

DIPLOMARBEIT

**Towards Executable UML -
Code Generation
From Interaction
And State Chart Diagrams**

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Diplom-Ingenieurs unter der Leitung von

O.UNIV.PROF. DIPL.-ING. DR.TECHN. RICHARD EIER
Institut für Computertechnik, E384

und

UNIV.ASS. DIPL.-ING. WOLFGANG RADINGER
als verantwortlich mitwirkendem Betreuer

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

MARTIN MARINSCHKEK
Matrikelnummer: 9803246
Apostelgasse 20/30, A-1030 Wien

Wien, Mai 2003

Abstract

The Unified Modeling Language (UML) has grown in its importance for the software development industry and strives to provide the *blueprints* for a new approach to software design. The idea is to drive the software architecture by models drawn in UML, creating a new Model Driven Architecture (MDA) for software systems. An important aspect of the MDA is code generation from the models which is today often restricted to the *static structure* of the software system. If the models shall be the only necessary prerequisite of software development, the *dynamical aspect* has to be generated as well. Two approaches for achieving complete code generation from UML models are discussed in this thesis. The approach built on the basic UML uses *interaction diagrams*; the approach of the Executable UML (xUML) profile uses *state chart diagrams* for code generation of the software system's dynamical aspect.

This thesis uses an example, a training system for the mathematical skills of elementary school students, to illustrate the process of Object-Oriented Analysis and Design (OOAD) using UML and the xUML profile. The artefacts of this process are documented, and differences between the two approaches are discussed. Subsequently, the system is implemented by generating code from the xUML and UML models. Various commercial tools are used to generate both the *static structure* and the *dynamical aspect*. The goal is to automatically code as much as possible, leaving only little left for the error-prone manual coding.

This aids in reducing expenses spent on documentation, on implementation and on maintenance. Documentation costs are reduced as major parts of it can be retrieved from the model. Implementation is less expensive as the models are verifiable and executable soon, errors are found early in the process – a key cost reduction factor. Expenses spent on maintenance are reduced as the maintenance of the model happens together with the maintenance of the code and the system is better understood by developers when visualized in models.

A final comparison evaluates advantages and disadvantages of both approaches for the development of the above mentioned example. These differences are important when deciding which strategy to embark upon for designing a software system. It has to be mentioned that full code generation is possible with neither of these approaches. The main problems are obstacles in modelling the Graphical User Interface (GUI) and the persistency layer in UML. However, modelling of other parts of the system and code generation from these models can reduce the need for manual coding and hence lower the costs of software development.

Kurzfassung

Die Unified Modeling Language (UML) wird mehr und mehr zu einem wichtigen Faktor in der Softwareindustrie und stellt "Skizzen" für eine neue Art von Softwareentwurf bereit. Der Versuch, über in UML erstellte Modelle die Softwarearchitektur zu steuern, wird Model Driven Architecture (MDA) genannt. Ein wichtiger Aspekt dieser MDA ist die Generierung von Code; heute oft nur der *statischen Struktur* von Softwaresystemen. Wenn die Modelle die einzig notwendige Voraussetzung für die Softwareentwicklung sein sollen, muss aber auch der *dynamische Aspekt* berücksichtigt werden. Zwei Ansätze für die Codegenerierung auch des dynamischen Aspekts aus UML Modellen werden in dieser Arbeit behandelt, der Unterschied zwischen ihnen liegt in Ausgangsbasis für die Codegenerierung. Der auf der standardisierten UML basierende Vorschlag verwendet *Interaktionsdiagramme* zur Codegenerierung, das Executable UML (xUML) Profil von UML hingegen *Zustandsdiagramme*.

Diese Arbeit illustriert unter Verwendung von UML und xUML an einem Trainingssystem für die mathematischen Fähigkeiten von Volksschülern den Prozess von Objektorientierter Analyse und Design (OOAD). Die Ergebnisse des Prozesses werden dokumentiert und die Unterschiede zwischen den Ansätzen herausgearbeitet. Anschließend wird das Beispiel durch Codegenerierung von den Modellen der UML und des xUML-Profiles implementiert. Mehrere kommerzielle Softwarewerkzeuge werden für die Generierung der *statischen Struktur* und des *dynamischen Aspekts* eingesetzt. Das Ziel ist dabei die möglichst vollständige Generierung der Programmzeilen aus den Modellen; möglichst wenig soll der fehleranfälligen manuellen Programmierung überlassen bleiben.

