Chapter1

INTRODUCTION

1.1 SCOPE

The main scope of project:-

This project is consequences of research of area UML modeling. This project is going to help the user to model a new type of UML diagram. Project has wide scope in deriving executable java code. Project is also model UML diagram from Java code. Project deals with run able UML codes. Project is widely helpful for those users who are less efficient in java programming as well as less efficient in UML drawing. This thesis is always tried to present an output which must be better then Rational Rose, Meta Edit etc tools. The modelling language which attempts to merge all the pre-existing approaches to a modelling language for computer science is named the Unified Modelling Language – and is, in fact, the outcome of a method war of the 1990's. Today the UML is widely used as a notation when modelling software systems. This is the case with development processes as different as the Unified Process and the process suggested by the proponents of Extreme Programming.

1.2 OBJECTIVE

Object Oriented Programming is new evolving approach to model the requirement and convert them in to design. Later design derives the run able code. OOAD approach is now days customizes. Objective of this thesis is to derive a method whereby user can covert UML diagram To executable code easily. Software systems are growing in complexity - this is a statement which can be read in almost any treatment of computer science. As a matter of fact, software systems are in their increasing complexity harder to build, the process of building them is harder to manage and finally, they are much harder to maintain. As in any field of engineering, visualization can help to keep the system more easily manageable. Many software engineers can explain their thoughts easier in drawings than in words, and even though a good algorithm can be self explaining by the means of its pure existence, a good visualization can help in understanding as more of the mental capacity is used when drawings are involved. Ultimately the problem has risen up if this modelling language could provide the basis for the next step of abstraction in the development of programming languages and aid in bringing the software developer closer to being an architect using blueprints the UML provides. This step of abstraction would mean that the architect does not have to be personally involved in anything that follows the creation of his blueprints; others could finish his work based on the instructions embedded in the models he provides. That would not mean a complete abstraction from code as the blueprints can – and do in the proposals treated in this thesis – contain code, in the same way as the blueprints of architecture may contain directives in natural language, be it about measurements of a building or the texture of a wall. Essential is that a building can be generated in a straightforward way from the blueprint - and a software system might once be generated from an UML model.

The analogy to the science of architecture is often found in computer science; the very useful notion of patterns to be used in the software development process has been based on the findings of Christopher Alexander, an architect who proposed reusing well known patterns in designing buildings and their environments. Two ways have been proposed to achieve this goal.

Interaction diagrams: Interaction diagram helps to model the sequence diagram and collaboration diagram. It also generates the method bodies of sequence and collaboration diagram. The static structure of a software system can be generated by UML tools and provides a framework for these method bodies.

State chart diagrams: Interaction diagrams play a less important role in the suggested proposals based on generic state machine approach. The state chart diagrams of UML can be used to automatically generate executable code. This execution generally happens on the basis of a generic state machine.

UML provides a set of diagrams to model structural and behavioural aspects of an object-oriented system. The UML diagrams created in the design phase to model an object-oriented system are later used in the implementation phase. UML diagrams are powerful enough to hold most of the implementation details. However, manual translation of UML diagrams into object-oriented code does not guarantee conformance of the code with the UML diagrams due to the possibility of occurrence of human errors. Automatic translation of UML diagrams to object oriented code is highly desirable because it eliminates the chances of introduction of human errors in the translation process. Automatic code generation is efficient which, in turn, helps the software engineers deliver the software on time.

Realizing the potential of automated code generation, in recent years, the Object Management Group (OMG) introduced Model Driven Architecture (MDA) which supports automated translation of diagrams into code.

In this thesis, I append my work in on UML to Java Executable Code generation which generates executable Java code from UML 2.0 class, sequence, and activity diagrams. The Thesis provides warm knowledge of tool implementation. Implemented tool is developed to implement our proposed code generation idea. It combines both structural and behavioural features of an OO application. UML class diagram is used to generate code skeleton while UML sequence diagram adds behaviour in generated skeleton.

UML activity diagrams are used to provide code completeness and user interactions. Activity diagrams are referenced in sequence diagrams. Actions from UML superstructure are used in activity diagrams. The developed code generation tool is used to generate code for both the case studies to experimentally validate the generated code.

The results show that the generated code is consistent with the input UML diagrams. The generated code is fully understandable and functional. The simplicity of the generated Java code is ensured by the use of UML class and sequence diagrams. UML activity diagram actions incorporate fully functional features in the generated code which include object manipulations and user interactions. I have also compared our tool with the existing research-based, commercial, and open-source UML code generators. I conclude that the code generated by my tool is more thorough and understandable than the other tools.

1.3 Language customization

Customization of the language is making the language easier to use in ascertain context.

Before getting in to thesis, a crucial concept of customization must be defined as it is subsequently used in the thesis and all the related research.

A kind of definition explains how the customization is understood in the course of the thesis. The context in the definition is the purpose for which the customization is done.

For example it can be a particular domain or a specific software development process. If it is the process, the customization may involve making the semantics of the language more precise (i.e. providing additional consistency rules) or providing new modelling elements specific to the process. “Making the language easier to use” means that the customization provides new modelling elements or model libraries containing reusable solutions for a specific context. The Unified Modelling Language is a language which enables its customization in several ways which fulfil the above definition. The idea of one, unified language, which would be a basis for a whole family of languages, is a current issue in software engineering. Such an approach is deeply explored within the context of Meta modelling and Meta Object Facility. The design of the Unified Modelling Language allowing for the change of its specification by altering its Meta model provides a basis for language customization. The perspective on UML which puts it into the role of a family of languages rather than a single unified and fixed language is a starting point for extensibility of the language. The language was designed in such a way that it would enable adding new, “virtual” model constructs by the users of the language. However, the designers of the language reused for this purpose a concept which was initially not designed for this aim. It resulted in misunderstanding of one of the mechanisms for language extension and usage not according to the designers’ intentions.

1. With two exceptions; the first one is the publication based on my master thesis, in which the thesis formed a great majority of the paper; the second one is the series of two publications where the authors are ordered alphabetically with the exception when the contribution was not comparable with contributions of the other authors.

2. Having a fixed set of language constructs. The view of UML as a family of languages is an approach to using modelling languages in a more efficient way.

The main advantage of this approach is that it considers UML as a language which is twofold. It contains the core UML constructs, and the extended constructs. The core constructs are the same in every language in the family, and the extended constructs are specific for each language in the family. Cook spots the main drawbacks of UML models to be its usability in a variety of purposes with help of a fixed set of standard language elements. Since the semantics of similar constructs in different domains (for which the models are created) is not the same, the language should incorporate such variability. Examples of such two domains are multimedia systems, which stress the variability of multimedia items (modelled usually as classes) and real-time systems, which emphasize the timing constraints.

The standard UML has been found to be insufficient in areas critical to the development of domain specific software. For example, some limitations for real-time systems (modelling timing constraints and implementation architecture) have been identified by McLaughlin and Moore. The issues (among others) were later addressed by the Object Management Group (OMG) and resulted in the elaboration of the UML profile for performance and schedulability.

This and other similar initiatives lead to the perception of UML as a base language, which should be tailored for specific purposes incorporating domain-specific concepts.

It can be achieved by either using the built-in extension mechanisms of the language or engineering the language with help of metamodeling approaches. One of the built-in extension mechanisms in UML is the notion of stereotype. However, to be able to fully analyze stereotypes and their usage in UML, an overview of this notion in software development in general is required.

Metamodeling as an alternative for extension mechanisms

A completely different approach to customization of the language is the so-called first-class extensibility mechanism (heavyweight extension mechanism) – metamodeling. It is a mechanism and a technique of language engineering.

Although it can also be used for language customization, it is mainly aimed at creating, defining and changing of modelling languages and as such is not discussed in the thesis. The mechanism of metamodeling is beyond the UML specification and i can be used for changing the language to such extent, that it may not be possible to find commonalities with the original UML.

Significant amount of research on the issue of metamodeling and language engineering is being performed by the precise

UML group (pUML). A current initiative for formalization of the metamodeling approaches to language definitions, customizations and engineering is the Meta Object Facility – MOF – specification. It defines a four-layer metamodeling architecture in the context of which the UML definition is placed. The metamodeling architecture of MOF has a significant implication on the definition of UML and its customization mechanisms.

The metamodeling architecture layers can be summarized as follows:

Layer 3 (M3): meta-metmodel, defining the language used to define languages for modelling. The MOF itself is defined at this level, and is used to define languages at the M2 layer.

Layer 2 (M2): Meta model, which defines the language used for defining user created elements. The UML specification – abstract syntax – is placed at this level3.

Layer 1 (M1): model, which contains the user defined models. For example, the classes on UML class diagrams are the definitions of user objects, at the same time being instances of Meta classes from language definition.

Layer 0 (M0): user objects, which contain the instances of model elements. It corresponds to the level of objects that are instances of user defined classes. In the metamodeling approaches to language customization, the changes are introduced directly into the language specification – the Meta model. Since it is the modelling of modelling languages, a lot of issues associated with language engineering are involved.

However, they are outside of the scope of the thesis. Despite its advantages and the lack of restrictions, metamodeling has a significant threat – It should be stated that the UML version 1.5 is not an instance of MOF, however the future releases of the languages are supposed to be.

1.4 Stereotypes in software development

The common understanding of stereotypes is that they are a simplistic view of a group of people that share a common set of traits. An example of the widely known stereotype is the view of a young person, who is assumed to “love rock and roll and have no respect for elders”. Although neither all youngsters conform to the stereotype, nor is it their primary objective in life, they are perceived as such by a certain part of the human population. Stereotypes played a similar role in software development at the time they were introduced there. The idea of stereotyping was first introduced into software development by Rebecca Wirfs-Brock, who used the concept of stereotype to classify objects according to their “modus operandi”. Wirfs-Brock’s original intention behind the usage of stereotypes was similar to the aforementioned view of stereotypes in other areas i.e. as a way of oversimplifying the view of objects’ role or behavior.

She used a fixed set of stereotypes, useful in characterizing the special roles of objects in the system. The set consists of the following stereotypes:

- *Controller,* controlling the action of other objects,
- *Coordinator,* initiating activities and tying client objects with service provider objects,
- *Interfaces,* supporting communications between external systems and users,
- *Service Provider,* performing a specific operation on demand,
- *Information Holder,* holding values that other objects ask about, and
- *Structured,* primarily maintaining relationships.

The most significant part of the definition, adopted by behavioural sciences, is that the stereotype is “a mental picture…that represents an oversimplified opinion The Unified Modelling Language also contains a definition of stereotypes, but it specifies them as one of the possible extension mechanisms of the language. In UML, the stereotypes are a way of adding a new semantics to the existing model elements.

They allow branding the existing model elements with new semantics, thus enabling them to “look” and “behave” as virtual instances of new model elements. They are no longer seen (at least directly in the specification) as a way of additional classification of model elements, according to their “modus operandi”, but rather as a way of introducing new elements into the language. This perception of stereotypes contradicts with the original perception of the concept of stereotype. It is no longer aimed at providing oversimplification of elements, but at (to some extent) complication by adding new semantics to the already existing elements. On the other hand, the usage of stereotypes is not restricted to merely this purpose. The original way of using stereotypes according to Wirfs-Brock is not “illegal” with respect to the UML specification. An example of a stereotype in UML is presented in. It depicts the <<Interface>> stereotype, which changes the semantics of the class.

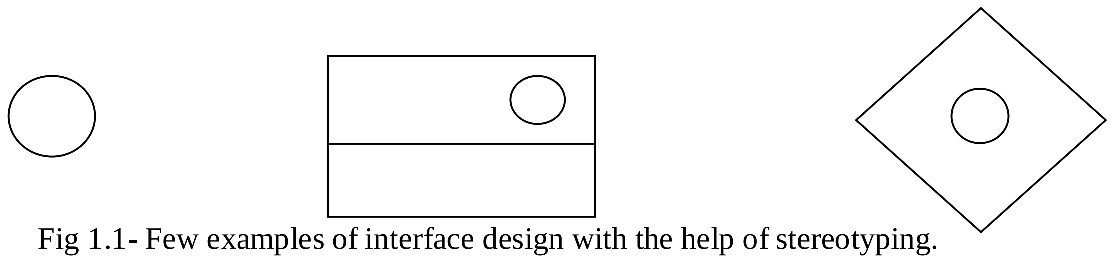


Fig 1.1- Few examples of interface design with the help of stereotyping.

1.4.1 Classification of stereotypes according to their expressiveness

According to the amount of changes in syntax and semantics they introduce. In their work the authors distinguished between four types of stereotypes.

1. *Decorative stereotypes*, i.e. stereotypes which do not change the semantics of a language element, but change its syntax (graphical representation),

2. *Descriptive stereotypes*, i.e. stereotypes which modify the abstract syntax of a language element and define the pragmatics of the newly introduced element without changing the semantics,

3. *Restrictive stereotypes* are descriptive stereotypes which modify the semantics of a language element,

4. *Redefining stereotypes*, which redefine a language element by changing its original semantics, w. r. t. syntax, they are similar to the restrictive stereotypes.

The classification attempts to address the complexity of a stereotype definition and provides guidelines for applying the different kinds of stereotypes. Although the authors present the examples of using the stereotypes for all types of stereotypes, they recommend a high degree of carefulness in using the redefining stereotypes.

Since they allow for a complete redefinition of a model element, they should be used by experienced method specialists and forbidden for "individual engineers or within projects". Although the classification addresses the problems of how stereotypes change the extended model element, it neglects the problems of practical aspects of mechanisms for supporting stereotypes and the metamodeling levels that the stereotype definition concerns.

1.4.2 Code Generation stereotypes

Certain stereotypes are intended for providing specific code generation features for the stereotyped elements. Since not all standard UML constructs can be translated directly into code in all programming languages, the precise code generation details can be specified by stereotypes, which can be interpreted by code generators. Code generation stereotypes are usually applied to modelling elements used in the late design stage, when the designs are supposed to be as closely related to the implementation as possible. Such stereotypes rarely put constraints on the usage of the stereotyped model elements in connection with other modelling elements.

However, the stereotypes restrict modelling elements to have only these constructs that can be mapped into the generated code. If the standard modelling element has constructs that cannot be expressed in the generated code, then such constructs should be forbidden by constraints attached to the stereotype. On the other hand, if there is additional information that may be added to the code, but is not a part of the standard model element, then tagged values are used to add the information. An example of such a stereotype is the stereotype defined for the singleton design pattern. Adding the singleton stereotype to a class results in the precise code generation for the class, as it is defined by the singleton design pattern. Fig. presents a class, which is stereotyped with the singleton stereotype. The standard code generation for this class in the Java language produces the following code, which ignores the existence of the stereotype:

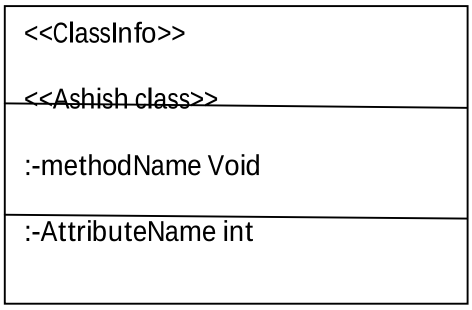


Fig 1.2 - Class Diagram

Class Ashish

{

Int attribute;

Public void method();

}

If the code generator is able to interpret the ClassInfo stereotype, the code generated for the same class is:

class Ashish

{

Int Attribute;

ClassInfo *_instance;

ClassInfo() {}

```
public:
static ClassInfo *getInstance() {
if (_instance == 0)
_instance = new Class();
return _instance;
}
};
```

1.5 Motivation:-

My motivation through this thesis is to explain how a new technique that convert code to UML diagram and UML diagram to executable Code in java. To understand better how I model UML diagram and further I covert that diagram into code, it is necessary to take a simple example. The motivation of this thesis is to evaluate the current situation of software development with UML. For further coming section we are going to take a simple example of ATM Machine problem to discuss the thesis. Furthermore, it shall be explained how far the above mentioned approaches have gone and how near the UML has come to the goal of full code generation, how much ground is still to cover if the current version of UML and the currently available tools are regarded. As an example a ATM Machine system for elementary school students shall be used. This system proposes a platform on which teachers can design exercise types and students can solve exams automatically created from these exercise types.

Modelling this ATM Machine seems to be easy if the description provided in upcoming section to read, but the many hidden obstacles in developing even such small a system provide a never-ending flood of possibilities, opportunities and threats. Explaining them and suggesting a solution for them will be the topic of this thesis.

The example is based on the requirement text shown below

The ATM verifies whether the customer's card number and PIN are correct. If it is, then the customer can check the account balance, deposit cash, and withdraw cash. Checking the balance displays the account balance. Depositing asks the customer to enter an amount, and then updates the balance Withdrawing asks the customer for the amount to withdraw; If the account has enough cash, the balance is updated. The ATM prints the customer's account balance on a receipt.

1.6 Goals:-

This thesis aims to implement the ATM Machine example in two ways: first using code generation from the starting point of UML interaction diagrams, and second using code generation from the starting point of executable UML state diagrams. The code generated from the models will either be executable Java code or C++ Code. Furthermore, this code should comprise a graphical user interface and a persistence layer. The documentation is not only about the final outcome, but also about the way that was gone to get there, particularly about the process of object-oriented analysis and design and about the major problems being encountered. A special effort is made to use patterns in all steps of the OOAD process and to show how these patterns can be applied in different situations of the development of a software system.

Chapter 2

Technical Background

This chapter will provide a technical background for a better understanding of the software development's requirements. The analysis is split into the organisational and the technical aspect. First, the organisational prerequisites are explained in the context of the software development life-cycle. As a life-cycle method, the Object-Oriented Analysis and Design is introduced, the organisational process is exemplified by the Rational Unified Process. A possible notation when working through the analysis and design phase of the software development life-cycle is UML, the first technology being introduced.
A profile of UML used to design executable models follows suit. Consecutively, the focus shifts to the implementation phase and Java as a quintessential Object-Oriented programming language is examined. Finally, the term pattern is explained as reuse of knowledge – patterns become more and more important for each phase of the software development life cycle.

2.1 The Software Development Life-Cycle

Systems Development Life Cycle (SDLC) is a process used by a systems analyst to develop an information system, including requirements, validation, training, and user (stakeholder) ownership. Any SDLC should result in a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance.

Computer systems are complex and often (especially with the recent rise of Service-Oriented Architecture) link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models or methodologies have been created, such as "wateSystems Development Life Cycle (SDLC) is a process used by a systems analyst to develop an information system, including requirements, validation, training, and user (stakeholder) ownership. Any SDLC should result in a high quality system that meets or exceeds customer expectations, reaches completion within time and cost estimates, works effectively and efficiently in the current and planned Information Technology infrastructure, and is inexpensive to maintain and cost-effective to enhance.

Computer systems are complex and often (especially with the recent rise of Service-Oriented Architecture) link multiple traditional systems potentially supplied by different software vendors. To manage this level of complexity, a number of SDLC models or methodologies have been created, such as "waterfall"; "spiral"; "Agile"; "rapid prototyping"; "incremental"; and "synchronize and stabilize".

SDLC models can be described along a spectrum of agile to iterative to sequential. Agile methodologies, such as XP and Scrum, focus on lightweight processes which allow for rapid changes along the development cycle. Iterative methodologies, such as Rational Unified Process and Dynamic Systems Development Method, focus on limited project scope and expanding or improving products by multiple iterations. Sequential or big-design-up-front (BDUF) models, such as Waterfall, focus on complete and correct planning to guide large projects and risks to successful and predictable results. Other models, such as Anamorphic Development, tend to focus on a form of development that is guided by project scope and adaptive iterations of feature development.

In project management a project can be defined both with a project life cycle (PLC) and an SDLC, during which slightly different activities occur. According to Taylor (2004) "the project life cycle encompasses all the activities of the project, while the systems development life cycle focuses on realizing the product requirements"

2.1.1 Waterfall Model

The **waterfall model** is a sequential design process, often used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.

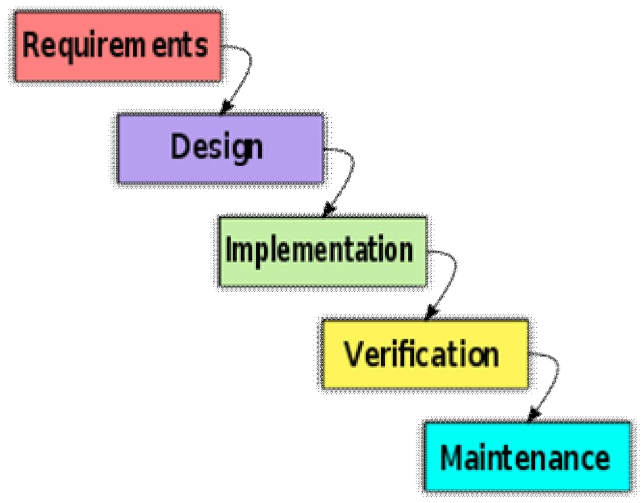


Figure 2.1 SDLC model

The unmodified "waterfall model”. Progress flows from the top to the bottom, like a waterfall.

The waterfall development model originates in the manufacturing and construction industries: highly structured physical environments in which after-the-fact changes are prohibitively costly, if not impossible. Since no formal software development methodologies existed at the time.

2.1.2 Spiral Model:-

The spiral model combines the idea of iterative development (prototyping) with the systematic, controlled aspects of the waterfall model. It allows for incremental releases of the product, or incremental refinement through each time around the spiral. The spiral model also explicitly includes risk management within software development. Identifying major risks, both technical and managerial, and determining how to lessen the risk helps keep the software development process under control.

The spiral model is based on continuous refinement of key products for requirements definition and analysis, system and software design, and implementation (the code).At each iteration around the cycle, the products are extensions of an earlier product. Thismodel uses many of the same phases as the waterfall model, in essentially the same order, separated by planning, risk assessment, and the building of prototypes and simulations.

Documents are produced when they are required, and the content reflects the information necessary at that point in the process. All documents will not be created at the beginning of the process, nor all at the end (hopefully).

