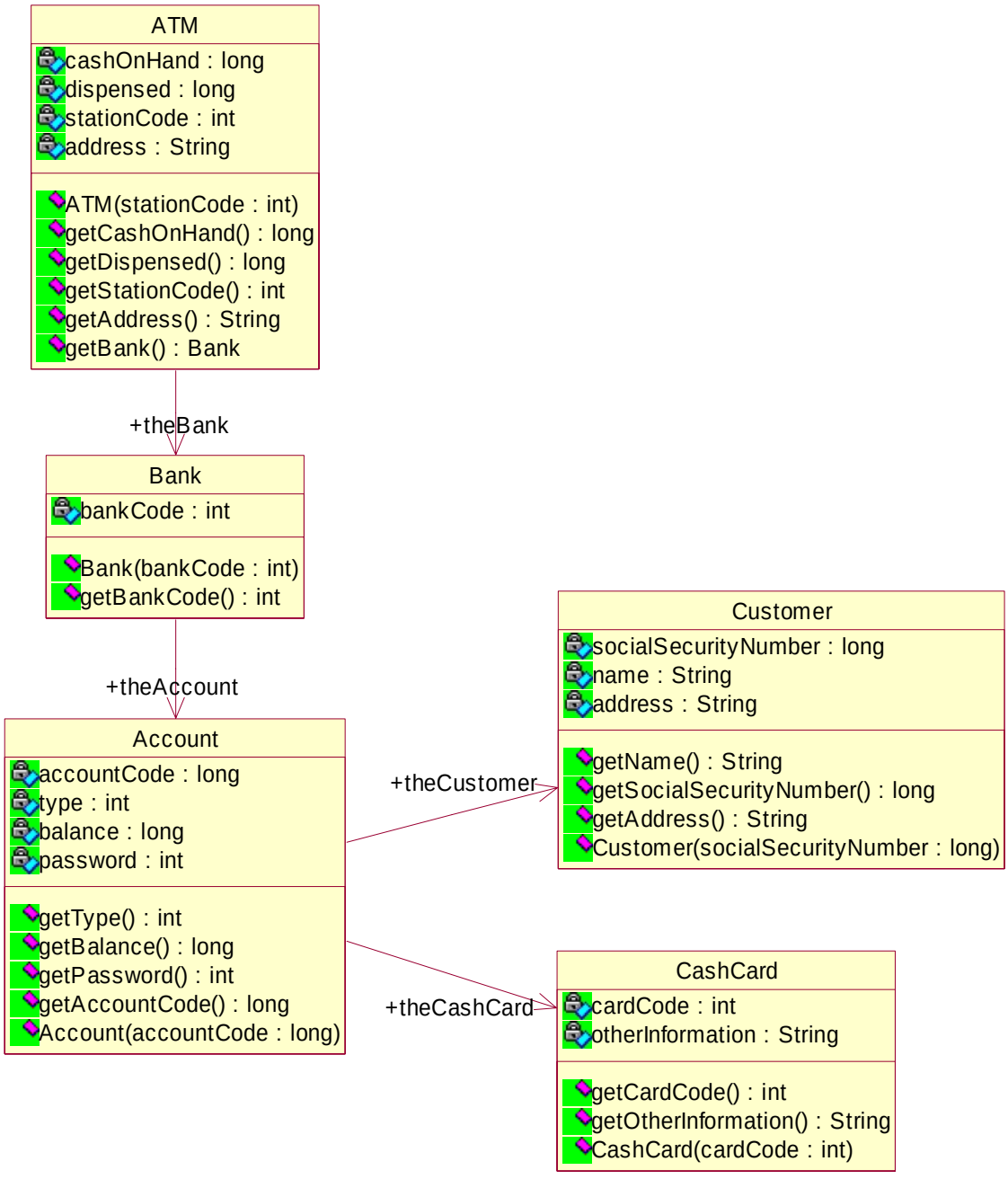

Figure 38. BusinessLogicPackage, second version.

The graphical user interface and all the UI classes were designed in Visual Café IDE (see Figure 39). The classes in the BusinessLogicPackage were inserted into the Visual Café project and the functionality of the system was refined further in this environment.