Diese Herangehensweise hilft bei der Reduzierung der Dokumentations-, Implementierungs- und Wartungskosten. Die Dokumentationskosten sind niedriger, da wichtige Teile der Dokumentation direkt aus dem Modell generiert werden. Die Implementierungskosten werden durch die frühe Verifizier- und Ausführbarkeit der Modelle herabgesetzt weil die dadurch eintretende Fehlerfrüherkennung zu niedrigeren Behebungskosten führt. Die Wartungskosten werden durch die gemeinsame Wartung von Modellen und Code und durch die bessere Verständlichkeit des Systems durch immer aktuelle Modelle verringert.

Ein abschließender Vergleich evaluiert die Vor- und Nachteile der beiden Herangehensweisen für die Entwicklung des oben genannten Beispielsystems; diese Unterschiede sind für die Auswahl einer der Entwicklungsstrategien wichtig. Letztendlich kann keine dieser Varianten eine vollständige Codegenerierung ermöglichen, Probleme dabei sind vor allem die Hindernisse bei der Modellierung der graphischen Benutzerumgebung und der Persistenzschicht in UML. Trotzdem kann die Codegenerierung anderer Teile des Systems zu einer Verringerung des Aufwands an manueller Kodierung und damit zu einer Verkleinerung der Softwareentwicklungskosten führen.

Acknowledgements

First, a very special thanks to my thesis advisor Wolfgang Radinger for his support – both his immense knowledge and his intense long-distance coaching made it possible to finish this thesis while being in France.

Thanks to the people at Kennedy Carter and ProjTech, who helped me with invaluable hints in using their software.

Special thanks to my flat-mates, Claudia Hofstadler for her willingness to accept my distraction and her first review of this text and Doris Marinschek, for her willingness to put up with the organizational work while I was in France.

Most of all I would like to thank my parents–without their endless support and understanding my studies would not have been possible. This thesis is dedicated to them.

Contents

1	Introduction	1
1.1	Motivation	3
1.2	Goals	4
2	Technical Background	5
2.1	The Software Development Life-Cycle	5
2.2	The Unified Modeling Language	8
2.3	The Executable Unified Modeling Language Profile	14
2.3.1	The Maturity of xUML	15
2.3.2	Modelling in xUML	16
2.4	The Java Technology	20
2.5	Patterns for Object-Oriented Software Engineering	22
3	Object-Oriented Analysis and Design Phase	24
3.1	The Process	24
3.2	The Analysis and Design Tools	27
3.2.1	Rational Rose 2002	27
3.2.2	Together ControlCenter	27
3.2.3	iUML	28
3.3	The Requirements	28

3.4	The Object-Oriented Analysis and Design Phase using UML	29
3.4.1	The Object-Oriented Analysis	29
3.4.2	The Object-Oriented Design	43
3.5	The Object-Oriented Analysis and Design Phase using xUML	56
3.5.1	The Object-Oriented Analysis	57
3.5.2	The Object-Oriented Design	62
3.6	Summary	69
4	Implementation Phase	71
4.1	Implementation on the Basis of the UML model	71
4.1.1	Code Generation	71
4.1.2	The Graphic User Interface Layer	76
4.1.3	The Persistence Layer	78
4.2	Implementation on the Basis of the xUML Model	80
4.2.1	Code generation	80
4.2.2	The Graphic User Interface Layer	83
4.2.3	The Persistence Layer	84
4.3	Summary	84
5	Summary, Results and Future Work	86
5.1	Results	87
5.2	Related Work	88
5.3	Future Work	89
A	Additional Use Case Descriptions	91
B	Additional Sequence Diagrams	105

C Additional State-Chart Diagrams	110
List of Figures	117
List of Tables	118
List of Examples	119

Abbreviations

API	Application Programming Interface
ASL	Action Specification Language
CASE	Computer Aided Software Engineering
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CVS	Concurrent Versions System
CWA	Common Warehouse Metamodel
DSTC	Distributed Systems Technology Center
EAI	Enterprise Application Integration
EBC	Entity-Boundary-Controller
EBP	Elementary Business Process
EDOC	Enterprise Distributed Object Computing
EJB	Enterprise Java Beans
ER	Entity Relationship
GUI	Graphical User Interface
JDBC	Java Database Connectivity
JDK	Java Development Kit
JDO	Java Data Objects
JIT	Just In Time
JNI	Java Native Interface
JRE	Java Runtime Environment
MDA	Model Driven Architecture
MFC	Microsoft Foundation Classes
MOF	Meta Object Facility