2.1.3 IBM Rational Unified Process

RUP is based on a set of building blocks, or content elements, describing what is to be produced, the necessary skills required and the step-by-step explanation describing how specific development goals are to be achieved. The main building blocks, or content elements, are the following:

- Roles (who) – A Role defines a set of related skills, competencies and responsibilities.
- Work Products (what) – A Work Product represents something resulting from a task, including all the documents and models produced while working through the process.
- Tasks (how) – A Task describes a unit of work assigned to a Role that provides a meaningful result.

Within each iteration, the tasks are categorized into nine disciplines:

- Six "engineering disciplines"
    - Business Modeling
    - Requirements
    - Analysis and Design
    - Implementation
    - Test
    - Deployment

- Three supporting disciplines
    - Configuration and Change Management
    - Project Management
    - Environment

Four Project Life cycle Phases

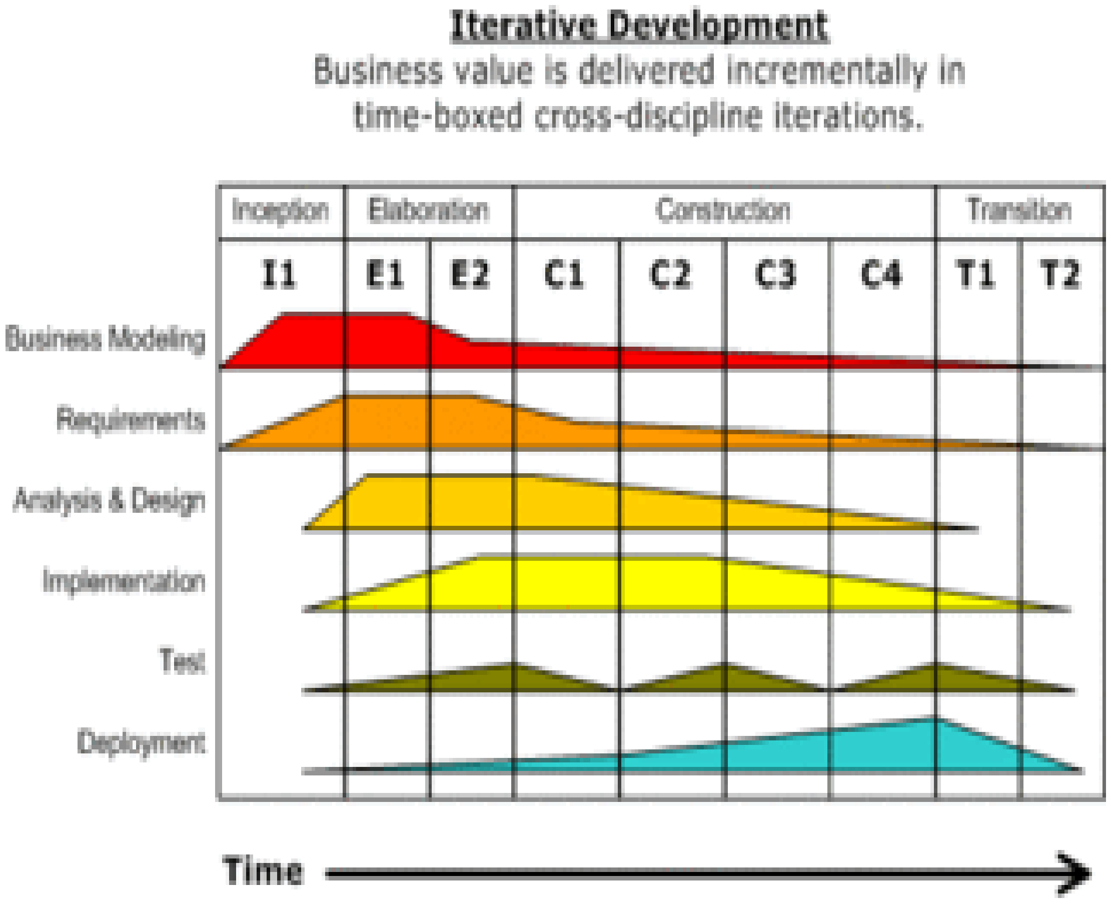


Figure 2.3 RUP phases and disciplines.

The RUP has determined a project life cycle consisting of four phases. These phases allow the process to be presented at a high level in a similar way to how a 'waterfall'-styled project might be presented, although in essence the key to the process lies in the iterations of development that lie within all of the phases. Also, each phase has one key objective and milestone at the end that denotes the objective being accomplished. The visualization of RUP phases and disciplines over time is referred to as the RUP hump chart.

Inception Phase

The primary objective is to scope the system adequately as a basis for validating initial costing and budgets. In this phase the business case which includes business context, success factors (expected revenue, market recognition, etc.), and financial forecast is established. To complement the business case, a basic use case model, project plan, initial risk assessment and project description (the core project requirements, constraints and key features) are generated. After these are completed, the project is checked against the following criteria:

- Stakeholder concurrence on scope definition and cost/schedule estimates.
- Requirements understanding as evidenced by the fidelity of the primary use cases.
- Credibility of the cost/schedule estimates, priorities, risks, and development process.
- Depth and breadth of any architectural prototype that was developed.
- Establishing a baseline by which to compare actual expenditures versus planned expenditures.

If the project does not pass this milestone, called the Lifecycle Objective Milestone, it either can be canceled or repeated after being redesigned to better meet the criteria.

Elaboration Phase

The primary objective is to mitigate the key risk items identified by analysis up to the end of this phase. The elaboration phase is where the project starts to take shape. In this phase the problem domain analysis is made and the architecture of the project gets its basic form.

This phase must pass the Lifecycle Architecture Milestone by meeting the following deliverables:

- A use-case model in which the use-cases and the actors have been identified and most of the use-case descriptions are developed. The use-case model should be 80% complete.
- A description of the software architecture in a software system development process.
- An executable architecture that realizes architecturally significant use cases.
- Business case and risk list which are revised.
- A development plan for the overall project.
- Prototypes that demonstrably mitigate each identified technical risk.

If the project cannot pass this milestone, there is still time for it to be canceled or redesigned. However, after leaving this phase, the project transitions into a high-risk operation where changes are much more difficult and detrimental when made.The key domain analysis for the elaboration is the system architecture.

Construction Phase

The primary objective is to build the software system. In this phase, the main focus is on the development of components and other features of the system. This is the phase when the bulk of the coding takes place. In larger projects, several construction iterations may be developed in an effort to divide the use cases into manageable segments that produce demonstrable prototypes.

This phase produces the first external release of the software. Its conclusion is marked by the Initial Operational Capability Milestone.

Transition Phase

The primary objective is to 'transit' the system from development into production, making it available to and understood by the end user. The activities of this phase include training the end users and maintainers and beta testing the system to validate it against the end users' expectations. The product is also checked against the quality level set in the Inception phase.

If all objectives are met, the Product Release Milestone is reached and the development cycle ends.

2.2 Object Oriented Analysis & Design-

In dealing with object-oriented technology, Object-Oriented Analysis and Design is the method of choice for the software development life-cycle. It can be applied in the analysis and design phase and provides general instructions as for what has to be accomplished. In discussing Object-Oriented Analysis and Design the distinction between these two phases has to be clarified first. According to [) OOAD-Roadmap] the distinction between OOA and OOD is the question the developer mainly poses. In the phase of OOA the typical question starts with what...? Like
 "What will my program need to do?", "What will the classes in my program be?" and "What will each class be responsible for?".

Hence, OOA cares about the real world and how to model this real world without getting into much detail. Larman describes in the OOA phase as an investigation of the problem and requirements, rather than finding a solution to the problem. In contrast, in the OOD phase, the question typically starts with How...? Like
"How will this class handle its responsibilities?", "How to ensure that this class knows all the information it needs?" and "How will classes in the design communicate?". The OOD phase deals with finding a conceptual solution to the problem – it is about fulfilling the requirements, but not about implementing the solution.
The two phases can be summarized in the short phrase:
"Do the right thing (analysis), and do the things right (design)".
 The transition from OOA to OOD seems to be rather easy: the task the developer has to fulfil is to enhance the OOA objects with the constructs being necessary to come closer to the actual software system.
OOA is an opponent of the easy transition view and sees a clear and distinct separation between objects in OOA and objects in OOD. Some of the objects elaborated in OOA are not necessary in OOD.

Some objects have to be created additionally and even the objects needed in both phases can have different attributes and behaviour and should therefore clearly be marked as different objects. Figure 2.4 visualizes these dependencies.

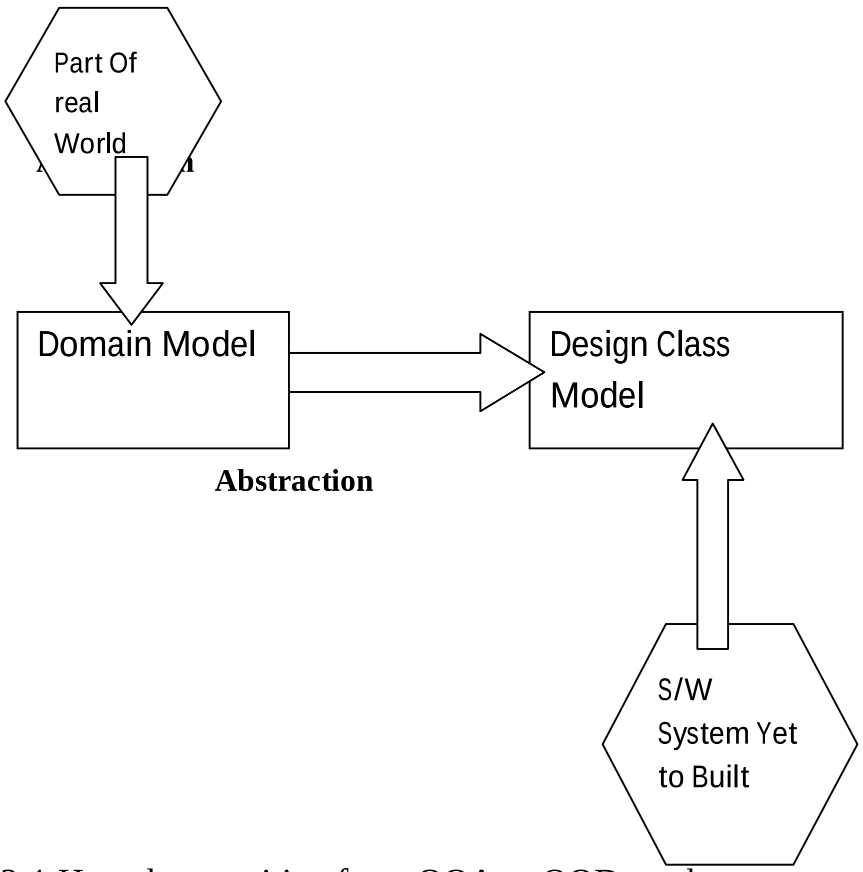


Figure 2.4-How the transition from OOA to OOD works

The Object-Oriented Design phase is not about abstracting from the real world anymore: it is rather about abstracting from a software system yet to be built – this is why this phase is so difficult to accomplish. Two of the models used in OOD are the interaction diagrams and the class diagrams of the design phase. The class diagrams of the design phase resemble very often the domain diagrams of the analysis phase but are - as defined above - clearly different constructs. By defining interaction diagrams, the path the information takes when it flows from object to object is outlined. Therefore, the most relevant information of this sort of diagram is in which order and under which conditions operations get called or messages.

2.3 The Unified Modelling Language

The Unified Modeling Language (UML) is used to specify, visualize, modify, construct and document the artifacts of an object-oriented software-intensive system under development. UML offers a standard way to visualize a system's architectural blueprints, including elements such as:

- actors
- business processes
- database schemas
- (logical) components
- programming language statements
- reusable software components.

UML combines techniques from data modeling (entity relationship diagrams), business modeling (work flows), object modeling, and component modeling. It can be used with all processes, throughout the software development life cycle, and across different implementation technologies. UML has synthesized the notations of the Booch method, the Object-modeling technique (OMT) and Object-oriented software engineering (OOSE) by fusing them into a single, common and widely usable modeling language. UML aims to be a standard modeling language which can model concurrent and distributed systems. UML is a de facto industry standard, and is evolving under the auspices of the Object Management Group (OMG).

UML models may be automatically transformed to other representations (e.g. Java) by means of QVT-like transformation languages. UML is extensible, with two mechanisms for customization: profiles and stereotypes.

2.3.1 before UML 1.x

Under the technical leadership of the Three Amigos, an international consortium called the UML Partners was organized in 1996 to complete the *Unified Modeling Language (UML)* specification, and propose it as a response to the OMG RFP. The UML Partners' UML 1.0 specification draft was proposed to the OMG in January 1997. During the same month the UML Partners formed a Semantics Task Force, chaired by Cris Kobryn and administered by Ed Eykholt, to finalize the semantics of the specification and integrate it with other standardization efforts. The result of this work, UML 1.1, was submitted to the OMG in August 1997 and adopted by the OMG in November 1997.

UML 1.x

Concepts from many other OO methods were also loosely integrated with UML with the intent that UML would support all OO methods. Many others also contributed, with their approaches flavoring the many models of the day, including: Tony Wasserman and Peter Pircher with the "Object-Oriented Structured Design (OOSD)" notation (not a method), Ray Buhr's "Systems Design with Ada", Archie Bowen's use case and timing analysis, Paul Ward's data analysis and David Harel's "Statecharts"; as the group tried to ensure broad coverage in the real-time systems domain. As a result, UML is useful in a variety of engineering problems, from single process, single user applications to concurrent, distributed systems, making UML rich but also large.

The Unified Modeling Language is an international standard.

2.4 UML 2.x

UML has matured significantly since UML 1.1. Several minor revisions (UML 1.3, 1.4, and 1.5) fixed shortcomings and bugs with the first version of UML, followed by the UML 2.0 major revision that was adopted by the OMG in 2005.

Although UML 2.1 was never released as a formal specification, versions 2.1.1 and 2.1.2 appeared in 2007, followed by UML 2.2 in February 2009. UML 2.3 was formally released in May 2010. UML 2.4 is in the beta stage as of March 2011.

There are four parts to the UML 2.x specification:

1. The Superstructure that defines the notation and semantics for diagrams and their model elements
2. The Infrastructure that defines the core metamodel on which the Superstructure is based
3. The Object Constraint Language (OCL) for defining rules for model elements
4. The UML Diagram Interchange that defines how UML 2 diagram layouts are exchanged

The current versions of these standards follow: UML Superstructure version 2.3, UML Infrastructure version 2.3, OCL version 2.2, and UML Diagram Interchange version 1.0.

Although many UML tools support some of the new features of UML 2.x, the OMG provides no test suite to objectively test compliance with its specifications.

Currently the OMG is working on extensions to the UML, the next version will be termed UML 2.0 and will be a major overhaul of the UML standard in trying to fix many of the issues currently regarded as pitfalls of the UML standard. Some of these pitfalls are:

• Incomplete semantics and notation for activity graphs: The construct of an activity graph was added relatively late into the concept of the UML, its elements depend on the semantics of a state-diagram but are not correctly derived from these basics.

• The standard elements bloat: Many competing groups tried to have UML tweaked to their necessities and so there were added standard elements that are semantically weak and not well defined.

• Architectural misalignments: The originally taken approach of a strict-metamodel architecture was replaced by a loose meta-modelling approach. This helped with deploying UML to the market faster, but is now an obstacle in integrating UML with the other major standards of the OMG, especially the Meta Object Facility (MOF).

Adds more problems to this list.

• Separate Specifications: The diagrams of the UML are specified separately, nothing is said about their interconnection and their relation to each other in a standardized way. This would resemble to specifying a Java class and a Java method, and saying nothing about how these two constructs interact.

Model Interchange: The interchange of models based on XML Metadata Interchange (XMI) is an interchange based on the abstract syntax of diagrams. Rather than this interchange based on syntax, different tools should be able to exchange models based on the concrete meaning of these models.

• Presentation layer: Right at the core of UML should be a layer where the meaning is defined, and on top the presentation should be layered. Each developer should be able to change the presentation layer as he desires, but still work on the same core components.

Some of these problems will not be fixed by UML version 2.0, and some might be solved which did not occur in the list above – this depends on which proposal the OMG will finally decide upon as there are five proposals for the next generation of UML. An outline of each of these is given below, along with the respective abbreviation used for distinguishing the different proposals below in section

The UML version 2.0 consists of diagram types enabling the developer to express her software building plans in manifold ways. Duplicity is inherent in these diagram types -as the definition of these types has arisen from the many interest groups who all wanted their notation to be in UML. When modelling with UML, any of the following eight diagram types can be chosen.

1. Static structure diagram is often termed class diagram, and describes the static aspect of the system in the form of classes, packages and their relations.
2. Use case diagram describes the functionality of the software system from the viewpoint of the user. This is a top level view of the system; the details are not visible at this level of abstraction.

3. Sequence Diagram shows the interactions occurring between objects to fulfil a certain task, which might be anything from an use case to an operation. Sequence Diagrams show these interactions with respect to the time passing between the occurring calls.

4. Collaboration Diagram is essentially the same construct as the sequence diagram. However, the focus here is not mainly on the sequence of events but the structural relationship of the objects taking part in fulfilling the task.

5. State Chart Diagram describes the life-cycle of an object. The three basic constructs are states, transitions and events. An object remains in a state and – upon occurrence of an event - transitions to the next state.

6. Activity Diagram describes a sequence of actions and is used to specify use cases in more detail and to model business processes. Its semantics are based on those of the state chart diagram, which is an often criticized point of UML – the semantics of state chart diagrams are not viable to express all the constructs generally being used in modelling business processes.

7. Component Diagram shows the dependency of components on each other. Components are pieces of source-code, binary-code or executable code.

8. Deployment diagram visualizes the architecture of the software system as a graph

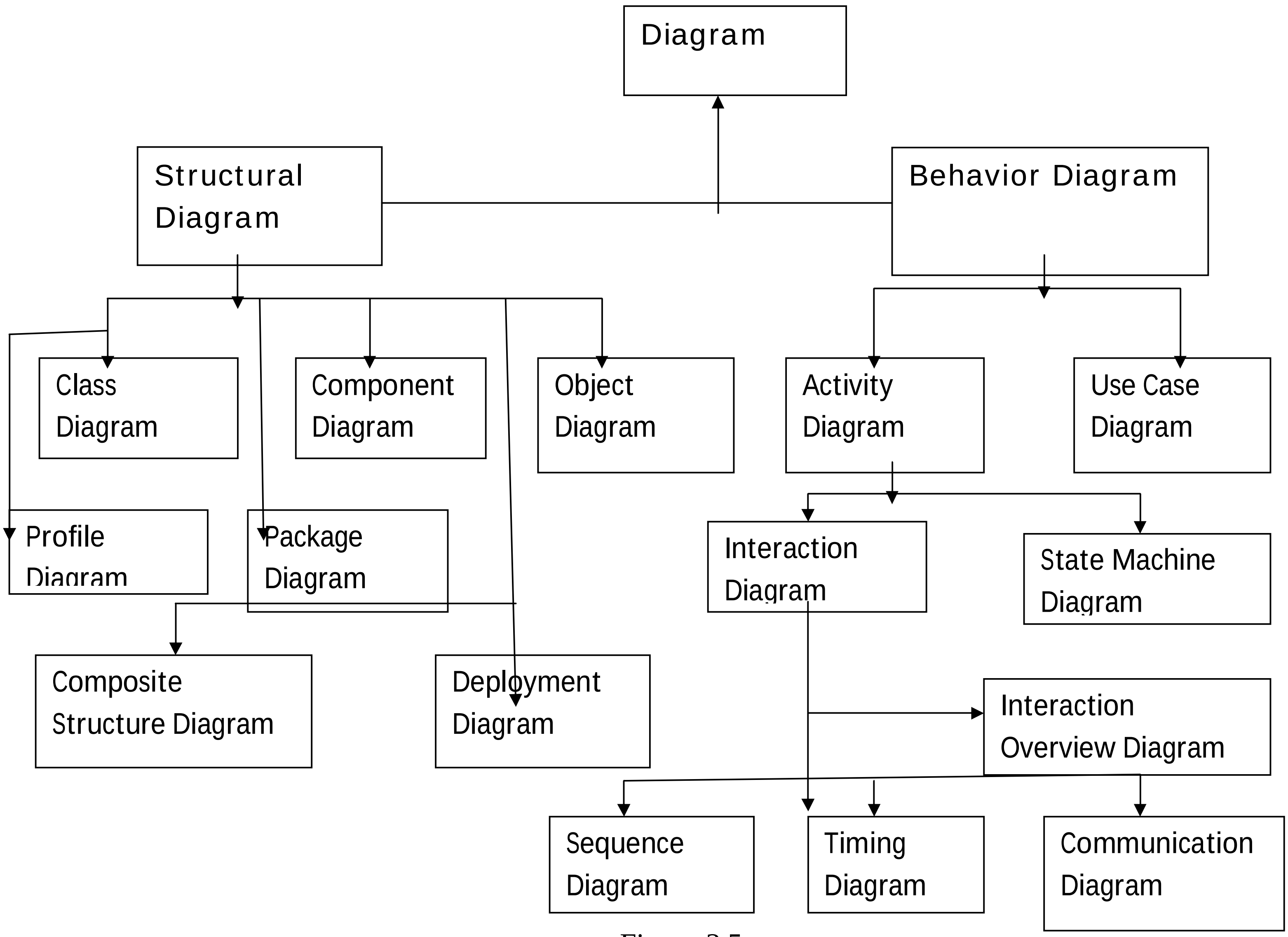


Figure 2.5

2.6 The 2U Proposal: Make Models be Assets.

When models fulfil these definitions they start to be assets themselves. He argues that it is necessary to define the meaning of a model's constructs clarifying what these constructs mean in the context of another model – it is necessary to relate syntaxes. When the UML is to become a family of languages, there must be a limited base of first-class concepts and composition rules on which all those languages build. The UML must be known to be coherent and orthogonal. Layering on top of the core has to be consistent with the meaning of the underlying layer. If these conditions are fulfilled, the UML is executable – but not only in a platform dependent meaning, but in a platform independent one. By defining a Platform Independent Model (PIM) and compiling it with a model compiler it shall be possible to map this PIM to a Platform Specific Model (PSM) and finally execute it on the desired computer platform.