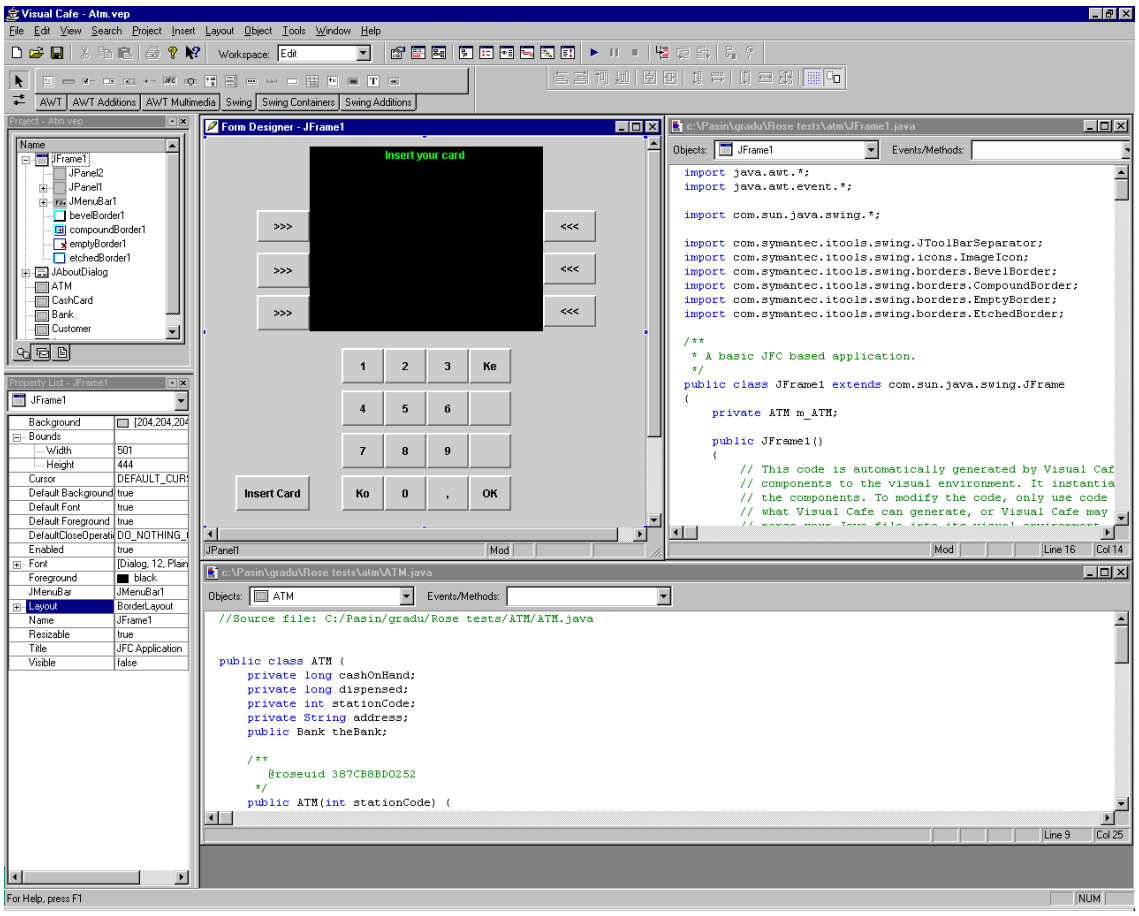

Figure 39. ATM user interface developed in Visual Café IDE.

8.3. Lessions learned

Rational Rose's round-trip engineering features are limited to only class, component and package diagrams. Other diagrams in UML are useless in Rose's round-trip engineering process. Rose doesn't support simultaneous round-trip engineering. It is up to the developer to explicitly execute the forward engineering and reverse engineering processes.