MVC	Model View Controller
OCL	Object Constraint Language
OMG	Object Management Group
OO	Object-Orientation, Object-Oriented
OOA	Object-Oriented Analysis
OOAD	Object-Oriented Analysis and Design
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOSE	Object-Oriented Software Engineering
OPM	Object Process Methodology
OS	Operating System
PIM	Platform Independent Model
PSM	Platform Specific Model
SQL	Structured Query Language
RFP	Request For Process
RMI	Remote Method Invocation
RUP	Rational Unified Process
UML	Unified Modeling Language
UP	Unified Process
VCS	Version Control System
VM	Virtual Machine
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	Extreme Programming
xUML	Executable Unified Modeling Language

Chapter 1

Introduction

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 involved¹.

The modeling language which attempts to *unify* all the pre-existing approaches to a modeling language for computer science is named the *Unified Modeling 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*².

Finally the question has arisen 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 provides³. This step of abstraction would mean that the architect does not have to be

¹For the process of understanding a drawing, both the right and the left cerebral hemisphere are used. This is a fact well known in the science of *human cognition*.

²XP, against common knowledge, is often coupled with some modelling. Important is a high *agility* of this modelling; it should help the developer to *travel light*, as explained in [Amb02]. A good introduction to development processes is given in [Bal00, p. 214], although in the German language.

³Critics of this step's probability compare it with the hype that was generated when the first CASE

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 [GHJV94]. Two ways have been proposed to achieve this goal⁴.

Interaction diagrams: Using the interaction diagrams of UML, *sequence diagram* and *collaboration diagram*, method bodies can be generated⁵. The static structure of a software system can be generated by UML tools and provides a framework for these method bodies.

State chart diagrams: The state chart diagrams of UML can be used to automatically generate code that is executable. This execution generally happens on the basis of a generic state machine. Interaction diagrams play a less important role in the suggested proposals based on this approach⁶.

tools came to market in the 1980's and data modelling was proposed as a basic layer for automatically generating a system [Amb02], nevertheless, the current functionality of some database systems and their corresponding development tools, e.g. those provided by Oracle, come quite close to these original promises.

⁴In [Lar02, p. 444], Larman advocates to use *state chart diagrams* only for the purpose of illustrating external and temporal events and the object's reaction to them. He himself excludes from this rule the possibilities of applying a code compiler to the state model and generating code, as obviously is the case with xUML. Still, he argues that on a large scale, the use of interaction diagrams and state chart diagrams is equivalent - both show interaction between objects and the objects' reaction to them, from his point of view using interaction diagrams is in most cases more advisable if the *code generation possibility* is neglected.

⁵A tool providing this functionality today is *ControlCenter* by *Together*.

⁶The most prominent proponent of a state chart based code generation is Stephen J. Mellor who also defined a profile called xUML (executable UML), explained in section 2.3.

1.1 Motivation

The motivation of this thesis is to evaluate the current situation of software development with UML. 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 *MathTrainer* system for elementary school students shall be used⁷. This system proposes a platform on which teachers can design exercise types and students can solve exams automatically created from these exercise types. The example is based on the requirement text shown in figure 1.1.

The Requirements of MathTrainer

MathTrainer aids in perfecting the mental arithmetic of elementary school students.

MathTrainer 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 MathTrainer.

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 MathTrainer 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 students which are assigned to them.

During the realization of a test (ten exercises) the time needed for each exercise is stopped. However, the scores are based on the cumulative time.

Figure 1.1: The requirements for the MathTrainer - example

Modelling this *MathTrainer* seems to be easy if the description provided in section 3.3

⁷This example is proposed by [HK99] as an additional exercise.

is 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.

1.2 Goals

This thesis aims to implement the *MathTrainer* 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 xUML *state diagrams* (for an explanation of xUML, see section 2.3). 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

The software development life-cycle is the sequence of tasks occurring in software development. This general sequence can further be specified by choosing a *process* and a *method*. The software developer can design the process according to many different process models, some of them explained in the following.

Waterfall model: Segmenting the life-cycle into distinct stages (some of them being analysis of system feasibility, elaborating software plans, product design, coding and integration), it emphasizes fully elaborated documents after each of these. To limit expensive rework, the possibility of feedback is confined to successive

stages. When the requirements change during the development process, the waterfall model can lead to unusable code or even discarded projects [Boe88].