The UML2-Partners (U2P) Proposal: Evolution, not Revolution.

Towering on this approach they propose that necessary changes are a precise definition of the concepts of UML1 and a consolidated semantic foundation through the introduction of the UML infrastructure. Additionally they suggest increased support for multiple standard languages specified by the Meta-Object Facility (MOF), examples are the Common Warehouse Metamodel (CWA) and the Enterprise Application Integration (EAI) model, opening of UML itself to be a family of languages to avoid the "language bloat" syndrome. Finally, the proposal integrates some additional feature improvements like modelling the structure of component-based software systems and more possibilities for modelling complex behaviour and a formal graphical syntax and diagram interchange on top of the current "model element" interchange format XMI.

2.6 The Object Process Methodology (OPM) Proposal: Why significant change is unlikely.

UML is difficult to grasp for a typical developer and slows down the process of software development. Additionally, he presents the Object-Process Methodology (OPM) as a remedy for the situation. In this modelling language, the structure and the behaviour of a system are viewed in the same diagram and so these two most important aspects of OOSE can be viewed at once. Starting from this single diagram other views could be created if necessary.

Additionally, Dori mandates the notion of a “process” concept in UML which is supposed to be a pattern of transformation experienced by one or more objects. He argues that this process concept ought to be at the base of an ontologically correct, system-theoretical foundation for UML. Finally Dori objects the usage of programming jargon in the UML specification as both programmers and the users of a system should be able to view the UML diagrams.

2.7 The 3C Proposal:

The 3C proposal strives to reduce the 144 primitives which are defined by UML to just fifteen types. Using these fifteen6 concepts any other concept of UML can be derived as a non-primitive one. This attempt to clear up the foundations would help to learn, use and automate the UML and to extend its life, so the proposal.

2.7.1 The Executable Unified Modelling Language Profile

Executable UML, as the name implies, is an attempt at making UML directly executable or translatable to 3rd or 4th generation programming languages. UML in its current state is not directly executable.

It contains a hefty number of constructs that can be used in different ways by different organizations. It also lacks action semantics to describe the steps executed by the system in response to events. Executable UML tries mitigating the risk described in the paragraph above, by making the UML artefacts an integral part of the production of systems, not just non-imperative design. Executable UML, coupled with Model-Driven Architecture, may be the foundation for a new software development methodology in which domain (business) experts would be involved in the crafting of high-level models and technologists would be concerned with building model compilers that would translate the models to 3rd or 4th generation code. These model compilers would be created by highly skilled computer specialists using the best practices and advanced knowledge of the computer science and software technology.

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

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

Executable UML is a profile of UML trying to go one step further than the ordinary UML. It enables the developer to declare his intentions down to a more detailed level than UML and so lets him create executable models. The xUML language is a subset of UML on the one hand, and a superset with the additions of an action specification language on the other hand (figure 2.5. With the adoption of UML version 2.0 the situation has changed as the action semantics are now parts of the UML standard. Still, there are other additions in xUML going beyond the scope of the UML standard, especially the precise meaning of a diagram’s elements for another diagram – this interaction specification is left out in UML. The interesting part of xUML is that models created in this language can be compiled into any programming language and afterwards executed on any chosen platform.

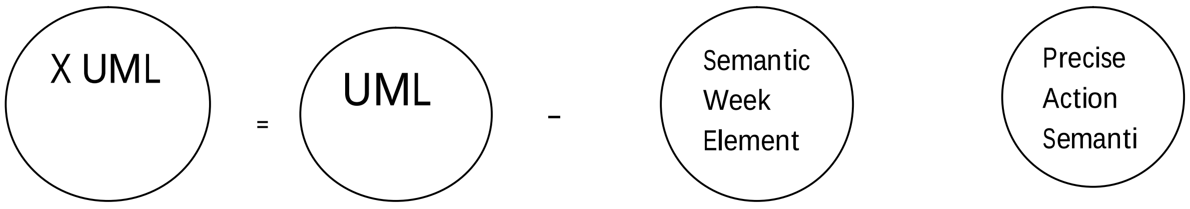


Figure 2.5 XUML

2.7.2 Modelling in Executable UML

Executable UML tries to abstract from both specific programming languages and the decisions about how the software should be organized. An executable UML model can be

• built by graphical model builders,

• verified by interpreting the model with real values,

• compiled to a wide array of platforms,

• debugged to see the model in action,

• analyzed to find paths leading through the model execution and unreachable states

• tested by generating and running test cases for the model

In the following sections, the process of OOAD in xUML shall be shortly outlined, more to this in chapter 3. The explanations are based on the assumption that the reader has knowledge of UML and is therefore especially interested in the differences of the xUML and the UML notation.

2.7.3 Building a System Model

Starting of from the requirements provided for the system, a model is built by identifying the domains being relevant. The construct resembling the notion of a domain in the ordinary UML is a package: it provides a common boundary for objects of a system belonging together.

So the UML package diagram, as provided by figure 2.3, can also be seen as a domain model diagram in xUML: the classes necessary for the metamodel of UML are partitioned in packages providing a rough structure. The process of partitioning the real world in domains which are "...autonomous worlds inhabited by conceptual entities..." is called domain identification.

Along with this building of a domain model, use cases can provide means to understand the functional or behavioural requirements of the system. These constructs closely match use cases in the UML.

When the process of identifying domains and specifying use cases has produced a domain model satisfying the developer - satisfying does not mean perfect here, as there is always the necessity for iteration and coming back to further detail the basics – he can start with modelling any of the domains previously defined.

2.7.4 Modelling a single Domain

Starting from the requirements the next step is to find the abstractions necessary to build the precise model of a domain, capturing the behaviour of the parts it is comprised of.
This model consists of
• Class diagrams,
• State machines and
• Action specifications which will be explained in the following.

2.7.5 Building class diagrams:

In an attempt to abstract from the use cases built earlier to describe structure and behaviour of the domain, it is necessary to search for things which are alike and model them as classes, ignoring the biggest part of these things – those not being necessary for the system.
The next step in capturing classes is finding the properties necessary to describe such classes; each of these properties which is relevant for the problem will be modelled as an attribute of the class. There are two differences to UML in this step worth being noted: the first is xUML not having a visibility attached to an attribute
– it is the choice of the model compiler to transform the attribute appropriately to the destination language. The second thing is the notion of identifiers which are special attributes having a similar role as unique identifiers in the relational data theory with its ER-diagrams: they allow telling a special object - and just this special object - to be told apart from the other instances of a class.
Finally, the relationships between classes have to be established. Again, relationships in xUML are much alike their counterparts in UML, with some exceptions worth to be noted.
First, every relationship in xUML gets a unique name, starting with an R followed by a unique number which is automatically assigned by most tools. Second, xUML misses the notion of navigability – therefore the associations have no arrows, just straight lines (with the exception of the generalization relationship, which looks just like its counterpart in UML). Next, there are no n-ary associations8 in xUML which is a seldom used construct

anyway, reasons for this gives. And finally a very important semantic, but not notational difference: in xUML, an object can change
its class during execution, taking on the behaviour of another class of its generalization tree. This is due to the fact that the generalization is not treated much different from any other relationship - a instance of the class Teacher as a subclass of the class User will have an instance of the class User associated to it, rather than “being” a user itself.
The result of this process is a class diagram much alike to the class diagrams of UML, examples for these diagrams will be given in section.
An additional artefact showing classes interacting with each other is the dynamical collaboration diagram - Mellor suggests that the creation of these diagrams as well as the syntactically equivalent sequence diagrams could (and should) be done automatically, driven by the execution of the model. Possibly different dynamic diagrams can be created depending on the test case momentarily being driven through the system, or diagrams of this type, if used properly and automated, can even be used for real time logging of what happened to the system during its execution.

2.7 State machines:

Finishing the class diagram, the developer should now return to the objects comprising the problem and see if they have a lifecycle that is interesting to be modelled. Here as before, he has to think about lifecycles common to all instances of a class. This lifecycle can be modelled in a state chart diagram which is a way of looking on the class’ actual state machine lying beneath it. State chart diagrams, again, resemble the state chart diagrams of UML. They are
comprised of states, transitions and actions that are executed upon entry to a state.
The interesting part in xUML is that the actions are defined using a predefined language - for example the Action Specification Language (ASL) - which allows to specify the behaviour of the system in detail.

Action Specifications:

The actions being executed in a state machine upon entry into a state are specified using a predefined language. The semantics of this language
- But not its syntax - is standardized as of today. The standardization happened in the official UML version 1.5. With these action specifications the developer can design

precisely the behaviour of the system. The language supports creating and deleting objects, sending signals out to other objects navigate along relationships to the classes interconnected with the current class as well as manipulating these relationships and the reading and writing of attributes. An example for a concrete ASL implementation is given in [WKC+03]. With ASL, the structure of the classes is filled up with the behaviour of its instances – an unavoidable step if executable models are the goal.
When all domains are finally modelled, the view shifts back to the bigger picture and to preparing the model for execution.

Verification and Compilation of the Model

For the verification the development of test sequences is necessary - these sequences define how the model is to be driven through the simulation and eventually what is the correct output at the final stage (using pre- and post conditions). These test cases are a necessity in simulation as well, where the user has more possibilities to interact with the model.

For the compilation the model can be enhanced with further information. This helps optimizing the output and the runtime behaviour of the model on the desired platform.
The process of layering this information on top of the model is called colouring.
Often the execution of the model requires initialization sequences which have to be coded. These initialization sequences are comprised of creating objects which are essential for the execution of the system and of providing the necessary data to these objects.

2.9 The Java Technology

"Java language" redirects here. For the natural language from the Indonesian island of Java, see Javanese language.

Java is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities.

Java applications are typically compiled to byte code (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere". Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java, GNU Class path, and Dalvik.

Java is both a programming language and a programming environment of wide use in the context of heterogeneous and network-wide distributed applications. Java's origin lies in a research project with the goal of developing a small, reliable, portable, distributed and real-time platform for executing applications. At first, C++ was chosen as a programming language, but the goals could not entirely be achieved with this language; as a consequence, a new language was created. Nowadays, both the language and the platform are used to create applications running on a wide variety of hardware platforms, their operating systems and graphical user interfaces.
The portability of the Java language environment makes it also an ideal language for programming and deploying web applications as these applications are highly distributed. The other major characteristics of the Java programming language are that it is simple, object-orientated and familiar; robust and secure; architecture neutral and portable; it offers high performance; it is interpreted, threaded and dynamic. These terms shall be explained subsequently.

Simple, object-oriented and familiar:

Java can be learned in a much shorter time than other programming languages require and it enables the programmer to be productive from the beginning on. Java is object-oriented from the ground up; even the smallest program consists of a Class. Hence, it greatly encourages object oriented development which is necessary in the age of distributed client-server applications. Java very much resembles C++ and is therefore familiar to programmers having knowledge of C++. For them, the migration to the new language is easy.

Robust and Secure:

The Java language environment enables easy creation of reliable software. This fact stems from the new memory management of Java making it unnecessary to deal with pointers anymore. Hence, many errors of C++ programs are eliminated. The compile time checking is also more elaborated and followed by run-time checking. Due to the run-time checking a Java program does not crash without providing information which error occurred at which location in the code. Additionally, Java provides an unprecedented level of security. Security is designed right into the language and the runtime environment, making it hard to breach distributed Java applications.

Architecture neutral and portable:

Java is designed for distributed applications; these applications can run on many different hardware and software platforms. Hence, Java is an interpreted language. The Java compiler creates interoperable byte code which can immediately be executed on a multitude of operating systems. Additionally, the language definition is strict about the basics, for instance the size of data-types and the outcome of mathematical operations.

High performance:

Even though the performance of interpreted languages is lower than the performance of directly executable code, the Java language environment struggles hard to achieve sufficient performance. Java programs are therefore first compiled and later interpreted - a new approach to this topic.

The performance of such programs is not as good as compiled software written in C or C++, but better than of programs written in languages just being interpreted, as Basic and especially its most widely used sibling, Visual Basic.

Interpreted, Threaded and Dynamic:

Java is interpreted and therefore offers very short link phases. The development cycle is fast, aiding in prototyping, experimentation and rapid development. The Java language also offers inherent support for multithreading – many concurrent threads of activity enable the user to do many things at once. Finally, Java offers the concept of dynamic linking. At runtime, new modules can be loaded from any accessible place – the memory, disk drives or the network.

A program written in the Java language is first compiled to a platform independent byte code then being interpreted on the desired platform. This interpretation is done by the Java platform on the target system. The platform consists of two components: the Java VM and the Java API. On top of this foundation, which has been ported to all the major OS currently being on the market, the individual Java program executes.

The greatest advantage of Java is the large amount of APIs being available. They can readily be used by any developer deploying its programs with Java technology.

It comprises essential classes - without them programming would be almost impossible, e.g. the String-class, as well as many useful capabilities such as Graphical User Interface (GUI) components, to be found in the javax.swing-package. This API is one of the most complete and thoroughly defined available.

Chapter 3
Object-Oriented Analysis and Design Phase

3.1 Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterized by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented analysis (OOA) applies object-modeling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on *what* the system does, OOD on *how* the system does it.

3.2 The Process

The process used in this thesis is based on to the Rational Unified Process (RUP). Even though it is termed Rational Unified Process, it is actually a process model and aids
in creating a development process suiting the needs for any special project. Hence, the RUP is described in the following, and whenever small deviations from the RUP occurred, they are explained shortly. For the sake of simplicity, the highly iterative sequence of steps in the RUP will be explained in this section, but not be used in the documentation of the OOAD. Still, emphasis shall be laid upon this major principle of iterating through the software development. The other major components of the RUP are the management of requirements, regarding the architecture of a software system as component-based and visually modelling the structure and behaviour of these components.

3.2.1 Business Modelling: In this early stage, the current business is assessed as it is the environment of the software system in development. For the ATM Machine example, there was no environment to be assessed; this step can be skipped in the development efforts. Additionally, the domain model is built in this stage.

Analysis and Design: This workflow begins with planning the software architecture. For the ATM Machine example, this architecture was decided to be twofold, first, a Java application connecting to a relational database, second, an executable UML model.

Additionally, the use case models are created in this step, as well as the design class diagrams and all the behavioural diagrams like the interaction diagrams. These steps will occur time and again in the current chapter.

Implementation: Here, The emphasis lies upon coding and testing. By generating as much code as possible, the ATM Machine example was developed with low efforts at this stage. The implementation efforts are described in chapter 4.

Test: In this stage, the previously implemented code is tested and verified. Part of the efforts in testing the ATM Machine is explained in this chapter. With xUML, much work is done for this stage in the elaboration iterations, whereas in UML, the testing is delayed to the construction iterations. This is one of the main advantages of xUML.

Deployment: This step copes with deploying the software product to the end-user. This may either be packaging or distributing it, or to provide a web-location for downloading the application. As the ATM Machine example is not deployed to any end-user, this step is not described.

Configuration and Change Management: As the requirements of a software application can seldom be frozen during the development efforts, it is necessary to manage the changes occurring in the needs of the end-users. The configuration management consists of enforcing principles on developers to ensure the seamless access to process artefacts. It also includes the responsibility of providing a sound working environment for each developer. Due to the small type of the ATM Machine example, these steps are not necessary.

Project Management: These responsibilities include all steps necessary for planning the whole project as every single iteration, and the planning necessary if the project has to be changed or if it failed. Additionally, the project maintenance has to be supervised by the project management. Again, due to the small size of the project, this step is not necessary.

Environment: The environment management ensures working tools, prepared templates, and the availability of necessary documents at the beginning of each iteration.

Due to the small size of the ATM Machine example, this step is not described in further detail.

Following the process outlined above will be the topic of the next sections, after a short introduction to the tools being used in the OOAD phase has been given.

3.3 The Analysis and Design Tools

Three different tools were used in the OOAD phase, two for the work with UML and one for working with xUML. As xUML is a profile of UML, it is possible to design xUML models with UML tools, but they usually lack the possibility to generate code directly from the state chart diagrams. Rational Rose 2002 was very much used in the OOA and is introduced first, followed by StarUML , mainly used in the OOD. Finally, the xUML tool iUML of Kennedy Carter is presented.

Rational Rose- First released in 1985, the Rational Environment was an integrated development environment for the Ada programming language, which provided good support for abstraction through strong typing. Its goal was to provide the productivity benefits associated with academic single-user programming environments to teams of developers developing mission-critical applications that could execute on a range of computing platforms.

The Rational Environment was organized around a persistent intermediate representation (DIANA), providing users with syntactic and semantic completion, incremental compilation, and integrated configuration management and version control. To overcome a conflict between strong typing and iterative development that produced recompilation times proportional to system size rather than size-of-change, the Rational Environment supported the definition of subsystems with explicit architectural imports and exports; this mechanism later proved useful in protecting application architectures from inadvertent degradation. The Environment's Command Window mechanism made it easy to directly invoke Ada functions and procedures, which encouraged developer-driven unit testing.

The Rational Environment ran on custom hardware, the Rational R1000, which implemented a high-level architecture optimized for execution of Ada programs in general and the Rational Environment in particular. The horizontally-micro programmed

R1000 provided two independent 64-bit data paths, permitting simultaneous computation and type checking. Memory was organized as a single-level store; a 64-bit virtual address presented to the memory system either immediately returned data, or triggered a page fault handled by the processor's microcode.

The company's name was later changed from "Rational Machines" to **Rational** to avoid emphasizing this proprietary hardware.

Rational provided code generators and the cross-debuggers for then-popular instruction set architectures such as the VAX, Motorola 68000, and x86; much of this was accomplished through a partnership with Tartan Labs, founded by Bill Wulf to commercialize his work on optimizing code generators semi-automatically produced from architecture descriptions (PQCC)

StarUML

StarUML is an open source project to develop fast, flexible, extensible, featureful, and freely-available UML/MDA platform running on Win32 platform. The goal of the StarUML project is to build a software modeling tool and also platform that is a compelling replacement of commercial UML tools such as Rational Rose, Together and so on.

- **UML 2.0**: UML is continuously expanding standard managed by OMG (Object Management Group). Recently, UML 2.0 is released and StarUML support UML 2.0 and will support latest UML standard.
- **MDA (Model Driven Architecture)**: MDA is a new technology introduced by OMG. To get advantages of MDA, software modeling tool should support many customization variables. StarUML is designed to support MDA and provides many customization variables like as UML profile, Approach, Model Framework, NX(notation extension), MDA code and document template and so on. They will help you fitting tool into your organizational cultures, processes, and projects.
- **Plug-in Architecture**: Many users require more and more functionalities to software modeling tools. To meet the requirements, the tool must have well-

defined plug-in platform. StarUML provides simple and powerful plug-in architecture so anyone can develop plug-in modules in COM-compatible languages (C++, Delphi, C#, VB, ...)

- **Usability**: Usability is most important issue in software development. StarUML is implemented to provide many user-friend features such as Quick dialog, Keyboard manipulation, Diagram overview, etc.

StarUML is mostly written in Delphi. However, StarUML is *multi-lingual project* and not tied to specific programming language, so any programming languages can be used to develop StarUML. (for example, C/C++, Java, Visual Basic, Delphi, JScript, VBScript, C#, VB.NET, ...)

iUML

iUML is provided by Kennedy Carter, a consultancy specialized in developing software systems by designing executable models. The modelling notation used by the tool is the xUML profile, both a subset and a superset of the UML standard. This ambiguity is reflected in the notation; xUML diagrams look familiar, but are not similar to UML diagrams. Another speciality of xUML tools is that usually a process is imposed on the developer as he has to follow the steps outlined by the program.
The usability of the product is high, this even more so, as the superimposed sequence of steps aids in achieving first results very fast. The modelling is straightforward, coding for state chart diagrams is done in a special language, this being the Action Specification Language (ASL) for iUML. The development environment for coding the states is a plain text editor, not even providing keyword highlighting; more effort could have been put into this aspect. Code generation is done from state diagrams using a model compiler; any target language supported by this model compiler is possible. Code generation from interaction diagrams is not supported. The tool uses a proprietary repository, many developers can work on this repository at once, the changes are then subsequently submitted and conflicts handled.

3.4 The Requirements:

In the case of the Examination System, the requirements are purely functional, for larger development efforts, the requirements can come from a wide variety of categories.
The categories proposed in the RUP divide the requirements based on what aspects they affect, the following incomplete list provides some of these categories.
Functional requirements deal with features, capabilities and security **Usability requirements** declare human factors, needed help and documentation.
Reliability requirements impose restrictions on frequency of failure, recoverability and predictability.
Performance requirements deal with response times, throughput, accuracy and further aspects.
Supportability requirements explain the needs of adaptability, maintainability, internationalization and configurability.
Additional categories could deal with the restriction to special tools, languages or hardware, with interfaces to other systems or legal aspects.

3.4.1 Requirement of Resultant Tool:

Code to diagram: Resultant tool should draw different uml diagram from piece of object oriented code.
UML Diagram to Java code: Tool should also convert the class diagram sequence diagram and use case diagram to Skelton of code.
Draw UML diagram: Tool will draw different uml diagram manually. UML diagram consists use case diagram sequence diagram class diagram collaboration diagram etc.
Customized Elements: Tool will have special quality for advance user to draw different uml element of their own choices.
Zoom In Zoom Out: User can change the size of diagram. Tool will provide zoon in zoom out facility.
Non Functional Requirements: Non functional requirements means safety, usability, maintainability, Correctness, security, consistency, simplicity etc.
Correctness: Resultant diagram should correct. Diagram Skelton, code Skelton should have consistency. Class diagram, sequence diagram should also have consistency.

Usability: Tool will be user friendly. Any known user of java can use this tool for customization of elements.

3.5 The Object-Oriented Analysis and Design Phase

Starting off from the requirements, the OOAD progresses by first executing a thorough analysis of the requirements and the real world being the inspiration for them and second designing structure and behaviour of an appropriate software system.

The Object-Oriented Analysis:

The requirements given above are strictly spoken no such use cases – however, they can be used to perform the analysis normally based on use cases.

Identifying and describing Use Cases

To properly identify use cases a possibility is to start off from the requirements and to do a linguistic analysis. Searching for use cases is equivalent to search for verb phrases in the requirements. The easiest way to do this is to go through the requirements and highlight all the verb phrases with a text marker, as shown in Block Letters.