Forward engineering in Rational Rose is component-centered. The Java source generation is based on the component specification rather than on the class specification. This means that each class has to be assigned to a component before forward engineering. In Rational Rose, components and component diagrams represent the physical (versus the logical) structure of your model. Hence, in code generation and reverse engineering Rose is based on components to model the physical Java file structure. (Rose 98i, 1998)

When the developer forward engineers a class or a package, its characteristics are mapped to corresponding Java constructs. For example, Rose classes are forward engineered as Java classes, Rose components are forward engineered as .java files, and so on. Also, when the developer forward

engineers a package, a .java file is generated for each component belonging to the package. Each component's .java file will contain the definitions for any classes assigned to that component.

I discovered a lot of problems because the UML models and source code were not automatically synchronized all the time. Rational Rose is limited to batch-mode round-trip engineering, which I think is a major disadvantage. In batch-mode round-trip engineering strict control is required from the development process in order that forward or reverse engineering is not executed in the wrong time. It is up to the developer to explicitly execute the forward and reverse engineering processes. It was difficult to know whether the model is up-to-date with the changes in code, or if the code is up-to-date with the changes in the model. A couple of times I lost the changes done in the source code because I forward engineered the old UML model.

Most organizations have their own coding standards and it is important that also the CASE tool generates source code that conforms to these standards. However, it was not possible to customize forward engineering or reverse engineering in Rational Rose. Rational Rose also added proprietary markers in the form of specially formatted comments into the source code (see Appendix for examples). However, Rational Rose uses these markers only as a quick lookup device for subsequent reverse engineering. While it is recommend that the tags are left in place, they aren't required. There is no significant affect on reverse engineering, if the comments are deleted.

It is not feasible to use UML models to design graphical user interface (GUI) classes. The easiest way to design a GUI to use the IDE's GUI designer tool, like Symantec Visual Café's Form Designer. With this kind of tool it is much easier to design the GUI by visually laying out the GUI components. As the GUI is designed visually, Visual Café automatically creates the Java code and updates the code. Thus, these tools have much same kind of round-trip engineering features as the CASE tools do. The major difference is that design and implementation is done in the same environment (IDE) all the time. The developer doesn't have to switch between separate environments. This is exactly what round-trip engineering CASE tools should be aiming for. An integrated CASE tool and IDE would probably be the most effective environment for round-trip engineering. Some tools are already moving towards this direction. Microsoft Visual Studio includes Visual Modeler that is a cut-down version of Rational Rose. Visual Modeler support all the main Visual Studio languages, being able to reverse-engineer classes from existing code and forward engineering new code in Visual Basic, Visual C++ or Java. Visual Modeler supports only a subset of UML.

9. Future research

There are a number of interesting topics related to round-trip engineering that are outside the scope of this thesis. In this chapter I present these topics for possible future research.

Software process is an interesting research area in the context of round-trip engineering. Other software processes besides OMT++, for example RUP (Kruchten, 2000) and OPEN (Graham *et al.*, 1997), could be studied from the point of the view of round-trip engineering.

Round-trip engineering can be considered as a way to improve a software process. To build a process infrastructure, organizations producing software need ways to appraise their ability to perform their software process successfully. They also need guidance to improve their process capability. There are a number of fully defined models providing organizations with an effective guidance for establishing process improvement programs. These models include for example *The Capacity Maturity Model for Software* (Paulk *et al.*, 1993) and *SPICE* (Emam *et al.*, 1998).

Round-trip engineering can also be considered as a way to automate a software process. Automation is the traditional industrial means for improving productivity and product quality. There is little practical experience in the day-to-day use of software process automation, but this fairly new technology has the potential to significantly improve software quality and software development productivity. The Software Engineering Institute (SEI) has done a lot of interesting research on this area. One of the best reports is the *Practical Guide to the Technology and Adoption of Software Process Automation* (Christie, 1994). SEI has also conducted a number of empirical studies on software process automation that show interesting findings on drivers and inhibitors, contributors to success and technology issues.

In this thesis, J-UML, UOL and XMI are covered only in general level. A thorough investigation into these methodologies could result in valuable results for round-trip engineering. There are other object-oriented modeling languages that could be studied in the context of round-trip engineering. While the UML metamodel and notation aim to be comprehensive, there are a number of areas in which this modeling language is seen to be deficient. One possible alternative to UML could be *The Object Modeling Language* (OML) (Firesmith *et el.*, 1997).