The Requirements of Examination Problem

Examination Problem **aids in perfecting** the mental arithmetic of elementary school students. Examination Problem **poses** each student ten random arithmetical exercises, which **should be solved** as fast and correct as possible. From the responses **scores are collected** which **can be viewed** by the users of Examination Problem. Teachers **can define** types of exercises by **determining** numerical ranges and allowed mathematical operations. They **can also delete** types which **were defined** by themselves. Students **are assigned** to their teacher and **can request** exercises for an exercise type of their teacher.

New teachers and students **are able to apply** as new Examination Problem users themselves by specifying username and password – this **is done** in the context of the usual user identification. The password **can be changed** anytime.
Teachers **can delete** the students which **are assigned** to them.

Checking verbs

aids in perfecting	This is a mission statement - not a use case, it is too high in its goal to be one.
poses	Here the first use case candidate is found - Examination Problem poses student arithmetical exercises.
Should be solved	The student solves exercises – this seems to be very close to the previous use case, so just one of these two use cases is necessary
scores are collected	The system creates scores from the responses. If the system is doing some internal work, this work should not be reported as a use case.
can be viewed	The A user can view scores.
can define	Teachers can define exercise types.
are assigned	A student is assigned to a teacher.
can request	A student can request exercises.
is done	This is a statement about where the above use case has to happen.
are assigned	A user can change the password anytime.
can delete	A teacher can delete his students.
can be stopped	The system stops the time for each exercise. Again, reporting this fact as a use case would not be in compliance with the black box view.

Table 3.1

Implicitly, this use case is mentioned in the requirements as they proclaim any person being able to apply as a new user in the context of the login procedure.
Additionally, not all verbal phrases are describing a standalone use case as could be observed in coping with the information of table 3.1. Some of the identified facts are constraints, some are further refining other use cases and some are system internal actions which should not be considered in this phase (for considering what the notion
“System internal” means, the system boundaries have to be defined correctly).
As for the naming of the use cases, a good advice is to use names starting of with a verb. This is to emphasize the dynamical aspect of a use case and to distinguish them from the other concepts of OOA. The found use cases are described in written stories. This is not the only way to represent use cases thoroughly and with added value for the user, another way would be to draw use case diagrams (to look at the system from a high-level view) and describe the internal actions of the use cases with activity diagrams.

The use case diagram for the Examination Problem system is shown in figure 3.1.
Use case diagrams show the identified use cases in ovals and the interactions between actors and use cases by arrows pointing from an actor to the associated use case. The actors of the use case diagram should be termed different to the names of the classes later used in the structural diagrams, this is why in figure 3.1 the term “Actor” is used as a prefix to the actor’s names.
Modelling the use cases is one of the first steps of the OOAD process – and therefore the use case diagram should apparently be one of the easiest diagrams to create and understand by both developer and user.

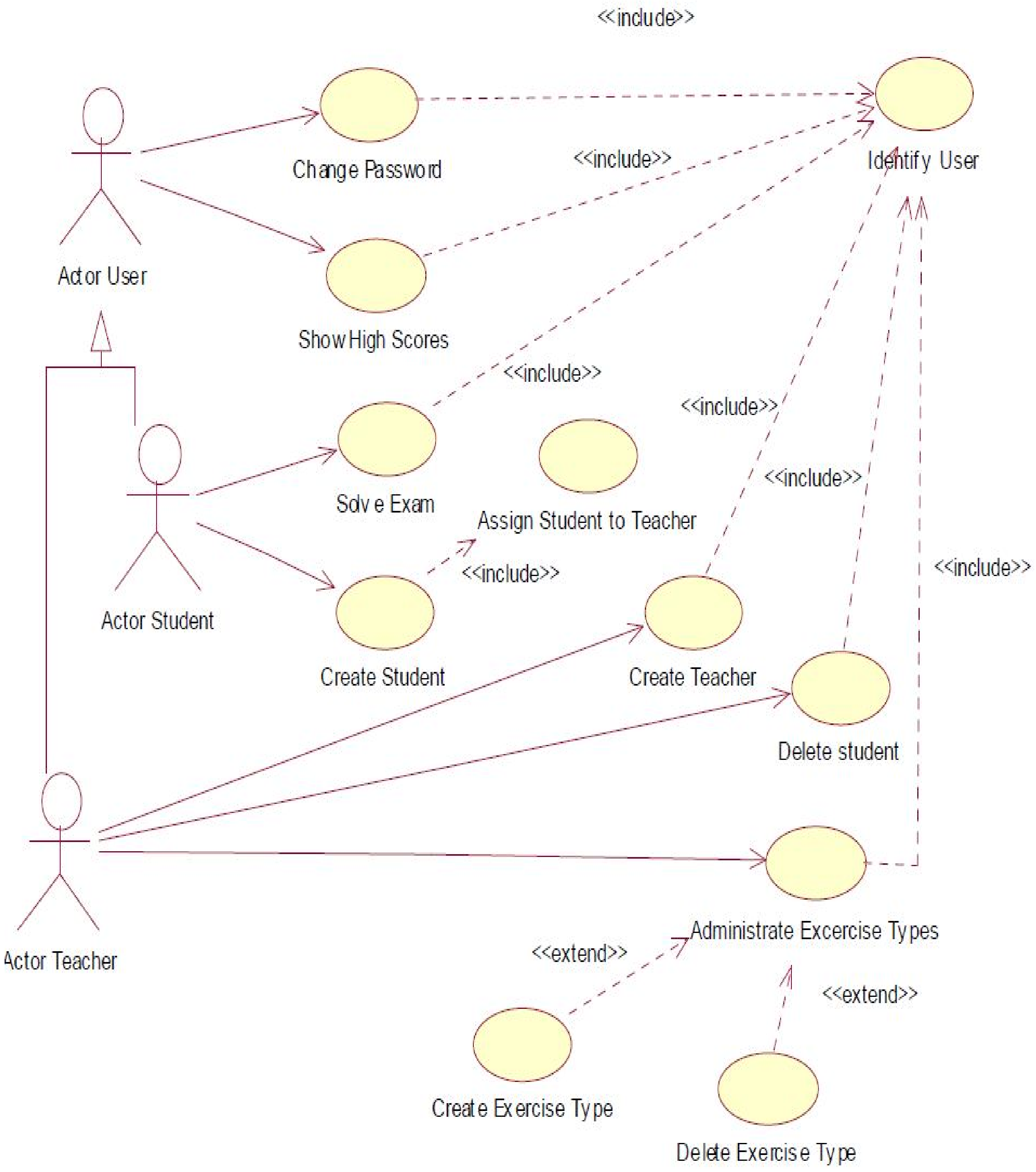


Figure 3.1 use case diagram

3.6 Building a Domain Model

When the use cases and the other requirements have been identified, the next step is to build a domain model – which is a model of objects of the real world, and not a software model yet. This domain model must include all the relevant conceptual classes, which are again found by linguistic analysis.

The preferred way to find classes is to highlight and identify nouns in the requirements, as can be seen in figure 3.5. After a noun has occurred the first time, it is not marked a second time, contradictory to the handling of the verb phrases.

Again, after the nouns are identified and highlighted, an analysis has to be done about which nouns constitute conceptual classes and which do not. This process will be shown in table 3.3. Again, after the nouns are identified and highlighted, an analysis has to be done about which nouns constitute conceptual classes and which do not. This process will be shown in table 3.3.

When looking at table 3.3, the question arising most frequently in modelling this example is if a given noun should be displayed as a stand-alone class or as an attribute to another one.

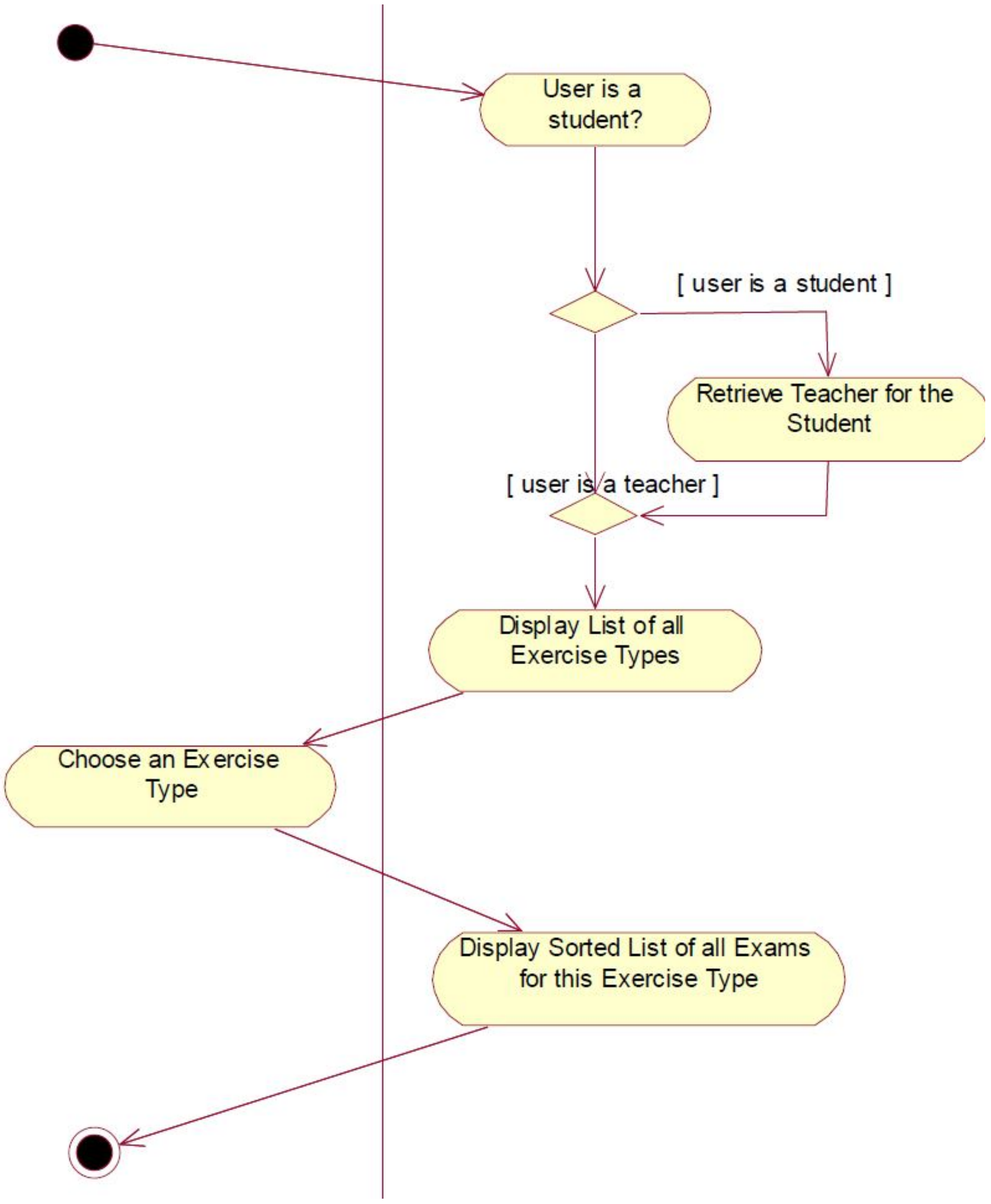

User is a student?
[user is a student]
Retrieve Teacher for the Student
[user is a teacher]
Display List of all Exercise Types
Choose an Exercise Type
Display Sorted List of all Exams for this Exercise Type

Figure 3.2 Domain use case model

Description	A student can be created by any user of the Examination Problem system. Necessary information are username and password which has to be entered twice.
Precondition	A student can be created by any user of the Examination Problem system. Necessary information are username and password which has to be entered twice.
Postcondition	A new student has been created. The student possesses a unique username and a password and is associated to an existing teacher.
Error in condition	1. The provided username is already given to a user of the Examination Problem system. 2. The entered passwords do not match
Error postcondition	The student was not created.
Actor	User

Table 3.2

Tooling - the transition from OOA to OOD: When using Rational Rose for the OOA, the major problems arising in this step can be dealt with – without a problem whatsoever. The interesting step is the transition into the OOD phase. Arguably, it is best to keep the model of the analysis separated from the design model as the classes of the domain model (the conceptual classes) are clearly different from the classes in the design model which are the design classes. If the step of separation is not taken and the classes of the design model are changed - providing that the same classes are used in both design and domain model - the changes show up in the domain model as well. These changes might be deleting or adding attributes and deleting or adding associations.

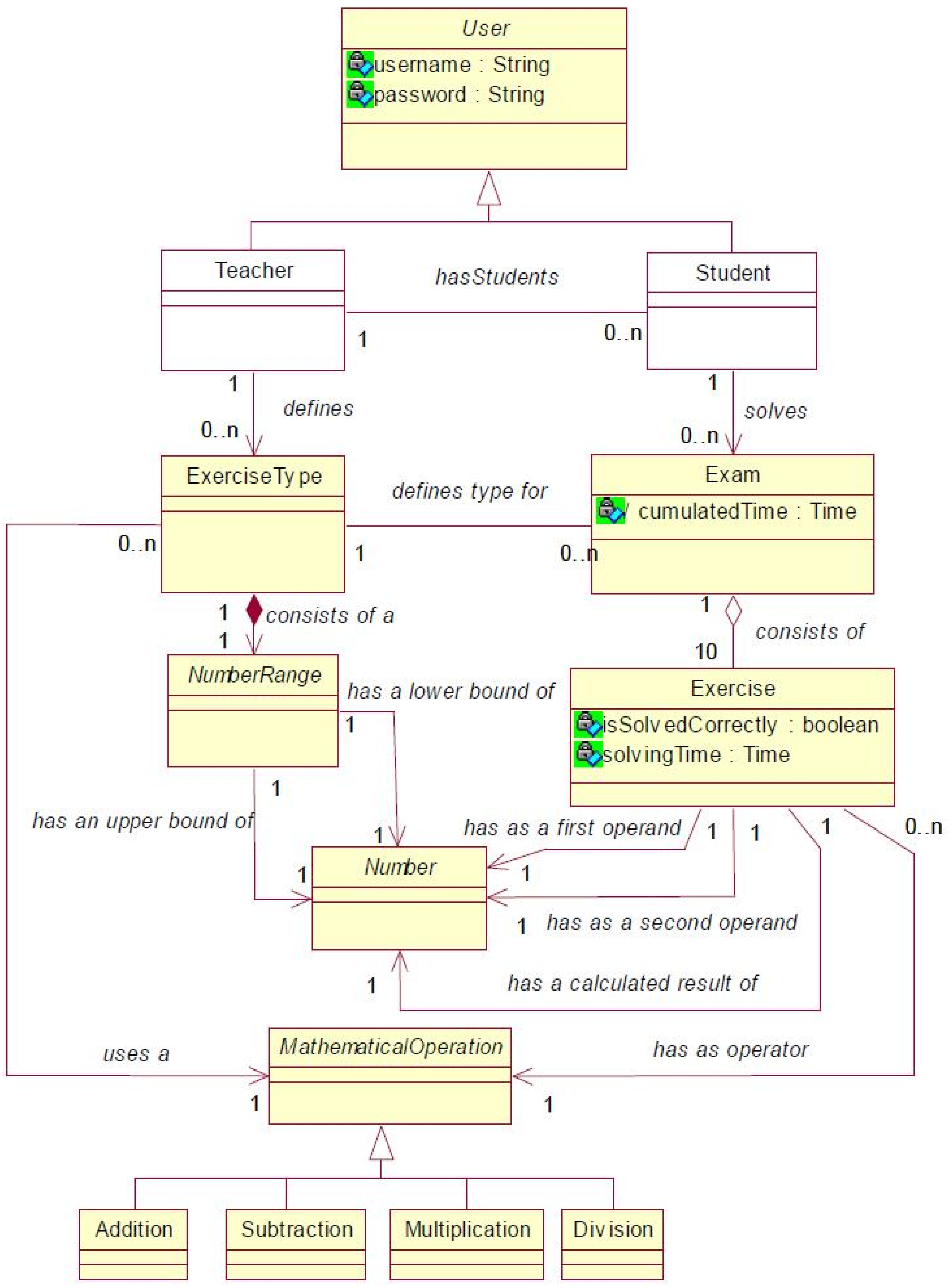


Figure 3.3 class diagram

3.7 The Object-Oriented Design

The OOD phase bases on the outcome of the OOA, the transition not being easy between these two steps ([Kai99]). The transition is mainly based on the domain model which is subsequently transformed to (or replaced by) a design class model. In the explanation of this artefact of the OOAD phase the transition will be exemplified on some of the interesting aspects of the Examination Problem. Complementary to the design class diagrams, the interaction diagrams are the other main model being built in the OOD phase.

Patterns

Patterns are solutions to problems very often occurring in the software development process. They can be used both in the interaction diagrams as in the design class diagrams to help solving problems frequently solved before

The Entity-Boundary-Controller Pattern:

The concept of entity, boundary and controller classes is used to model the interaction between objects and to separate sequencing, data and logic and the visualization of an object4. These concepts could already be used in the

analysis phase, as explained in, but to keep design decisions out of this first phase and display only the interesting information, it might be better to defer this classification of classes to the design phase, especially as many classes can be created in this step which are not abstractions of classes in the real world.

For the Examination Problem example, the modified approach was used. Figure 3.8 shows the static structure of the modified EBC pattern. From the structural point of view, this pattern is very easy to comprehend: there is a controller element which has two relationships with an entity element. One is an instantiation relationship, and the other an association named administrates <EntityElementName>. Additionally, it owns one boundary element having a link to the entity element itself. The link is only established after the controller object has passed the entity object to the boundary object, before the boundary object has no information about its existence. First the controller element instantiates the entity, then it instantiates the boundary and passes as creation data the entity element to the boundary (which now has a link to the entity).

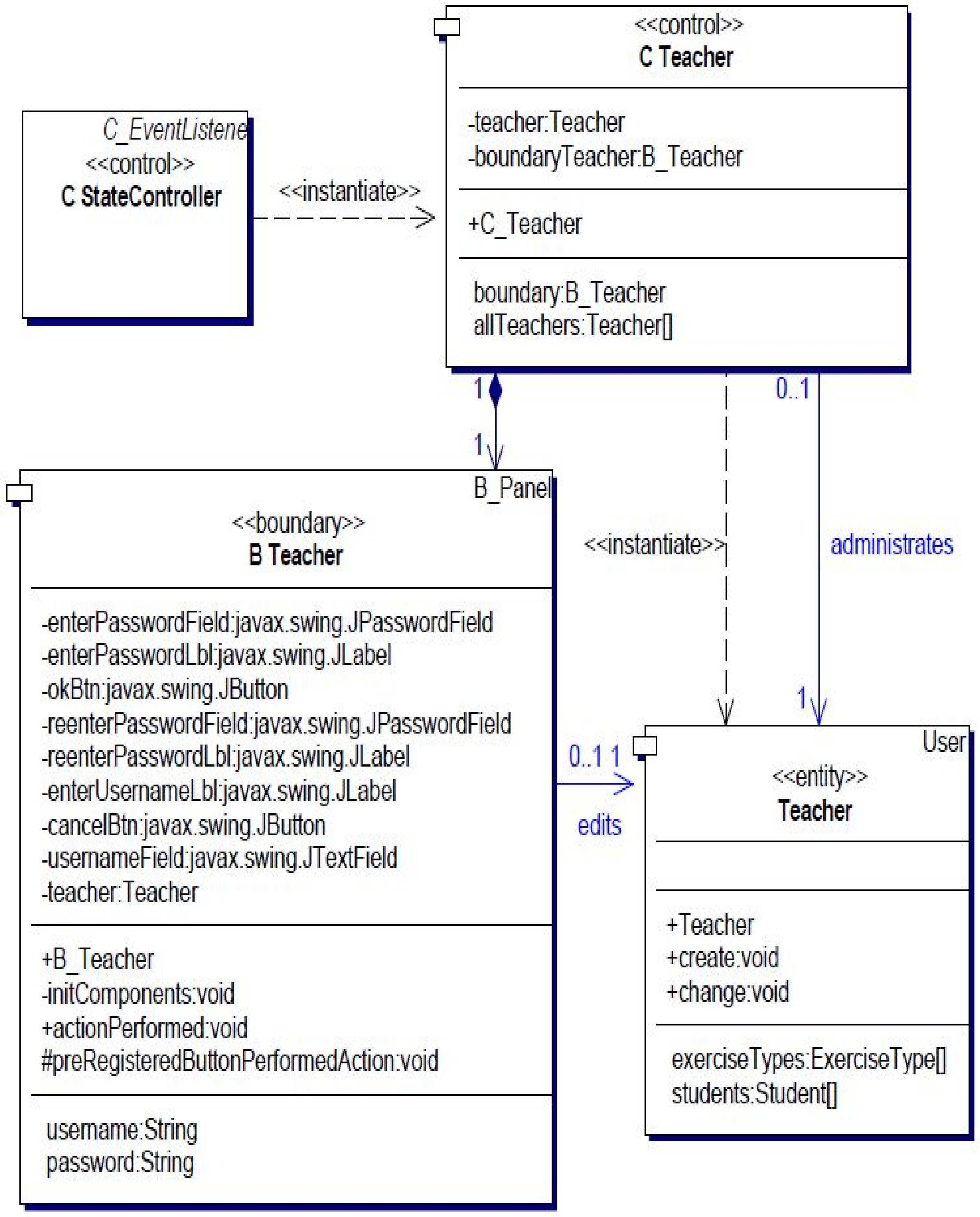

C_EventListene
<<control>>
C StateController
<<instantiate>>
<<control>>
C Teacher
-teacher:Teacher
-boundaryTeacher:B_Teacher
+C_Teacher
boundary:B_Teacher
allTeachers:Teacher[]
1
1
0..1
B_Panel
<<boundary>>
B Teacher
-enterPasswordField:javax.swing.JPasswordField
-enterPasswordLbl:javax.swing.JLabel
-okBtn:javax.swing.JButton
-reenterPasswordField:javax.swing.JPasswordField
-reenterPasswordLbl:javax.swing.JLabel
-enterUsernameLbl:javax.swing.JLabel
-cancelBtn:javax.swing.JButton
-usernameField:javax.swing.JTextField
-teacher:Teacher
+B_Teacher
-initComponents:void
+actionPerformed:void
#preRegisteredButtonPerformedAction:void
username:String
password:String
<<instantiate>>
administrates
1
0..1 1
edits
User
<<entity>>
Teacher
+Teacher
+create:void
+change:void
exerciseTypes:ExerciseType[]
students:Student[]

Figure 3.4 domain class diagram

The General Responsibility Assignment Pattern: Pattern usage in OOP as explained in section 2.5 does not start with programming – some patterns are a vital tool also in the design phase, the GRASP collection of patterns is one of these, introduce. GRASP means General Responsibility Assignment Patterns and consists of the patterns aiding a developer in handling this important task. In these patterns, responsibilities are assigned to components based on widely used design principles. These patterns can be used when designing interaction diagrams: whenever a message has to be sent from an object to another object the developer has to reflect about where to put the responsibilities of receiving and sending the message. Often, the part of sending is provided by the structure and the use case.