CASE tools have an important role in round-trip engineering as these tools execute the round-trip engineering process. Therefore, research in this area is essential also for round-trip engineering, especially with focus on software

process automation and application generation. The CASE tool technology is still quite immature. As technology continues to evolve toward Web-based applications, distributed applications, and ever-more heterogeneous environments, it is likely that CASE tools vendors are playing catch-up for some time to come.

10. Conclusions

In this thesis I have studied the problem domain of round-trip engineering. The research problems were presented in Chapter 1. In the following, I give answers to each research problem.

The first research problem was the following: *How are the transitions between the design and implementation phases implemented in state-of-the-art software?* In this thesis I have studied a state-of-the-art software process: OMT++. It can be concluded that round-trip engineering isn't utilized in OMT++. Forward engineering isn't utilized because design is done at a higher level with less detail. Reverse-engineering isn't utilized because the design artifacts aren't kept up-to-date in the implementation phase. The approach of OMT++ is such that round-trip engineering is not necessarily required. The disadvantage of the OMT++ approach is that it's difficult to build upon previously constructed code if the documentation is not up-to-date. I think that documentation is as essential to the application as the source code itself, especially in big projects where good communication along with good understanding is the key to developing good software. According to Aalto and Jaaksi (1994), OMT++ provides a seamless transition from analysis to implementation. However, they specify the techniques of doing this only vaguely.

The second research problem was the following: *How can round-trip engineering solve the problems associated to the transitions between the design and implementation phases?* It can be concluded that in theory, round-trip engineering enables seamless transition between the design and implementation phases. The source code is automatically generated from UML models (forward engineering) and UML models are automatically kept up-to-date (reverse engineering). Seamless integration between the design and implementation phases is especially important in iterative development where the software is prototyped, tested, measured, and analyzed, and then refined in subsequent iterations.

The third research problem was the following: *How can round-trip engineering with UML be utilized effectively?* At the moment, UML is not a suitable modeling language for round-trip engineering. UML is a visual modeling language, not a visual programming language. Thus, one-to-one

mapping between UML and programming language is not possible. There are quite a few constructs in UML that cannot be translated into programming language, and vice-versa. In practice, round-trip engineering is possible only for class diagrams and in this case, only to some extent. Another problem is that UML is not formal enough for round-trip engineering. Adequate formalism is necessary in order to translate UML models *unambiguously* to programming language.

One possible solution to this problem is to extend UML in such a way that it is 100% consistent with the implementation language. There are two methodologies intended for this purpose: J-UML and UOL. J-UML specifies an implementation oriented UML extension for modeling Java implementations. In J-UML, an implementation-oriented model is constructed between UML and the implementation language (Java in J-UML). UOL is a formal method. It enables round-trip engineering by defining the UML models in formal textual constraints.

J-UML is clearly more feasible than UOL for round-trip engineering. Experience tells us that software developers will not be willing to use abstract formal languages and notations to design software. However, J-UML isn't fully mature yet and it lacks support from major tool vendors.

Apart from methodologies, tool interoperability needs to be improved, otherwise the various tools in round-trip engineering cannot fully work together. So far, we've reached consensus on a common notation that helps both tool vendors and program designers to concentrate on more relevant issues than the direction in which arrows should be drawn, or the question whether to represent classes as rectangles or clouds. Consensus should also be reached on a common format for storing UML artifacts. The XMI specification integrates XML, UML and MOF specifications adopted by OMG. XMI promotes a repository approach where all the software development artifacts are stored in a shared repository and tools, like CASE tools, IDEs, report generators, only act as front ends to the repository. When all the software development artifacts are stored in the repository, design and implementation artifacts are always up-to-date and consistent. Adequate formalism enables unambiguous mapping between UML models and programming language. At the moment it seems that XML is becoming the universal data format in the industry. Most of the major vendors of UML CASE tools and other development tools already support XML and XMI.