Objects (of class A) have to be created, and this responsibility is assigned to a class B, if the objects of this class B aggregate, contain, record or closely use instances of class A. This pattern can also be applied when B possesses the information an object of class A needs for its construction. The list is prioritized, so when more than one relationship of classes to class A exist,

the classes having relations that stand on top of the list are chosen first for being creator.

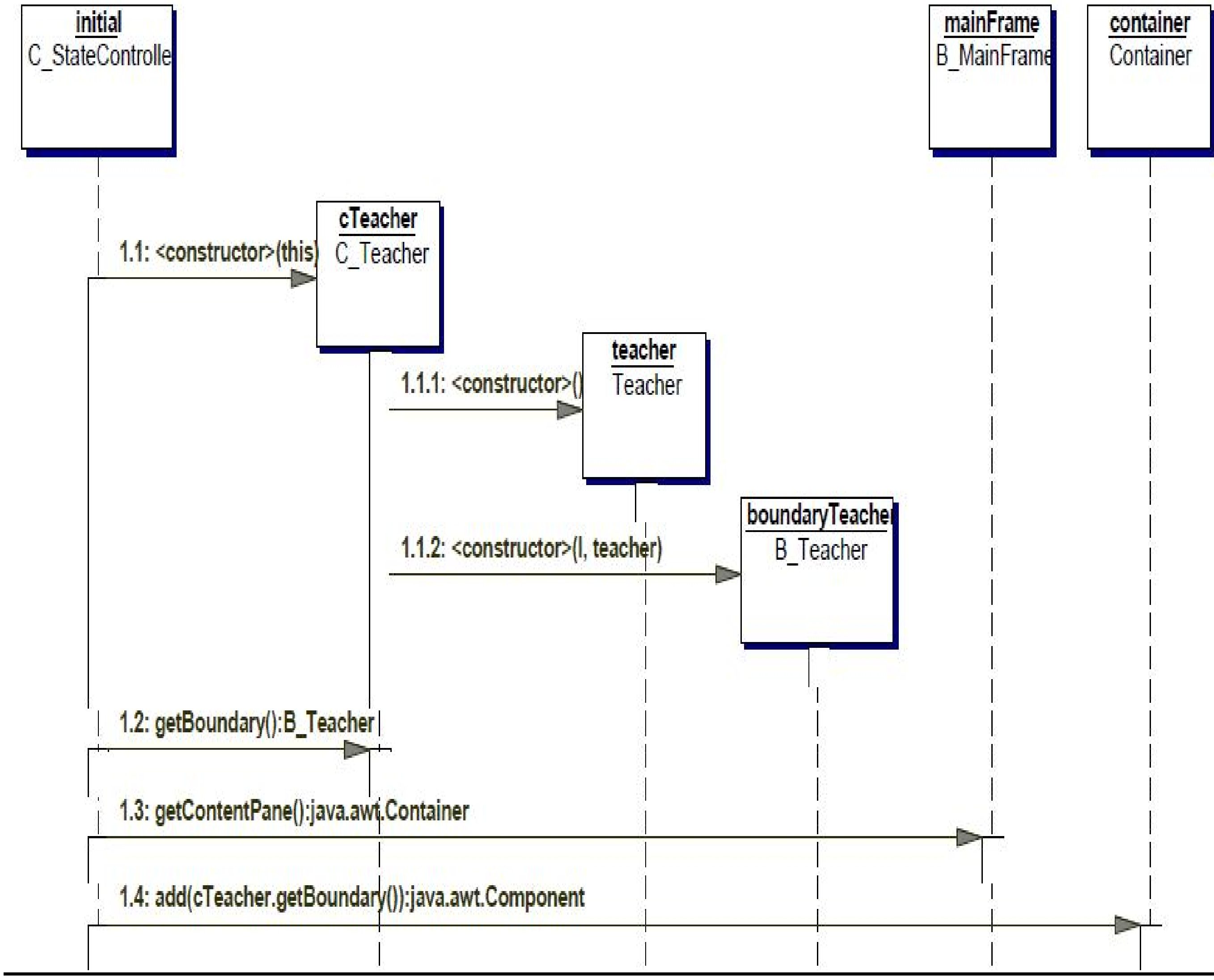


Figure 3.5 Sequence diagram

3.7 The Object-Oriented Analysis and Design Phase using Executable UML

The Domain Model in xUML

The domain model of the Examination Problem example is shown in figure 3.4; it displays the interdependencies of the domains. In xUML, the term "domain" is used for a part of the system in development, this part must have clearly defined boundaries. Hence, these elements are then analysed and designed separately and interfaces are envisaged to finally interconnect them.

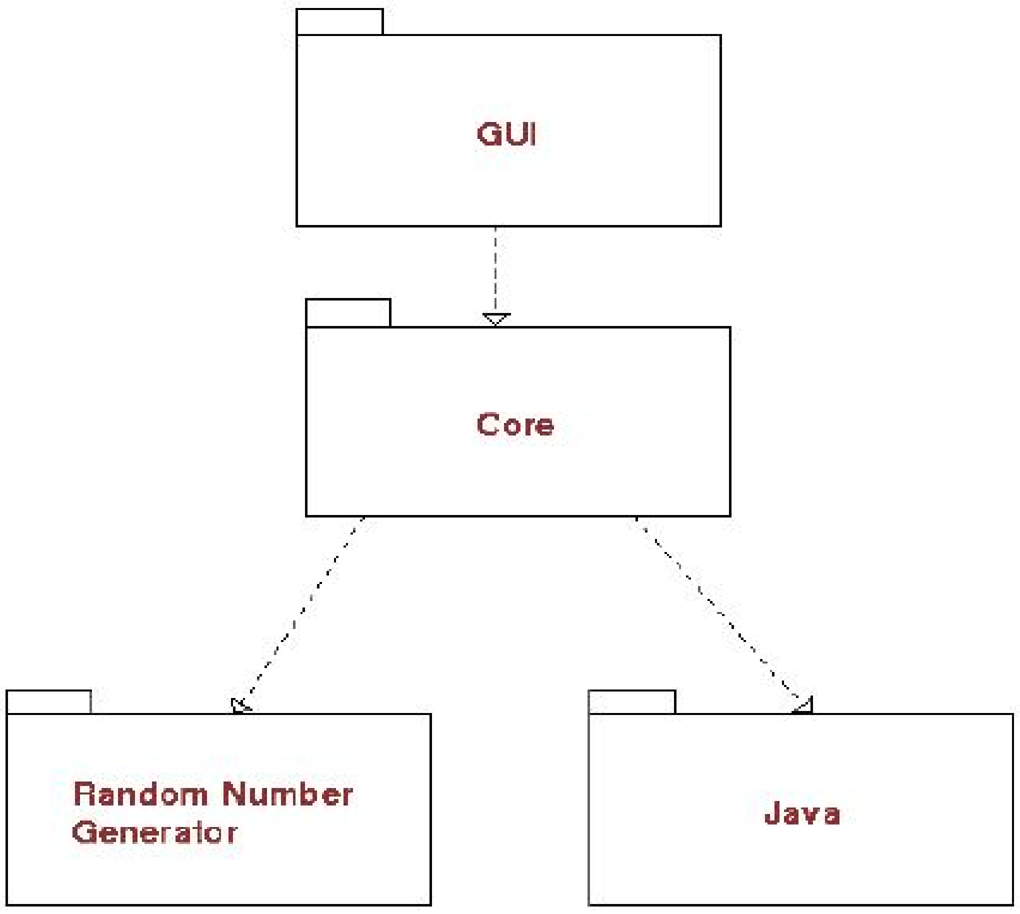
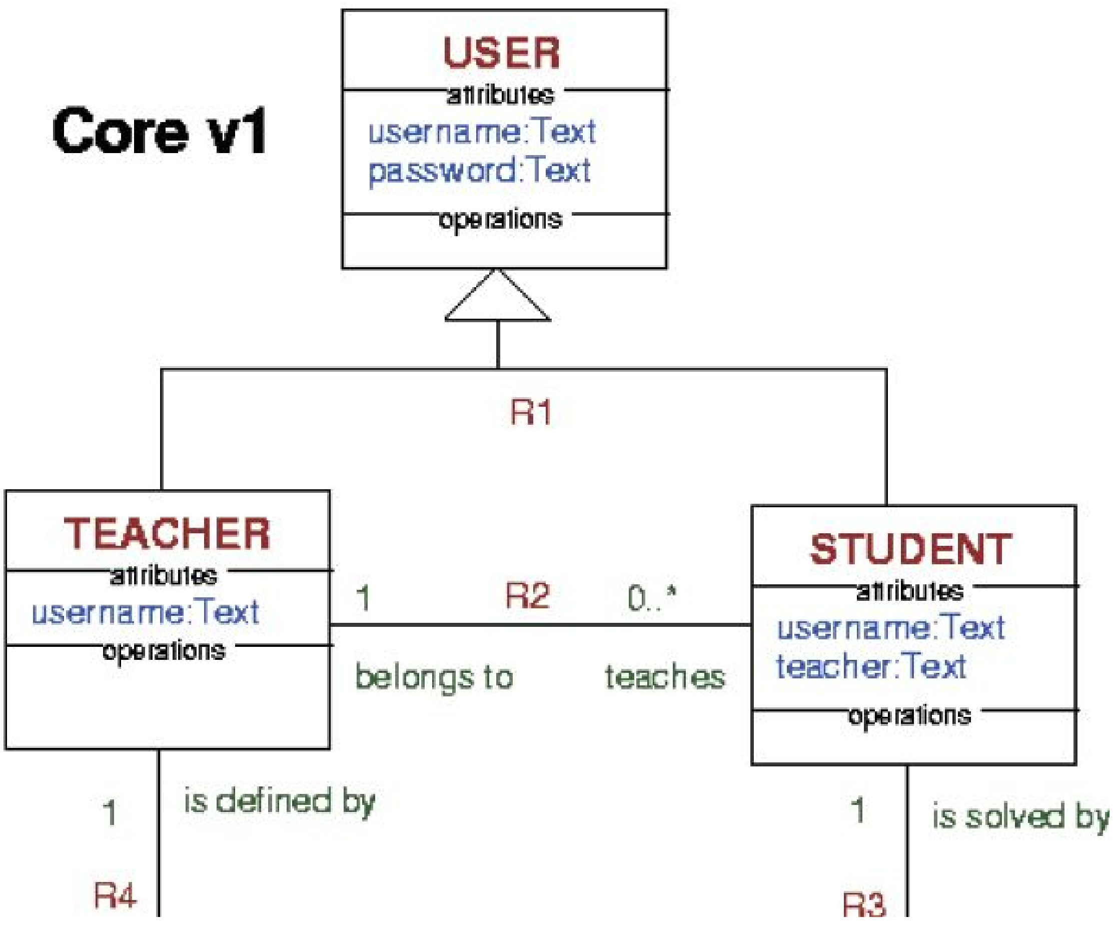


Figure 3.6 Domain model in XUML

Class Diagrams in Executable UML

Class diagrams are used to depict the structure of the real-world. More specifically, it shows the structure in the part of the real world being base for the software system under construction. The reasoning applied to find the elements of the UML design model and design class diagram can also be used in the case of xUML. The result of applying this reasoning, the xUML class diagram of the Examination Problem example, is shown in figure 3.6.



Core v1
USER
attributes
username:Text
password:Text
operations
R1
TEACHER
attributes
username:Text
operations
1
R2
0..*
belongs to
teaches
STUDENT
attributes
username:Text
teacher:Text
operations
1
is defined by
R4
1
is solved by
R3

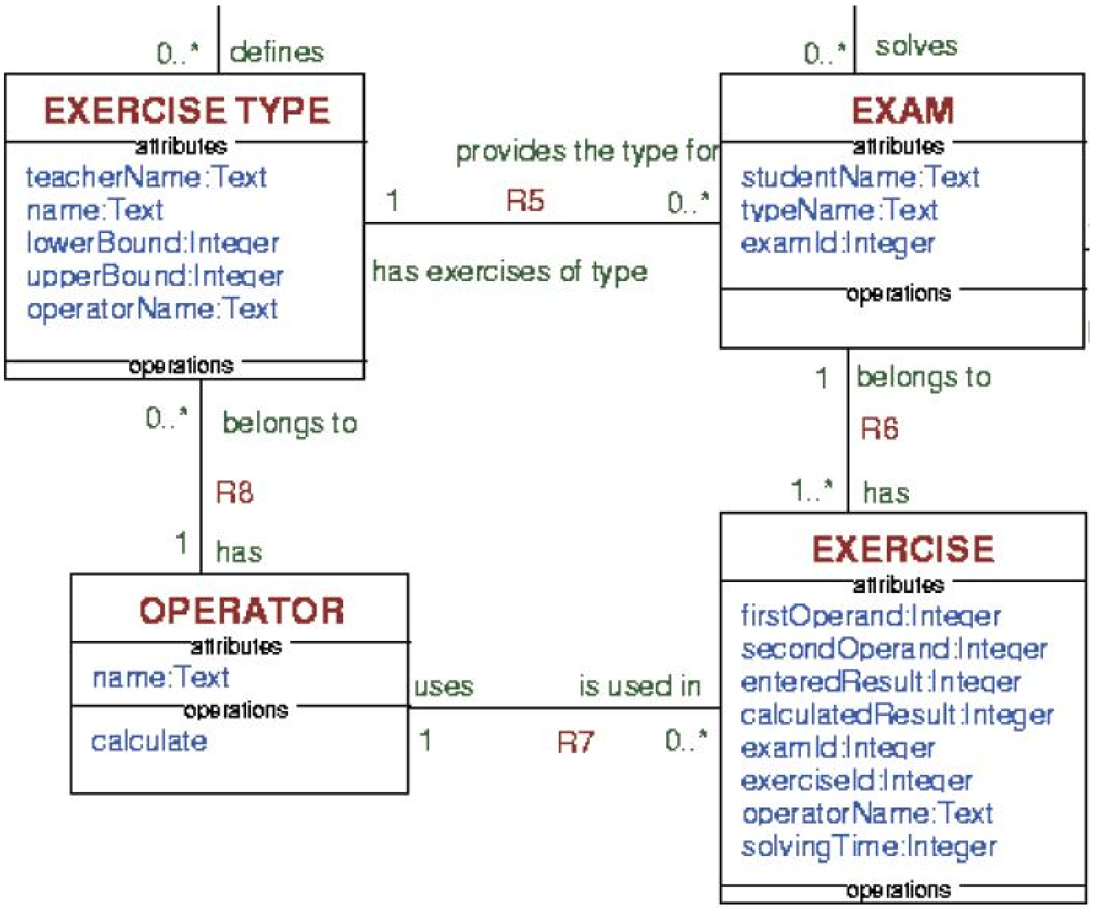


Figure 3.6 Domain class diagram in xuml

Chapter 4

Implementation

4.1 Working with Diagrams:

“Before we explore the details of UML, it would be wise to talk about when and why we use it. Why do engineers build models? Why do aerospace engineers build models of aircraft?

Why do structural engineers build models of bridges? What purposes do these models serve?

These engineers build models to find out if their designs will work. Aerospace engineers build models of aircraft and then put them into wind tunnels to see if they will fly.

Structural engineers build models of bridges to see if they will stand. Architects build models of buildings to see if their clients will like the way they look. *Models are built to find out if something will work*. This implies that models must be testable. It does no good to build a model if there are no criteria you can apply to that model in order to test it. If you can’t evaluate the model, the model has no value. The harm has been done to software projects through the misuse and overuse of UML.

Why build models of software?

Can a UML diagram be tested? Is it much cheaper to create and test than the software it represents? In both cases the answer is nowhere near as clear as it is for aerospace engineers and structural engineers. There are no firm criteria for testing a UML diagram. We can look at it, and evaluate it, and apply principles and patterns to it; but in the end the evaluation is still very subjective. UML diagrams are less expensive to draw than software is to write; but not by a huge factor. Indeed, there are times when it’s easier to change source code than it is to change a diagram. So then, does it make sense to use UML?

I wouldn’t be writing this book if UML didn’t make sense to use. However, the above illustrates just how easy UML is to misuse. *We make use of UML when we have something definitive we need to test, and when using UML to test it is cheaper than using code to test it.*

For example, let’s say I have an idea for a certain design. I need to test whether the other developers on my team think it is a good idea. So I write a UML diagram on the whiteboard and ask my teammates for their feedback.

Communicating with Others:

UML is enormously convenient for communicating design concepts between software developers. A lot can be done with a small group of developers at a whiteboard. If you have some ideas that you need to communicate to others, UML can be a big benefit.

UML is very good for communicating focussed design ideas. For example, the diagram in Figure 4.1 is very clear. We see the LoginServlet implementing the Servlet interface and using the UserDatabase. Apparently the classes HTTPRequest and HTTPResponse are needed by the LoginServlet..

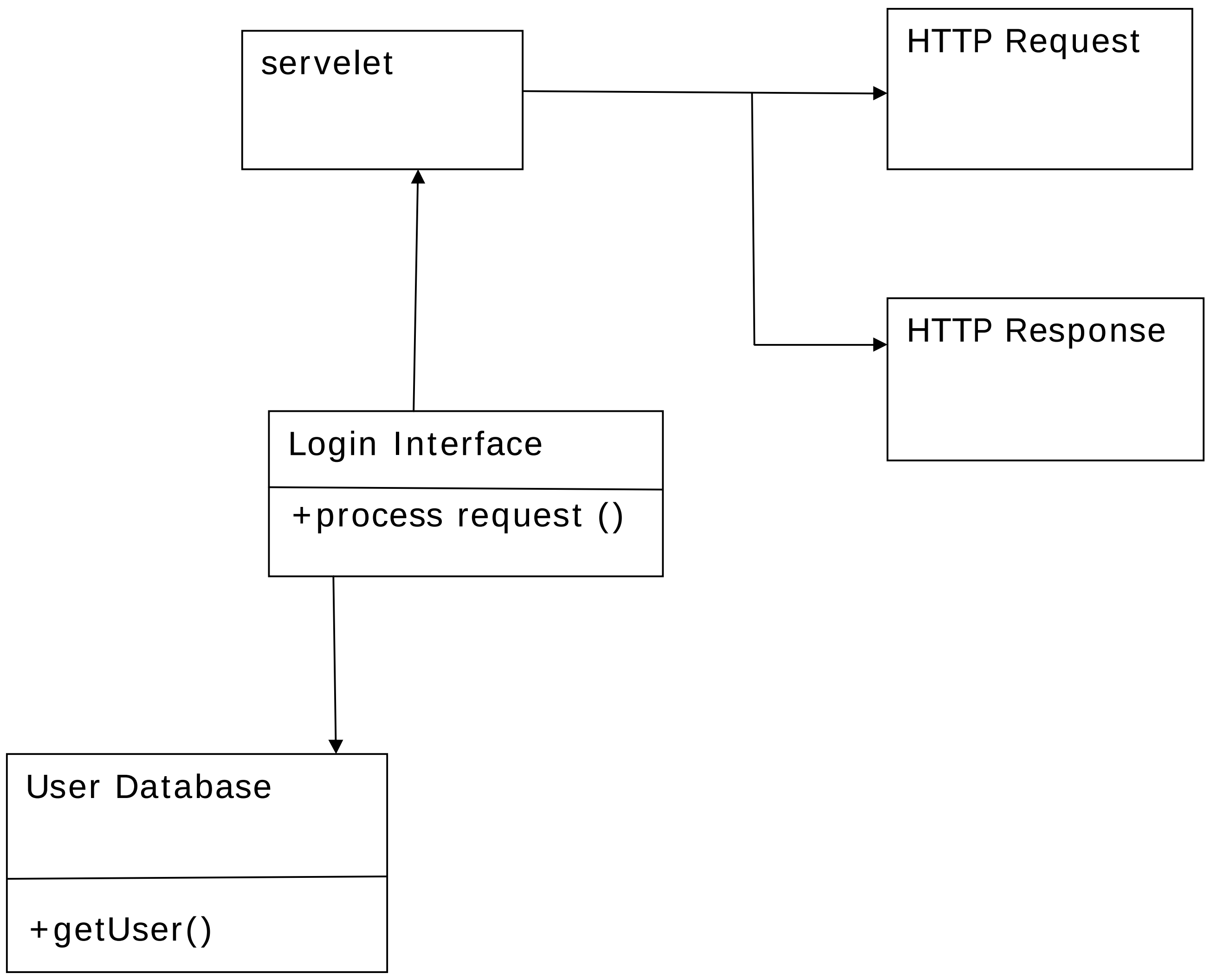


Figure 4.1

On the other hand, UML is not particularly good for communicating algorithmic detail. Consider the simple bubble sort code in Listing 2-1. Expressing this simple module in UML is not very satisfying..