The UML CASE tools don't yet have good enough support for round-trip engineering. There are some tools that have decent support for round-trip engineering, like Rational Rose from Rational Software and Together from

Together Software. The major problem with CASE tools in the context of round-trip engineering is that they dictate the way the developer works. The developer can’t customize the tool enough to reflect his way of working. It is important to allow users to work "their way". Tools that are rigid and force users into single modes of doing their work are very unproductive. Also, CASE tools usually embed information about the model into the generated code in the form of specially formatted comments. This can make it more difficult for programmer to understand the source code. It may also mean that large expanses of the code are declared "do-not-touch" and must not be modified manually.

It can be concluded that the trend in software engineering is towards programming in a higher level. In the past, programming languages were at a quite low level. Now the implementation deals with higher level concepts that are closer to real-life. With round-trip engineering, most of the software development is actually done at a higher level (UML models) with concepts that reflect real-life. Hence, the focus of software development is shifting towards design and analysis. I predict that in the following years, round-trip engineering will gain more support from tool vendors and other organizations. At the moment it is difficult to say whether the community of developers will accept the solution and what form it will eventually take.

References

(Aalto and Jaaksi, 1994) Juha-Markus Aalto and Ari Jaaksi, Object-Oriented Development of Interactive Systems with OMT++. In: *TOOLS 14, Technology of Object-Oriented Languages & Systems*, 205-218. Prentice Hall, 1994. Available as *http://www.cs.uta.fi/~aj/omtpp/tools94.ps* .

(Ambler, 1997) Scott W. Ambler, An object-oriented visual glossary. An AmbySoft Inc. white paper, 1997. Available as *http://www.ambysoft.com/ooGlossary.pdf*.

(Booch, 1998) Grady Booch, Software architecture and the UML. *UML World*, 1998. Available as *http://www.rational.com/products/rup/index.jtmpl*.

(Booch *et al.*, 1998) Grady Booch, James Rumbaugh, and Ivar Jacobson, *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

(Boyd, 1999) Nik Boyd, Using natural language in software development, 2000. Available as *http://www.jps.net/nikboyd/papers/rhetoric/road/index.htm* .

(Brodsky, 1999) Stephen Brodsky, XMI open application interchange. IBM white paper, 1999. Available as *http://www-4.ibm.com/software/ad/standards/xmiwhite0399.pdf*.

(Christie, 1994) Alan M. Christie, A practical guide to the technology and adoption of software process automation. Software Engineering Institute, technical report, CMU / SEI-94-TR-007 ESC-TR-94-007.

(Coad and Yourdon, 1991a) Peter Coad and Edward Yourdon, *Object-Oriented Analysis.* Englewood Cliffs, N.J.: Prentice Hall, 1991.

(Coad and Yourdon, 1991b) Peter Coad and Edward Yourdon, *Object-Oriented Analysis Design*. Englewood Cliffs, N.J.: Prentice Hall, 1991.

(Coleman *et al.*, 1993) Derek Coleman, Stephanie Bodoff, and Patrick Arnold, *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1993.

(Conallen, 1997) Jim Conallen, Incorporating developer feedback into the model. Conallen white papers, 1997. Available as *http://www.conallen.com/whitepapers/devfeedback/DevFeedback.htm*.

(Emam *et al.*, 1998) Khaled El Emam, Drouin, J. –N., and Melo W., SPICE – the theory and practice of software process improvement and capability Determination. IEEE Computer Society (1998).

(Evans *et al.*, 1999) Andy Evans, Steve Cook, Steve Mellor, and Jos Warmer, Alan Wills, Advanced methods and tools for a precise UML. Available as *http://www.cs.york.ac.uk/puml/papers/panel.pdf* .

(Demeyer *et al.*, 1999) Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar, UML shortcoming for coping with round-trip engineering. In: *UML'99 Conference Proceedings, Springer-Verlag.*

(Firesmith *et el.*, 1997) Don Firesmith, Brian Henderson-Sellers, and Ian Graham, *The OML Reference Manual*. SIGS Books, 1997.

(Fowler and Scott, 1999) Martin Fowler and Kendall Scott, *UML Distilled, Second Edition, A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 1999.