Bubble Sort

```
public class BubbleSorter
{
static int operations = 0;
public static int sort(int [] array)
{
operations = 0;
if (array.length <= 1)
return operations;
for (int nextToLast = array.length-2;
nextToLast >= 0; nextToLast--)
for (int index = 0; index <= nextToLast; index++)
compareAndSwap(array, index);
return operations;
}
private static void swap(int[] array, int index)
{
int temp = array[index];
array[index] = array[index+1];
array[index+1] = temp;
}
private static void compareAndSwap(int[] array, int index)
{
if (array[index] > array[index+1])
swap(array, index);
operations++;
}
```

The diagram in Figure 4.1 gives us a rough structure, but is cumbersome and reflects none of the interesting details. The diagram in Figure 4.3 is no easier to read than the code and is substantially more difficult to create. UML for these purposes leaves much to be desired.

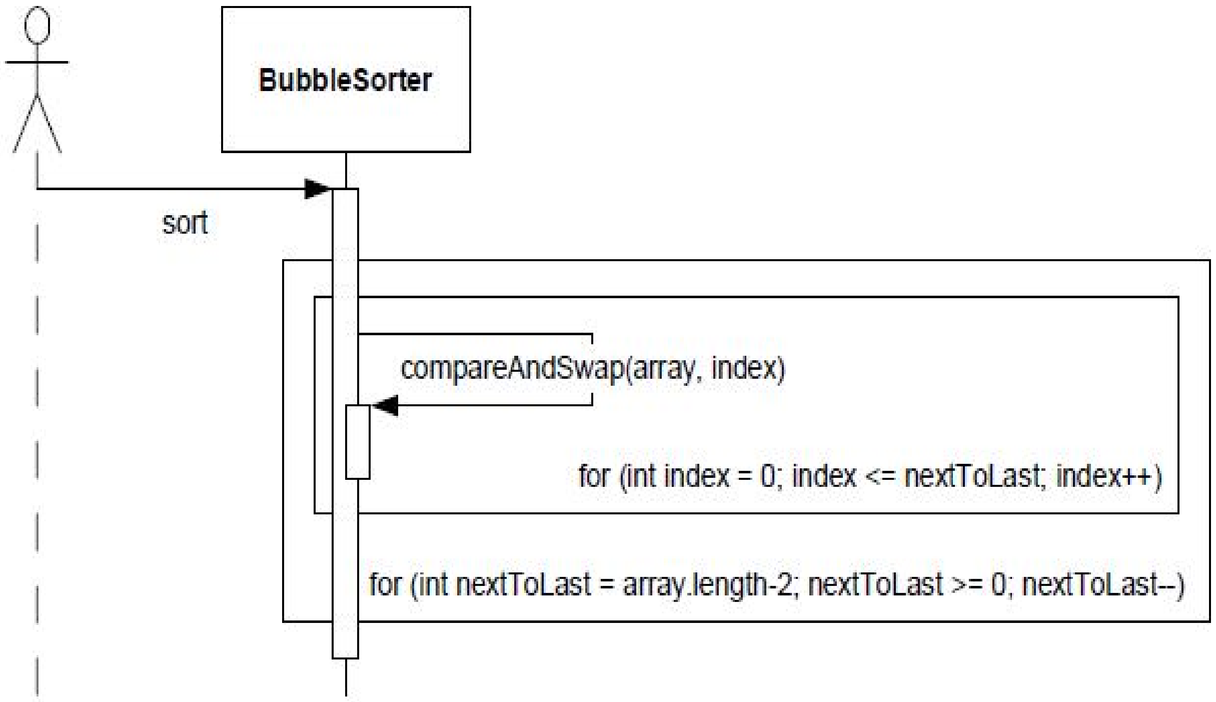
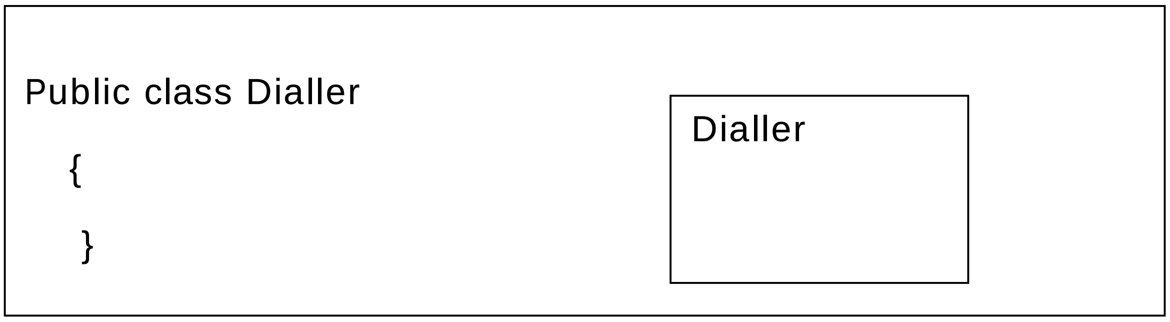

BubbleSorter
sort
compareAndSwap(array, index)
for (int index = 0; index <= nextToLast; index++)
for (int nextToLast = array.length-2; nextToLast >= 0; nextToLast--)

Figure 4.2

4.2 Class Diagrams

Classes

Figure 3-1 shows the simplest form of class diagram. The class named Dialler is represented as a simple rectangle. This diagram represents nothing more than the code shown to its right.



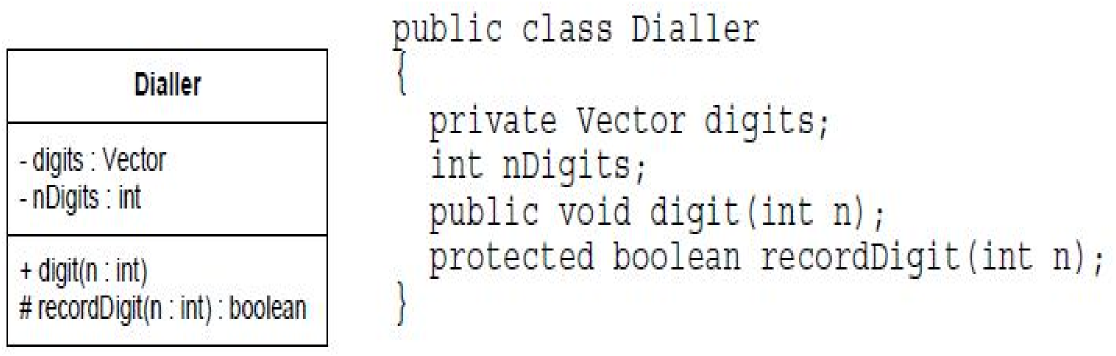
```
Public class Dialler
   {
    }
```

Figure 4.3

The Basics

This is the most common way you will represent a class. The classes on most diagramg don’t need any more than their name to make clear what is going on.

A class icon can be subdivided into compartments. The top compartment is for the name of the class, the second is for the variables of the class, and the third is for the methods of the class. Figure 4.4 shows these compartments and how they translate into code.



```
public class Dialler
{
  private Vector digits;
  int nDigits;
  public void digit(int n);
  protected boolean recordDigit(int n);
}
```

Figure 4.4

Notice the character in front of the variables and functions in the class icon. A dash (-) denotes private, hash (#) denotes protected, and plus (+) denotes public. The type of a variable or a function argument is shown after the colon following the variable or argument name. Similarly, the return value of a function is shows after the colon following the function.

This kind of detail is sometimes useful; but should not be used very often. UML diagrams are not the place to declare variables and function. Such declarations are better done in source code. Use these adornments only when they are essential to the purpose of the diagram.

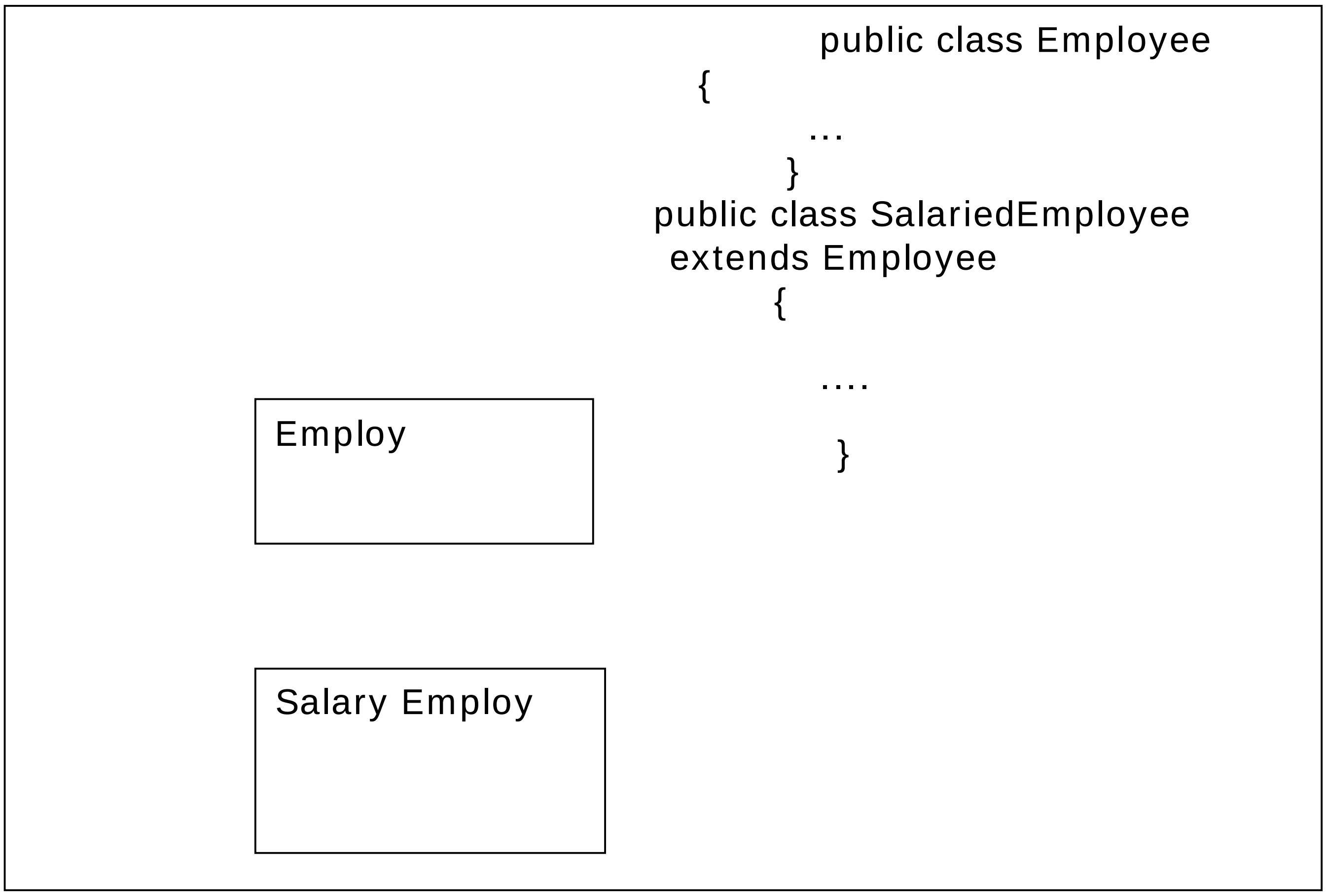
Association

Associations between classes most often represent instance variables that hold references to other objects. For example, in Figure 4.4 we see an association between Phone and Button. The direction of the arrow tells us that Phone holds a reference to Button. The name near the arrowhead is the name of the instance variable. The number near the arrowhead tells us how many references are held.

Inheritance

You have to be very careful with your arrowheads in UML. Figure 4.4 shows why. The little arrowhead pointing at Employee denotes *inheritance*1.

If you draw your arrowheads carelessly, it may be hard to tell whether you mean inheritance or association. To make it clearer, I often make inheritance relationships vertical and associations horizontal.



```
          public class Employee
  {
         ...
       }
public class SalariedEmployee
 extends Employee
       {

          ....

           }
```

Figure 4.5

In UML all arrowheads point in the direction of *source code dependency*. Since it is the SalariedEmployee class that mentions the name of Employee, the arrowhead points at Employee. So, in UML, inheritance arrows point at the base class.

UML has a special notation for the kind of inheritance used between a Java class and a Java interface. It is shown, in Figure 4.5, as a dashed inheritance arrow2. In the diagrams to come, you'll probably catch me forgetting to dash the arrows that point to interfaces.

An Example Class Diagram

A simple class diagram of part of an Examination System. This diagram is interesting both for what it shows, and for what it does not show. Note that I have taken pains to mark all the interfaces. I consider it crucial to make sure my readers know what classes I intend to be interfaces and which I intend to be implemented. For example, the diagram immediately tells you that WithdrawTransaction talks to a CashDispenser interface. Clearly some class in the system will have to implement the CashDispenser, but in this diagram we don't care which class it is.

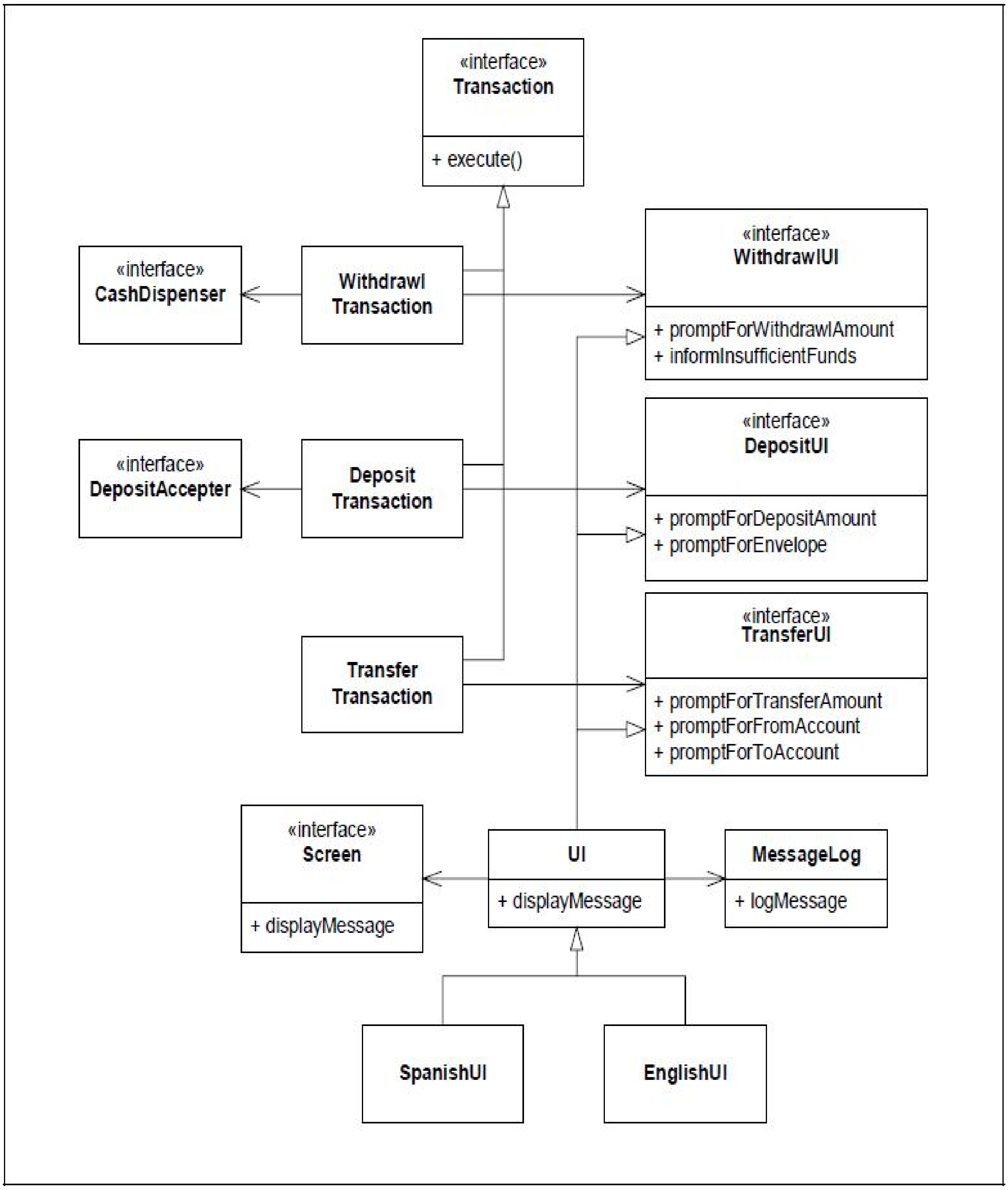
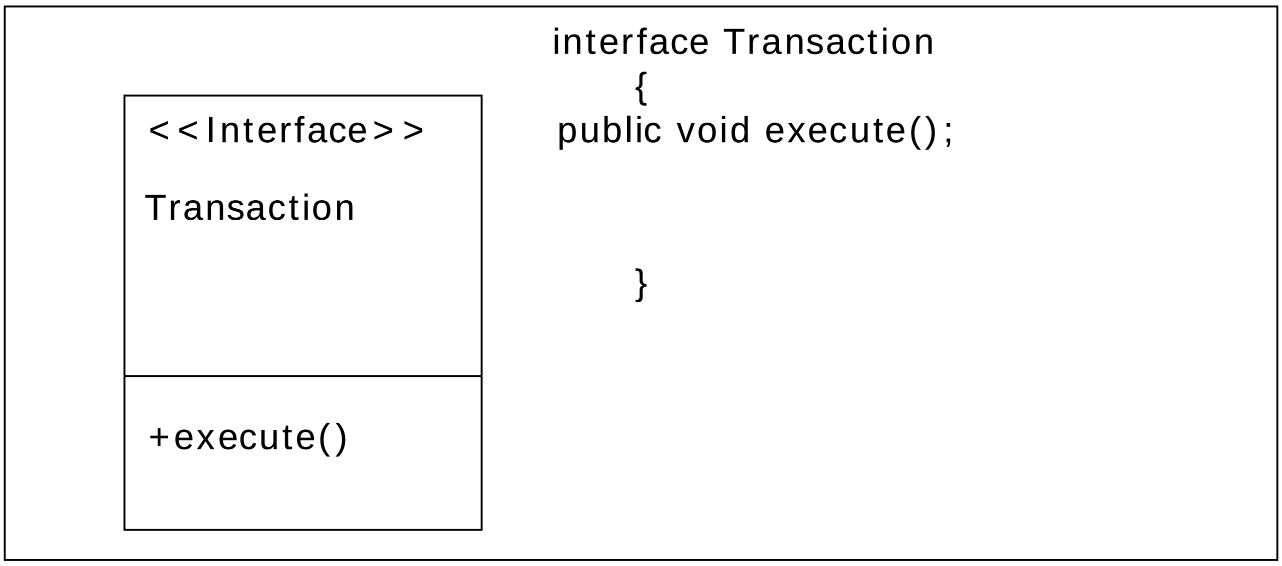

«interface»
Transaction
+ execute()
«interface»
CashDispenser
Withdrawl Transaction
«interface»
WithdrawlUI
+ promptForWithdrawlAmount
+ informInsufficientFunds
«interface»
DepositAccepter
Deposit Transaction
«interface»
DepositUI
+ promptForDepositAmount
+ promptForEnvelope
Transfer Transaction
«interface»
TransferUI
+ promptForTransferAmount
+ promptForFromAccount
+ promptForToAccount
«interface»
Screen
+ displayMessage
UI
+ displayMessage
MessageLog
+ logMessage
SpanishUI
EnglishUI

Figure 4- 6

4.2.1 Class Stereotypes

Class stereotypes appear between guilmette3 characters, usually above the name of the class. We have seen them before. The **«interface»** denotation in Figure 4.7 is a class stereotype. «interface» is one of two standard stereotypes that can be used by java programmers. The other is «utility».

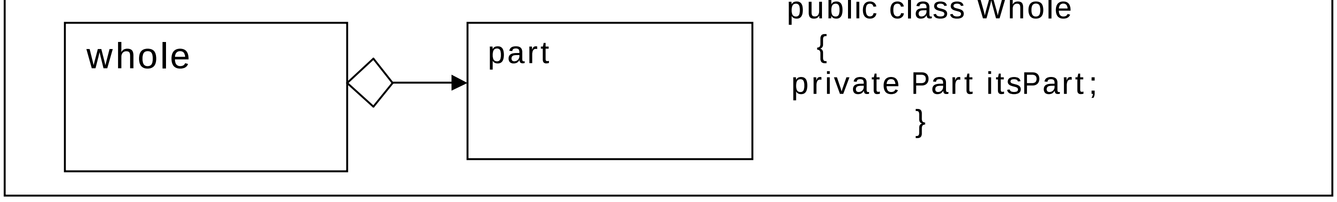
«interface». All the methods of classes marked with this stereotype are abstract. None of the methods can be implemented. Moreover, «interface» classes can have no instance variables. The only variables they can have are static variables. This corresponds exactly to java interfaces.



```
interface Transaction
      {
public void execute();


      }
```

Figure 4.7 shows how it is drawn and implemented.



```
public class Whole
  {
private Part itsPart;
          }
```

Figure 4-8

Unfortunately, UML does not provide a strong definition for this relationship. This leads to confusion because various programmers and analysts adopt their own pet definitions for the relationship. For that reason, I don't use the relationship at all; and I recommend that you avoid it as well.

The one hard rule that UML gives us regarding aggregations is simply this. A whole cannot be its own part. Therefore *instances* cannot form cycles of aggregations. A single object cannot be an aggregate of itself; two objects cannot be aggregates of each other; three objects cannot form a ring of aggregation, etc.

4.3 Sequence Diagrams:

Objects, Lifelines, Messages, and other odds and ends

Figure 4.9 shows a typical sequence diagram. The objects involved in the collaboration are shown at the top. The stick figure (actor) at left represents an anonymous object. It is the source and sink of all the messages entering and leaving the collaboration.

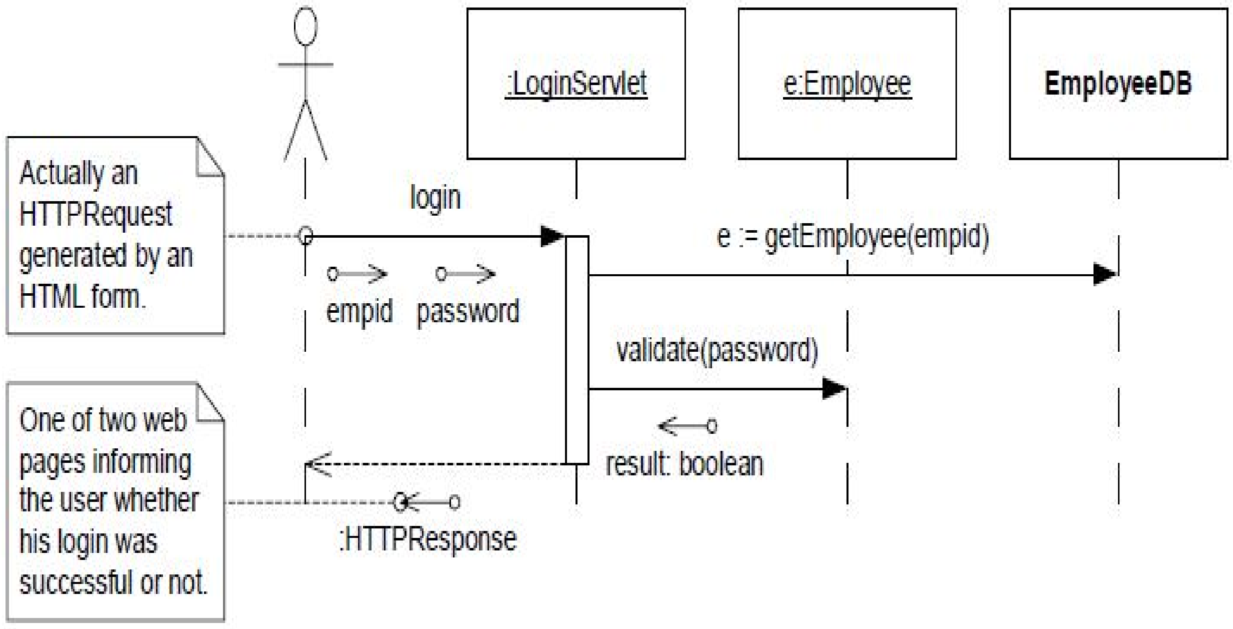

:LoginServlet
e:Employee
EmployeeDB
Actually an HTTPRequest generated by an HTML form.
login
empid
password
e := getEmployee(empid)
validate(password)
result: boolean
One of two web pages informing the user whether his login was successful or not.
:HTTPResponse

Figure 4.9

The dashed lines hanging down from the objects and the actor are called *lifelines*. A message being sent from one object to another is shown as an arrow between the two lifelines.

Each message is labeled with its name. Arguments appear either in the parenthesis that follow the name or next to *data tokens* (the little arrows with the circles on the end).

Time is in the vertical dimension, so the lower a message appears the later it is sent.

The skinny little rectangle on the lifeline of the LoginServlet object is called activation. Activations are optional; most diagrams don't need them. They represent the time that a function executes. In this case it shows how long the login function runs. The two messages leaving the activation to the right were sent by the login method. The unlabeled arrow shows the login function returning to the actor, and passing back a return value. Finally, notice that EmployeeDB is a class, not an object. This can only mean that getEmployee is a static method. Thus, we'd expect EmployeeDB to be coded as in Listing 4-1.

Figure 4-1

Typical Sequence Diagram

Listing 4-1

EmployeeDB.java

```
public class EmployeeDB
{
public static Employee getEmployee(String empid)
{
...
}
...
}
```

Creation and Destruction

We can show the creation of an object on a sequence diagram using the convention shown in Figure 4.10. An unlabeled message terminates on the object to be created, not on it's lifeline. We would expect ShapFactory to be implemented as shown in Listing 4.9.

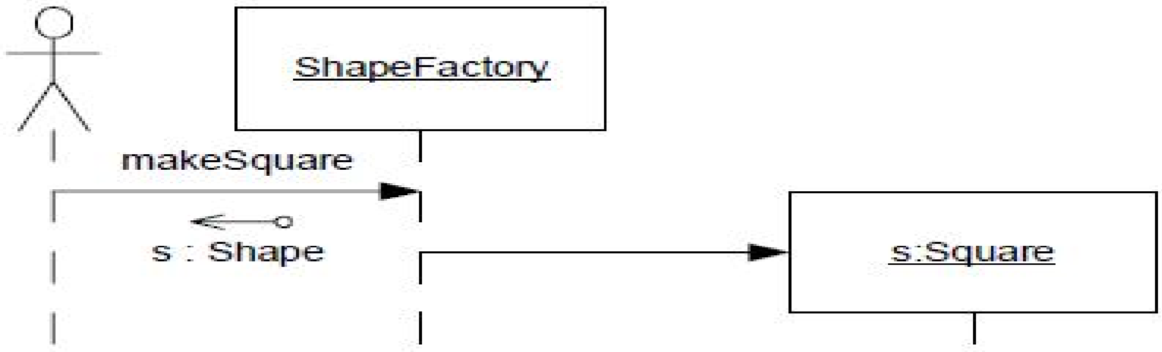


Figure 4.10

Listing 4-2

```
ShapeFactory.java
public class ShapeFactory
{
public Shape makeSquare()
{
return new Square();

}

}
```

In Java we don't explicitly destroy objects. The garbage collector does all the explicit destruction for us. However, there are times when we want to make it clear that we are done with an object and that, as far as we are concerned, the garbage collector can have it.

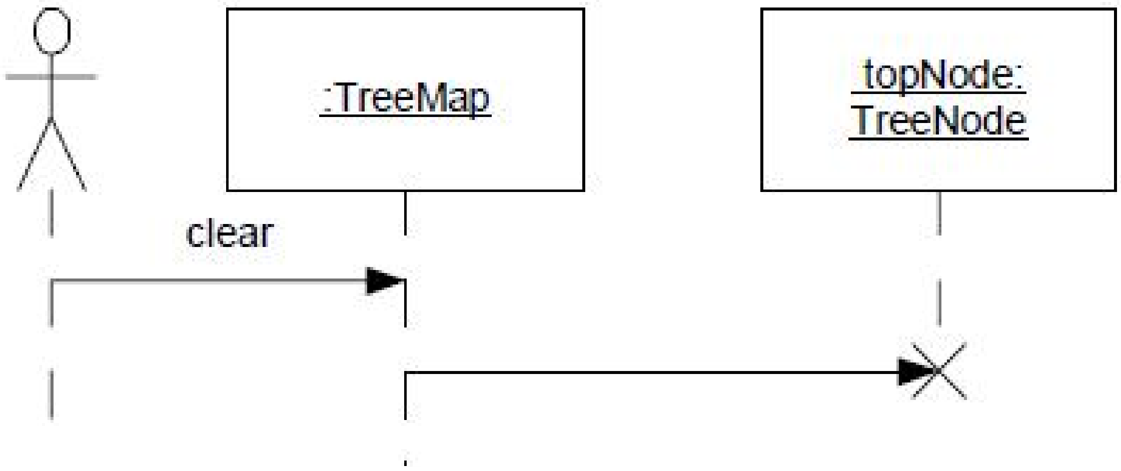


Figure 4.11

Listing 4-3

TreeMap.java

```
public class TreeMap
{
private TreeNode topNode;
public void clear()
{
topNode = nil;
}
```

Advanced Concepts

Loops and Conditions: In Figure 4.11 you can see the payroll algorithm, complete with well specified loops and if statements.

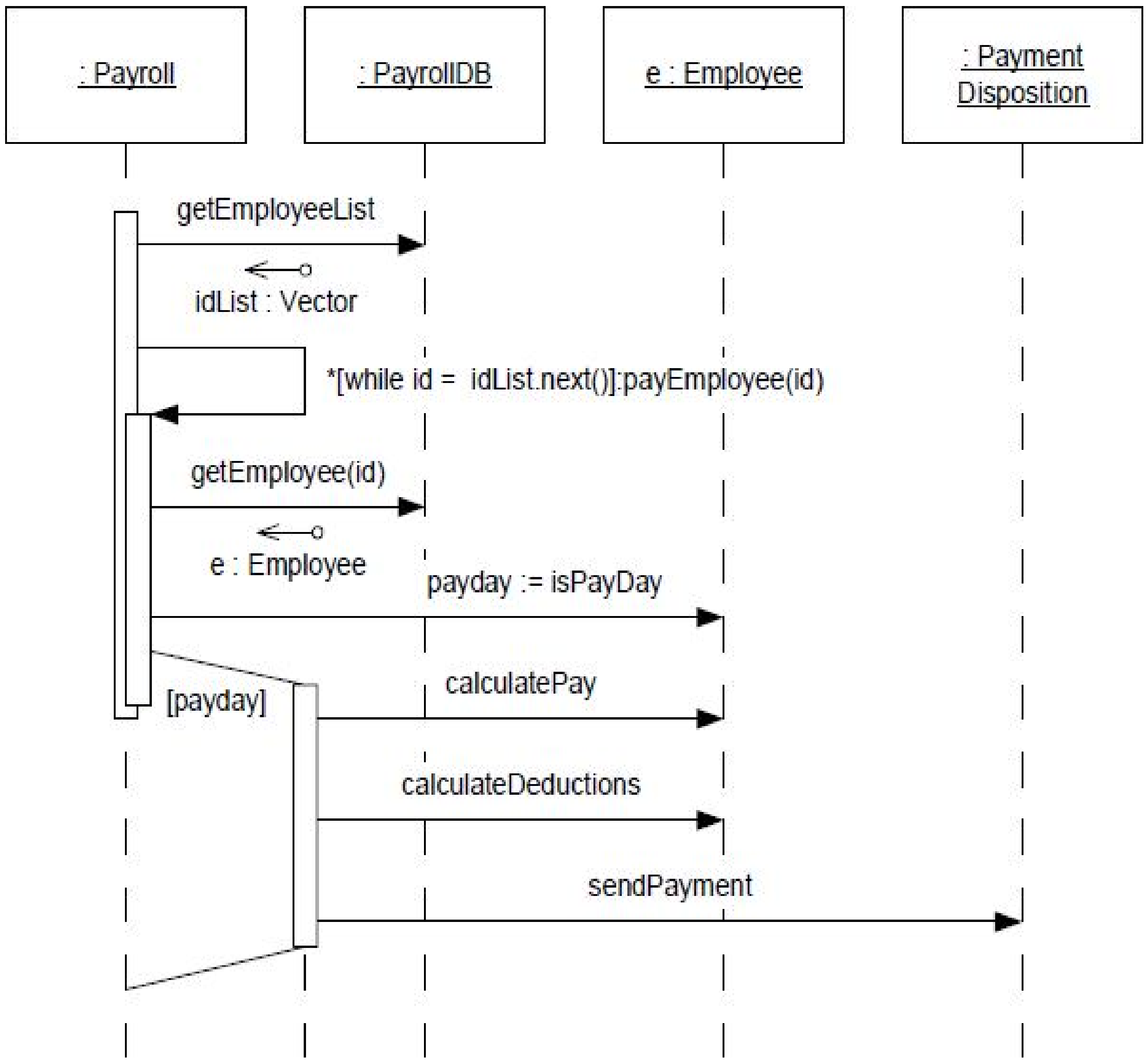


Figure 4.12

The payEmployee message is prefixed with an *recurrence* expression that looks like this:

*[while id := idList.next()]

The star tells us that this is iteration; the message will be sent repeatedly until the *guard* expression in the square brackets is false. UML does not specify a syntax for the guard expression so I have used a java-like pseudo code that suggests the use of an iterator.

The payEmployee message terminates on an activation that is touching, but offset from, the first. This denotes that there are now two functions executing in the same object.

Since the payEmployee message is recurrent, the second activation will also be recurrent, and so all the messages depending from it will be part of the loop.

4.4 Use Cases

Use cases are a wonderful idea that has been vastly overcomplicated. Over and over again I have seen teams sitting and spinning in their attempts to write use cases. Typically they thrash on issues of form rather than substance. They argue and debate over preconditions, post conditions, actors, secondary actors, and a whole bevy of other things that just don’t matter.

Use Cases Diagrams

Of all the diagrams in UML, use case diagrams are the most confusing, and the least useful.

With the exception of the System Boundary Diagram, which I’ll describe in a minute,
I recommend that you avoid them entirely.

System Boundary Diagram

Figure 4.12 shows a System Boundary Diagram. The large rectangle is the system boundary.

Everything inside the rectangle is part of the system under development. Outside the rectangle we see the *actors* that *act* upon the system. Actors are entities outside the system that provide the stimuli for the system. Typically they are human users. They might also be other systems, or even devices such as real-time clocks.

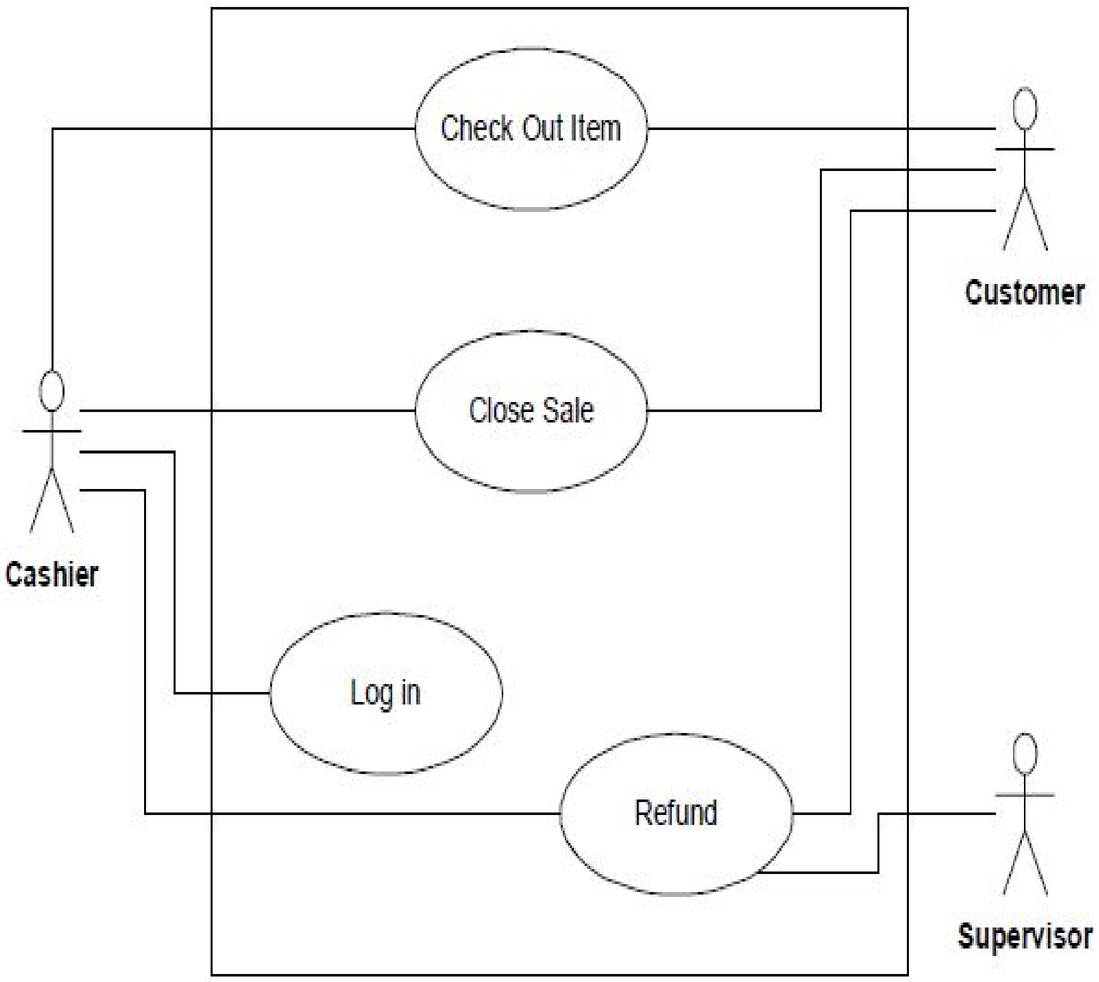


Figure 4.13

4.5 CLASS- AND STORY-DIAGRAM RECONSTRUCTION

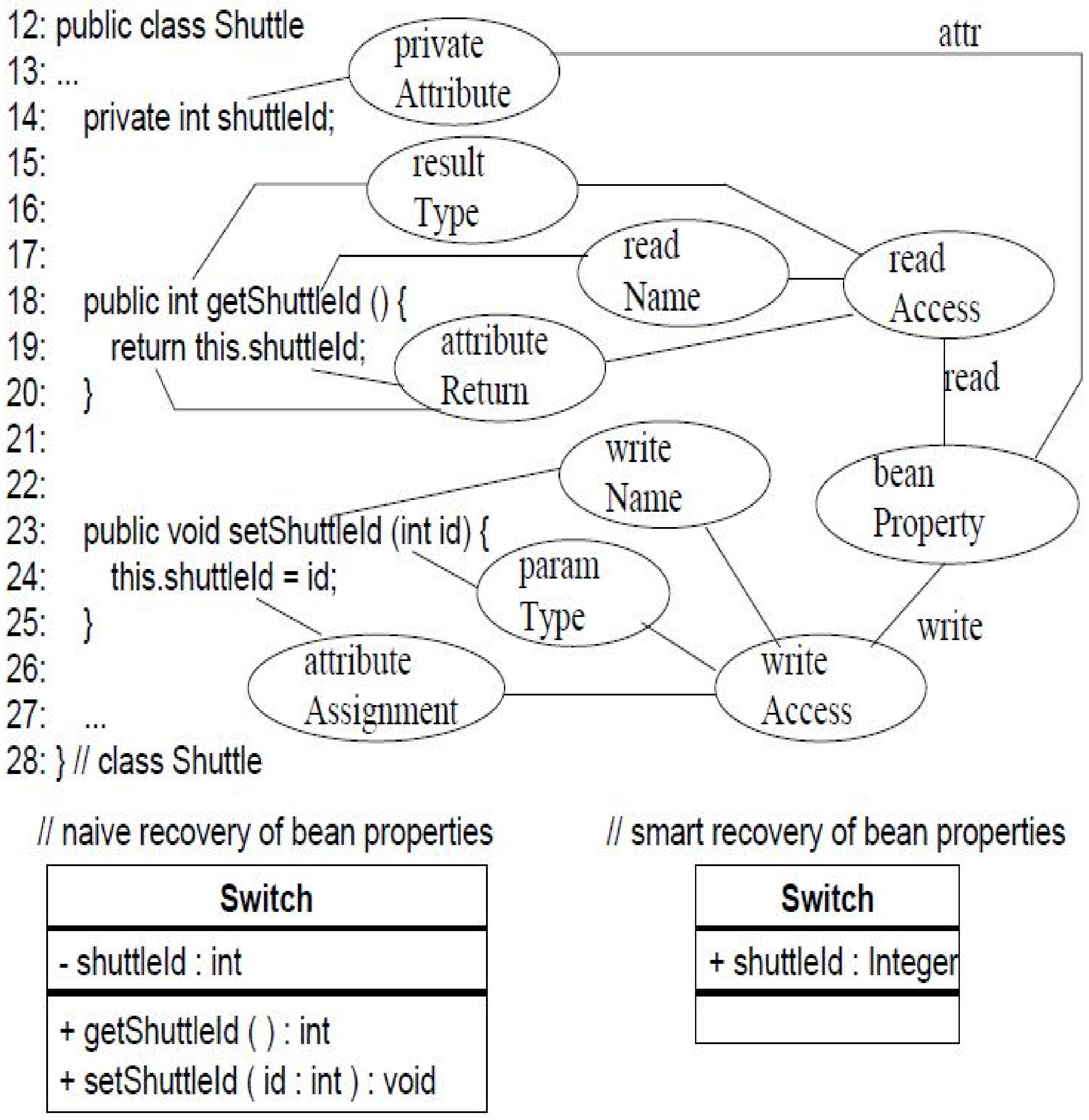
Recovering class and behaviour diagrams from Java code is divided into two tasks. First, the static information, the class diagrams, will be reconstructed and in a second task, the behaviour diagrams (here story-diagrams) will be recognized.

```
1: public class Switch extends Track {
2: ...
3: int shuttleId;
4: ...
5: Stopper stopper = new Stopper ();
6: ...
7: OrderedSet announced = new OrderedSet ();
8: ...
9: void welcome (int id) {...}
10: ...
11 :} // class Switch
```

From this code fragments a rudimentary class diagram recovery approach could reconstruct the class diagram shown below the Java code.

Classes become classes. Attributes of basic types like int or boolean become class attributes. Method declarations become methods of the corresponding UML classes. Inheritance in Java is directly mapped to inheritance relations in the diagram.