(Graham *et al.*, 1997) Ian Graham, Brian Henderson-Sellers, and Houman Younessi, *The Open Process Specification*. Addison-Wesley, 1997.

(Haikala and Märijärvi, 1998) Ilkka Haikala and Jukka Märijärvi, *Ohjelmistotuotanto*. Suomen Atk-kustannus Oy, Espoo, 1998.

(Hebbel, 1997) Fred Hebbel, Using object modeling CASE tools. *DBMS and Internet Systems* **10**, 8 (July 1997), 94-95. Available as *http://www.dbmsmag.com/9707d16.html*.

(Jaaksi *et al.*, 1999) Ari Jaaksi, Juha-Markus Aalto, Ari Aalto, and Kimmo Vättö, *Tried and True Object Development: Practical Approaches with UML*. Cambridge University Press, 1999.

(Jacobson, 1998) Ivar Jacobson, Applying the UML in the Unified Process. In: *UML World 1998*. Available as *http://www.rational.com/products/rup/index.jtmpl*.

(Jacobson *et al.*, 1992) I. Jacobson, M. Christenson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering – a Use Case Driven Approach*. Addison-Wesley, 1992.

(Kaitanen, 1999) Kari Kaitanen, J-UML specification, version 1.02. VTT, 1999. Available as *http://www.vtt.fi/tte/papers/j-uml/j-uml_specification.htm*.

(Koskimies *et al.*, 1996) Kai Koskimies, Tatu Männistö, Tarja Systä, and Jyrki Tuomi, Automated support for dynamic modeling of object-oriented software. Tampere University of Technology and University of Tampere, 1996. Available as *http://www.uta.fi/~cstasy/report.htm*.

(Kruchten, 2000) Philippe Kruchten, *The Rational Unified Process, An Introduction*. Addison-Wesley, 2000.

(Kuusela and Aalto, 1993) Juha Kuusela and Juha-Markus Aalto, OMT+ ohje. Nokia Research Center, 1993. (Internal document)

(Marciniak, 1994) John J. Marciniak, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.

(MOF 1.3, 1999) Object Management Group, Inc., Meta Object Facility (MOF) Specification, Version 1.3 RTF, 1999. Available as *http://www.dstc.com/Research/Projects/MOF/rtf/mof-rtf.bk.final-ncb.pdf*.

(Paulk *et al.*, 1993) Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, Capability maturity model for software, version 1.1. Software Engineering Institute, CMU / SEI-93-TR-24.

(Rational, 2000) Rational Rose product information. Available as *http://www.rational.com/products/rose/index.jtmpl*.

(Rose 98i, 1998) Rational Rose 98i Enterprise Edition online help.

(Rumbaugh *et al.*, 1991) James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-oriented Modeling and Design*. Prentice-Hall International, Inc., 1991.

(UML 1.3, 1999) Object Management Group, Inc., OMG Unified Modeling Language specification version 1.3, June 1999. Available as *http://www.omg.org/cgi-bin/doc?ad/99-06-08.pdf*.

(UOL 1.2, 1998) Recerca Informàtica, Universitat Politecnica de Catalunya and Daimler-Benz Research and Technology, Universal Object Language 1.2. specification, July 1998. Available as *http://www.recercai.com*.

(Si Alhir, 1998) Sinan Si Alhir, *UML in a Nutshell*. O'Reilly & Associates, Inc., 1998.

(Suzuki and Yamamoto, 1998) Junichi Suzuki and Yoshikazu Yamamoto, Making UML models interoperable with UXF. In: *Lecture note in Computer Science (LNCS) 1618, Springer-Verlag Heidelberg*. Available as *http://www.yy.cs.keio.ac.jp/~suzuki/project/pub/lncs_uml.ps.zip*.

(TogetherSoft, 2000) TogetherSoft: Together/J, 2000 Available as *http://www.togethersoft.com/together/togetherJ.html*.

(Webster, 1998) Merriam-Webster OnLine, 1999. Available as *http://www.m-w.com*.

(XMI, 1998) Object Management Group, Inc., XML Metadata Interface (XMI). Available as *ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf*.

(XML 1.0, 1998) W3C recommendation, Extensible Markup Language (XML) 1.0. 10-February-1998. Available as *http://www.w3.org/TR/1998/REC-xml-19980210.pdf*.

Appendix

In this appendix I have included source code produced in the ATM example.

Java code from ATM BusinessLogicPackage / first version

```
public class ATM {
            private long cashOnHand;
            private long dispensed;
            private int stationCode;
            public Bank theBank;

            /**
               @roseuid 387A071D027E
             */
            public ATM(int stationCode) {}

            /**
               @roseuid 387A071D02E2
             */
            public long getCashOnHand() {}

            /**
               @roseuid 387A071D0301
             */
            public long getDispensed() {}

            /**
               @roseuid 387A071D0351
             */
            public int getStationCode() {}
}

public class Bank {
            private int bankCode;

            /**
               @roseuid 387A097E01E4
             */
            public Bank() {}
```

```
                /**
                   @roseuid 387A097E023E
                 */
                public int getBankCode() {}
   }
```

Java code from ATM BusinessLogicPackage / second version

```
   public class ATM {
                private long cashOnHand;
                private long dispensed;
                private int stationCode;
                private String address;
                public Bank theBank;

                /**
                   @roseuid 387CB8BD0252
                 */
                public ATM(int stationCode) {
                                this.stationCode = stationCode;
                                this.theBank = new Bank(0000);
                }

                /**
                   @roseuid 387CB8BD0325
                 */
                public long getCashOnHand() {
                                return cashOnHand;
                }

                /**
                   @roseuid 387CB8BD034D
                 */
                public long getDispensed() {
                                return dispensed;
                }

                /**
                   @roseuid 38832798035A
```

```
                 */
                public int getStationCode() {
                                return stationCode;
                }

                /**
                   @roseuid 388328CE0194
                 */
                public String getAddress() {
                                return address;
                }

                /**
                   @roseuid 388339220248
                 */
                public Bank getBank() {
                                return theBank;
                }
}

public class Bank {
                private int bankCode;
                public Account theAccount;

                /**
                   @roseuid 387A097E01E4
                 */
                public Bank(int bankCode) {
                                this.bankCode = bankCode;
                }

                /**
                   @roseuid 387A097E023E
                 */
                public int getBankCode() {
                                return bankCode;
                }
}
public class Account {
```

```
private long accountCode;
private int type;
private long balance;
private int password;
public Customer theCustomer;
public CashCard theCashCard;

/**
   @roseuid 387CC0BD0293
 */
public int getType() {
                return type;
}

/**
   @roseuid 387CC0C30346
 */
public long getBalance() {
                return balance;
}

/**
   @roseuid 387CC0C90146
 */
public int getPassword() {
                return password;
}

/**
   @roseuid 387CC0D6028F
 */
public long getAccountCode() {
                return accountCode;
}

/**
   @roseuid 387CC3C6033B
 */
public Account(long accountCode) {
```

```
                        this.accountCode = accountCode;
            }
}
public class Customer {
            private long socialSecurityNumber;
            private String name;
            private String address;

            /**
               @roseuid 387CC11A0391
             */
            public String getName() {
                        return name;
            }

            /**
               @roseuid 387CC1270372
             */
            public long getSocialSecurityNumber() {
                        return socialSecurityNumber;
            }

            /**
               @roseuid 387CC1320396
             */
            public String getAddress() {
                        return address;
            }

            /**
               @roseuid 387CC3C700BB
             */
            public Customer(long socialSecurityNumber) {
                        this.socialSecurityNumber =
                        socialSecurityNumber;
            }
}
public class CashCard {
            private int cardCode;
```

```
                private String otherInformation;

                /**
                   @roseuid 387CC13F025E
                 */
                public int getCardCode() {
                                return cardCode;
                }

                /**
                   @roseuid 387CC14500FE
                 */
                public String getOtherInformation() {
                                return otherInformation;
                }

                /**
                   @roseuid 387CC3C701CA
                 */
                public CashCard(int cardCode) {
                                this.cardCode = cardCode;
                }
  }
```