We assume that the class diagram recovery mechanism has already knowledge about all basic Java types.



```
public class Shuttle
...
   private int shuttleId;



   public int getShuttleId () {
      return this.shuttleId;
   }


   public void setShuttleId (int id) {
      this.shuttleId = id;
   }

   ...
} // class Shuttle
```

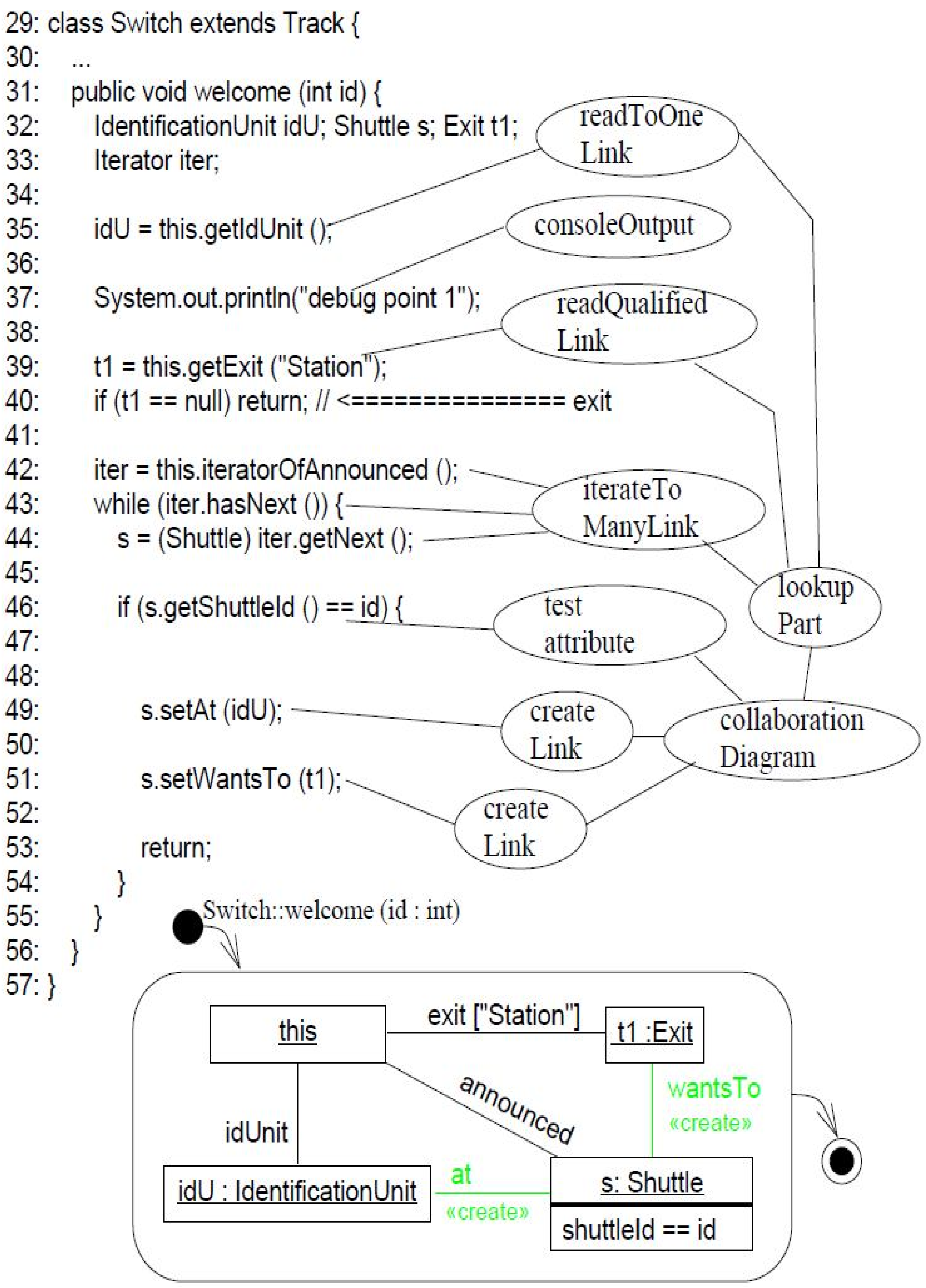
Figure 4.14

However, class Ordered Set is a pre-defined generic container class from the Java Foundation Class (JFC) library.

Equipping our reconstruction mechanism with this additional knowledge, it could turn the announced reference from Switch to Ordered Set into a to-many reference to class Object (the basic class of all classes in Java). In Java, we face the problem that generic container classes do not provide information about the types of the contained entities. To derive such information, our recovery mechanism needs to know the semantics of the access methods of container classes, e.g. method add inserts elements into the container. This allows us to derive the entry type for containers from the usages of the corresponding add method. In our approach, we use similar code clichés to implement bi-directional associations. Bi-directional associations are implemented using pairs of pointers. These pointers are encapsulated with appropriate read and write access methods. The write access methods guarantee the consistency of the pointer pairs by calling each other, mutually. For to-one associations simple attributes and set and get-methods are used. For to-many associations we employ generic container classes and methods for iterating through the set of neighbors, adding neighbors, and removing neighbors. Fujaba employs a flexible cliché detection mechanism the so-called annotation engines. The annotation engines enrich the abstract syntax tree of a parsed program with so-called annotations. Annotations are markers for detected occurrences of code clichés. In Figure 4.14 such annotations are shown as ovals. Annotations enrich the semantics information of abstract syntax trees and allow e.g. to simplify class diagrams. Moreover, such annotations assign certain semantics to certain methods or code fragments. This semantics may be used for further analysis of other method bodies.

Consider for example Figure 4.14. Line 35 employs method getIdUnit. Let us assume that method getIdUnit has been annotated as the read access method for an association between class Switch and class Identification Unit. This allows us to interpret line 35 as a link look-up operation. In a collaboration diagram such a link look-up operation is shown as a line labeled with the corresponding association name. Similarly, the knowledge about access methods may allow interpreting line 39 and 40 as look-up of a qualified association with cardinality 0...1. Lines 42 to 44 show a typical cliché for the look-up of a to-many association. In a collaboration diagram such look-ups are shown as lines between appropriate objects; cf. the bottom of Figure 4.14. Finally, our annotation knowledge allows us to infer, that line 49 creates an at link between object s and idU and line 51 creates a wants to link from s to t1. In the collaboration diagram we show link creation using grey color and the «create» stereotype.

Thus, the detection of Java bean property clichés allows us to assign a dedicated semantics to read- and write-access methods. This knowledge allows us to analyse the usage of such access methods and to recover collaboration diagrams from such code. Note, in more complex situations, a method body may contain several code fragments that correspond to collaboration diagrams.



```
class Switch extends Track {
    ...
    public void welcome (int id) {
        IdentificationUnit idU; Shuttle s; Exit t1;
        Iterator iter;

        idU = this.getIdUnit ();

        System.out.println("debug point 1");

        t1 = this.getExit ("Station");
        if (t1 == null) return; // <=============== exit

        iter = this.iteratorOfAnnounced ();
        while (iter.hasNext ()) {
            s = (Shuttle) iter.getNext ();

            if (s.getShuttleId () == id) {


                s.setAt (idU);

                s.setWantsTo (t1);

                return;
            }
        }
    }
}
```

Figure 4.15

To deal with such situations, we embed detected collaboration diagrams into activity diagrams. The activity diagram part shows the top level control flow and text activities for unrecognized code. Recognized code is turned into collaboration diagram activities.

So far, our annotation engines are able to deal with code that strictly conforms to the Fujaba code generation concepts.

Due to our experience, legacy and third party code frequently contains similar code fragments. However, the classification of access methods and especially the detection of collaboration diagram operations is very challenging. For example, there are numerous ways to implement a to-many association. Accordingly, there are very different ways to enumerate all neighbours of a given object. Similarly, there are various coding clichés for test operations and for dealing with test results. The next section describes how clichés can be specified using the Fujaba environment

Chapter 5

Result And Testing

For the interaction diagram approach, existing tools support both generating code from sequence diagrams and reverse engineering this code. Additionally, these tools can usually be integrated with the other necessities of software development, like GUI builders and persistence providers. The approach can be applied without having to stop changing program code by hand using the vast amount of very good development environments available today. The changed code can then be reverse engineered into the better visualization and documentation an UML model provides. However, when more than one iteration is employed, some of the manually entered program code will have to be re-entered as full code generation from sequence diagrams is not yet possible. Additionally, using interaction diagrams can become unfeasible when working with huge method bodies comprising a large amount of conditional logic.

For the state chart diagram approach, the tool support is good in the analysis and design phase. The usability of this approach for a certain project is dependent on the commercial availability of a flexible and robust model compiler for the target language. If this model compiler is not available as it was the case with the Java language, it has to be built and configured for the aims of the project.

5.1 Resultant tool's snap shot are below.

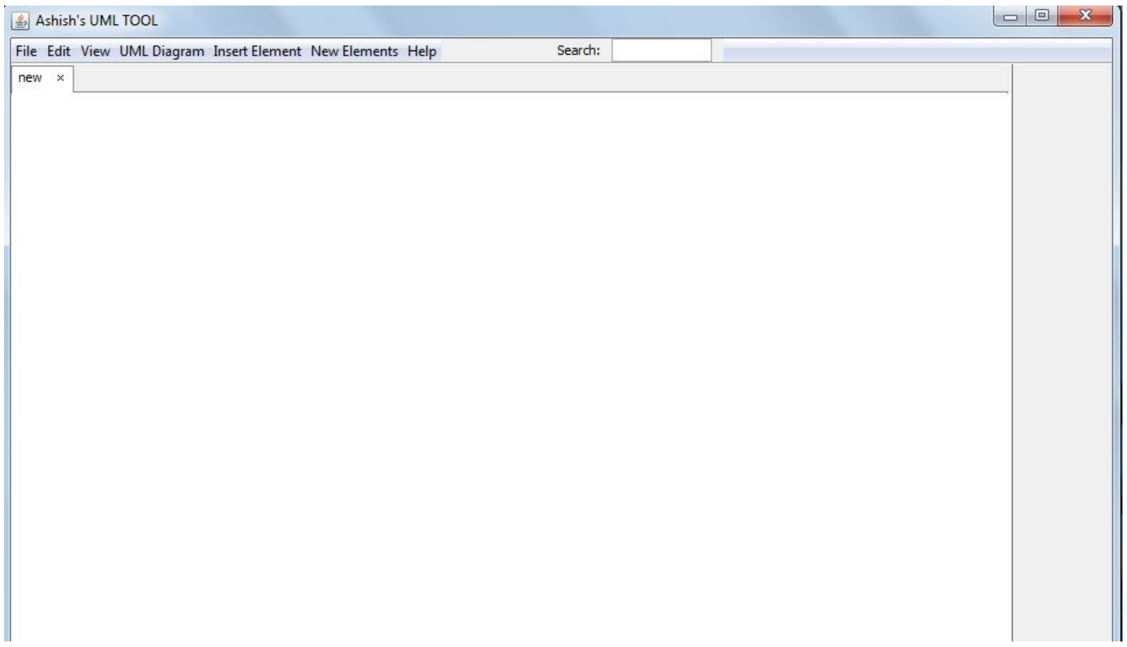


Figure 5.1 GUI of UML tool

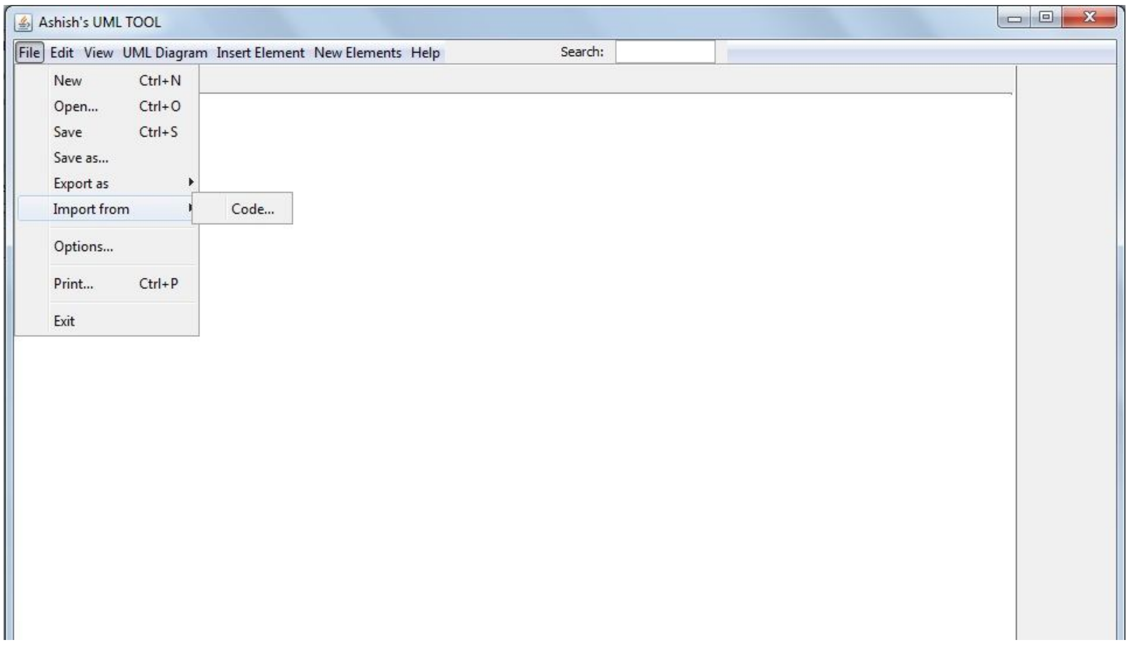
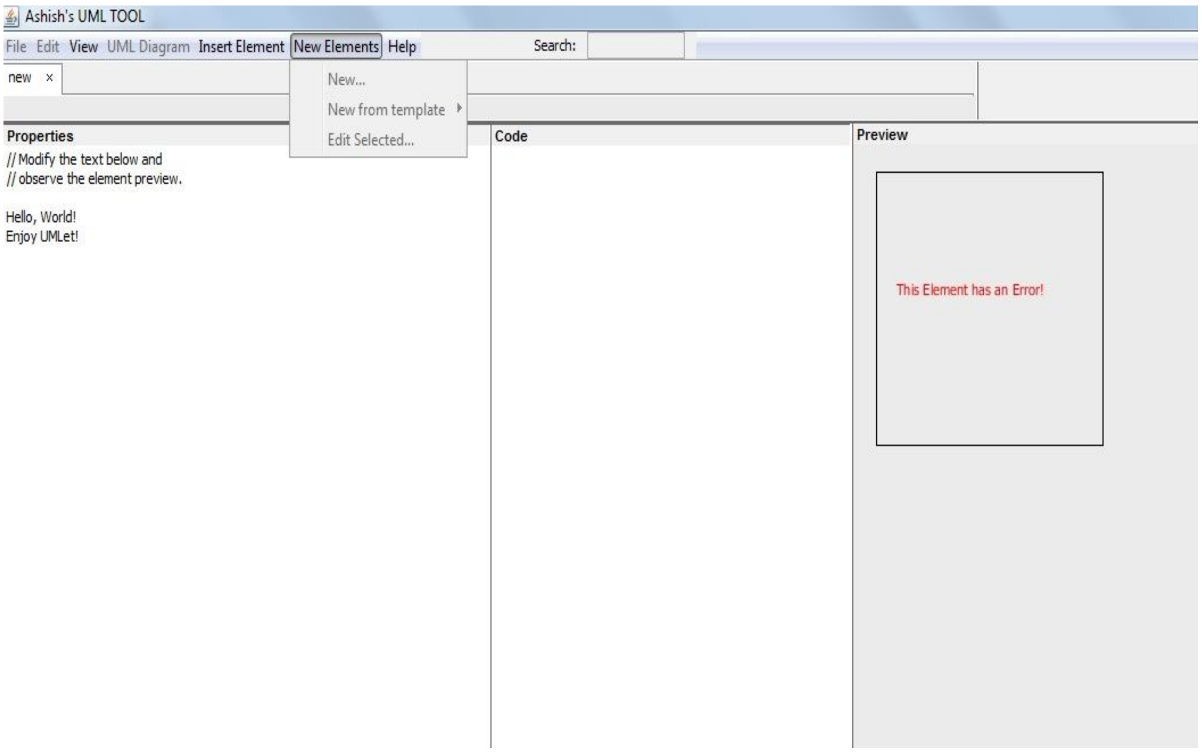


Figure 5.2 Codes to Diagram Converter



// Modify the text below and
// observe the element preview.

Hello, World!
Enjoy UMLet!

This Element has an Error!

Figure 5.3 Customization of UML elements

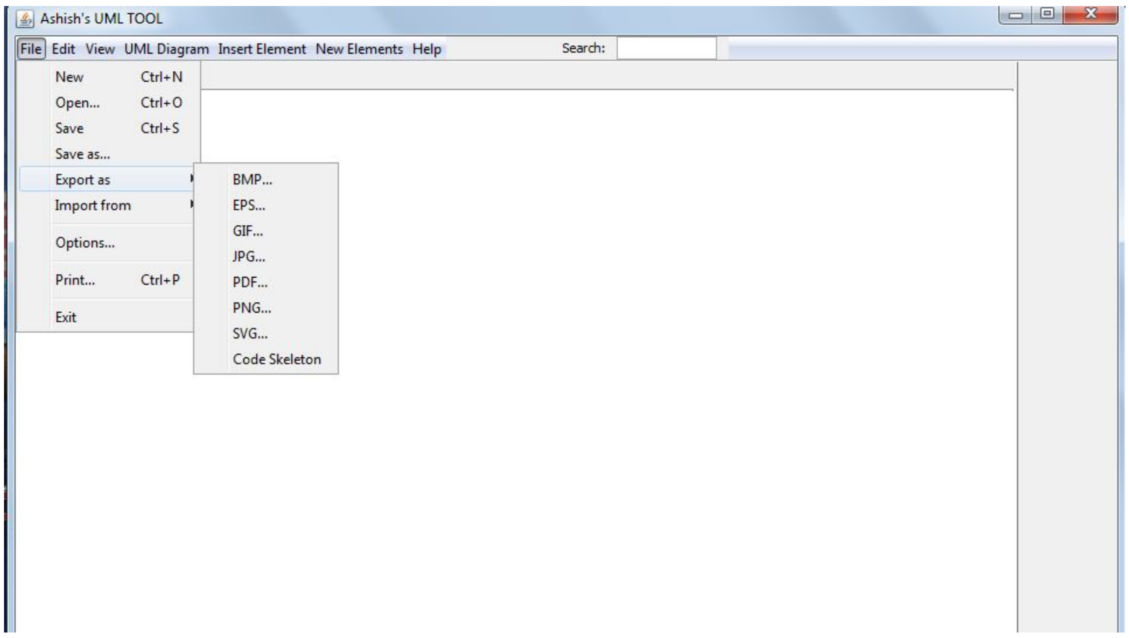


Figure 5.4 Diagram to code converter

5.2 Testing:

SYSTEM TESTING

Here the System testing involved is the most widely used testing process consisting of five stages as shown. In general, the sequence of testing activities is component testing, integration testing, and then user testing. However, as defects are discovered at any one stage, they require program modifications to correct them and this may require other stages in the testing process to be repeated Testing Flow.

TESTING OBJECTIVES:

The main objective of testing is to uncover a host of errors, systematically and with minimum effort and time. Stating formally, we can say,

- Testing is a process of executing a program with the intent of finding an error.
- A successful test is one that uncovers an as yet undiscovered error.
- A good test case is one that has a high probability of finding error, if it exists.

- The tests are inadequate to detect possibly present errors.
- The software more or less confirms to the quality and reliable standards.

Unit testing

Unit testing focuses verification effort on the smallest unit of software i.e. the module. Using the detailed design and the process specifications, testing is done to uncover errors within the boundary of the module. All modules must be successful in the unit test before the start of the integration testing begins.

In this project each service can be thought of a module. There are so many modules like Login, MARKETING Department, Interviewer Section, etc. Each module has been tested by giving different sets of inputs. When developing the module as well as finishing the development, the module works without any error. The inputs are validated when accepting them from the user.

Integration Testing

After unit testing, we have to perform integration testing. The goal here is to see if modules can be integrated properly, the emphasis being on testing interfaces between modules. This testing activity can be considered as testing the design and hence the emphasis on testing module interactions.

In this project the main system is formed by integrating all the modules. When integrating all the modules I have checked whether the integration effects working of any of the services by giving different combinations of inputs with which the two services run perfectly before Integration.

5.3 Related Work

The related work copes both with the generation of code from interaction diagrams as with the generation of code from state chart diagrams, especially with the xUML profile of UML.

For the first topic, mainly theoretical work has been done to enhance the interaction diagram semantics to be able to generate code from them is such a paper, presenting a way to map sequence diagrams to interaction graphs which deal with defining the scope of the variables included in the graph. Lieberherr also defines a possibility for automatic object passing were the object is automatically provided to all places the visibility allows. The developer does not have to care about passing the object on to subsequent method calls, the according parameters are generated automatically. Of course, this means the naming scope is being enhanced to be the whole sequence diagram; or at least the part of the sequence diagram following the object's creation. This approach is part of the so called aspect-oriented programming. Both forward engineering and reverse engineering are possible with Lieberherr's approach.

6.References:

1:- J. Rho and C. Wu. An efficient version model of software diagrams. In Oroceedings of the 5th Asia Pacific Software
Engineering Conference, page 236, Washington, DC, USA, 1998. IEEE Computer Society

2:- Z. Xing and E. Stroulia. Umldiff: an algorithm for objectoriented design differencing. In ASE ’05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 54–65, New York, NY, USA, 2005. ACM.

3:- KAGDI H., MALETIC J.I., SUTTON A.: ‘Context-free slicing of UML class models’. Proc. 21st IEEE Int. Conf. on Software Maintenance (ICSM’05), Washington, DC, USA, 2005, pp. 635–638

4:- *Engineering: Principles, Processes, Methods, and Tools.* SAE International, pp.164-166, June 2005.

5:-Rational Rose RealTime Online Help, rosert_tutorials_guide.chm, June 2008.

6:- M. Bittner and F. Kammller. Translating Fusion/UML to Object-Z. *First ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, IEEE Press, New York, 2003, pp. 49.

7:- S. Dupuy Y. Ledru and M. Chabre-Peccoud. An Overview of RoZ: A Tool for Integrating UML and Z Specifications. In *Advanced Information Systems Engineering*, Springer, Heidelberg, 2000, pp. 417–430.

8:- Fenzel D., The Semantic Web and its languages, IEEE Intelligent Systems, November/December 2000,

9-: Rational Unified Process User Guide, Rational Corp, www.rational.com.

10:- Resource Description Framework (RDF) Schema Specification 1.0, WWW Consortium, 2000, document http://www.w3c.org/TR/2000/CR-rdf-schema- 20000327.

11:- Joaquin Miller. What UML should be. Communications of the ACM, 45(11):67–69, 2002.

12:- Hermann Kaindl Difficulties in the transition from OO analysis to design. IEEE Software, 16(5):94–102, 2009.

13-: Cris Kobryn. UML 2001: A standardization odyssey. Communications of the ACM, 42(10):29–37, 2007.

14-: Dov Dori. Why significant UML change is unlikely. Communications of the ACM, 45(11):82–85, 2008.

15-: Keith Duddy. UML must enable a family of languages. Communications of the ACM, 45(11):73–75, 2010.

16-: Mary Campione, Kathy Walrath, and Alison Huml. The
Java(TM) Tutorial: A Short Course on the Basics (3rd Edition). Pearson Education, Upper Saddle River, third edition, 2008.

17-: Yai hui Difficulties in the transition from OO analysis to design. IEEE Software, 342(5):124–12, 2009.