Spiral model: To sidestep the problems caused by the waterfall model, a cyclic approach is suggested in the spiral model. The developer repeatedly advances through several stages (determining objectives and alternatives, evaluating the alternatives, developing and verifying and planning the next phase), in each cycle functionality is added to the system. The spiral model provides a risk driven approach to software development outperforming the waterfall model in many situations, but can be used much like the waterfall model in others [Boe88].

Rational Unified Process Model: Based on the Unified Process (UP), this is a special case of an iterative or spiral model. It goes through the steps inception, elaboration, construction and transition, in each step, several iterations take place. Essential elements of the RUP are to plan only for the next iteration and the expectation that rework will be necessary (the waterfall model, in comparison, denies the necessity of rework). As the process used in this thesis is a close sibling to the RUP, more details will be given in 3.1.

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?” [⇒OOAD-Roadmap]. Hence, OOA cares about the real world and how to model this real world without getting into much detail. Larman describes in [Lar02] 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 it’s responsibilities?”, “How to ensure that this class knows all the information it needs?” and “How will classes in the design communicate?” [⇒OOAD-Roadmap]. 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 ([Lar02]):

“Do the right thing (*analysis*), and do the thing right (*design*)”.

As for the OOA, it has itself two main steps: the definition of the use cases and the definition of the domain model, both are parts of the requirement analysis. In the definition of the use cases the focus lies on written stories describing domain related processes. This step is often not in the scope of the developer who is not yet involved in the process. By the definition of the domain model the domain is described by providing a classification of objects used in this domain. Important concepts, attributes and associations are identified and expressed in the domain model.

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.

[Kai99] 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 2.1 visualizes these dependencies.

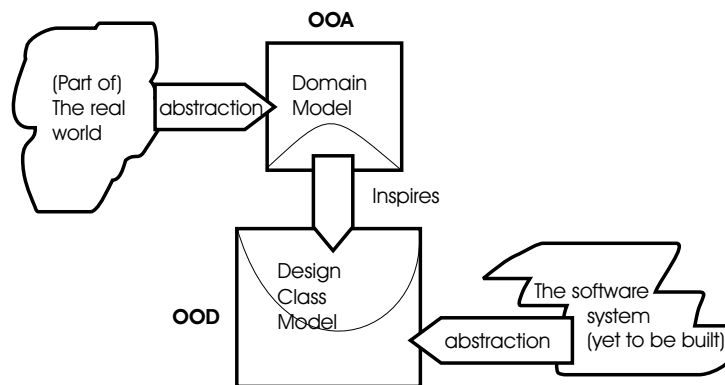


Figure 2.1: The phases of OOAD and how the transition from OOA to OOD works – based on the insights of [Kai99].

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 [Kai99]. 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

are sent. The interaction diagram's concern is the dynamical dataflow which occurs between the objects. On the other hand, the *design class diagram* is necessary to show the static structure of the classes. Unlike the domain model - which was created in the analysis phase - its components are near to actual software-classes, with attributes and methods which are drawn from the interaction diagrams.

2.2 The Unified Modeling Language

Over the years, software development has greatly increased in complexity. A typical project in the software industry can consist of huge amounts of code and a tremendous set of interfaces to other software. Due to this high complexity, efforts have been taken to structure the *process of software development*. Process models were developed to aid in determining and sequencing the necessary steps of software engineering. The higher complexity also called for graphical notations to visualize the engineering efforts.

In the 1980's, several modelling languages had evolved claiming to aid the developer in Object-Oriented Software Engineering. The most important were:

1. The Booch method by Grady Booch, being very close to programming.
2. The Object-Oriented Software Engineering (OOSE) method by Ivar Jacobson, being very user oriented and related to the telecommunication sector.
3. The Object-Modeling Technique (OMT) by James Rumbaugh, very much relying on data modelling.

The period termed *method wars* was ended by the joining of forces of these three outstanding methodologists of software engineering. They unified their own approaches to software modelling in the rules and definitions of UML [Kob99], today being the standard for building object-oriented software systems. The UML is widely adopted and accepted both with software developers using the notation and tool vendors providing aid in the software development process. The OMG maintains a list of available tools helping the software engineer using UML [\Rightarrow UML-Tools] – this list reflects the high reputation of UML as a modelling language.

Figure 2.2 displays the development of UML over the years, currently the version 1.5 is the OMG's accepted standard.

The UML version 1.5 consists of diagram types enabling the developer to express her software building plans in manifold ways. Duplicity is inherent in these diagram types -

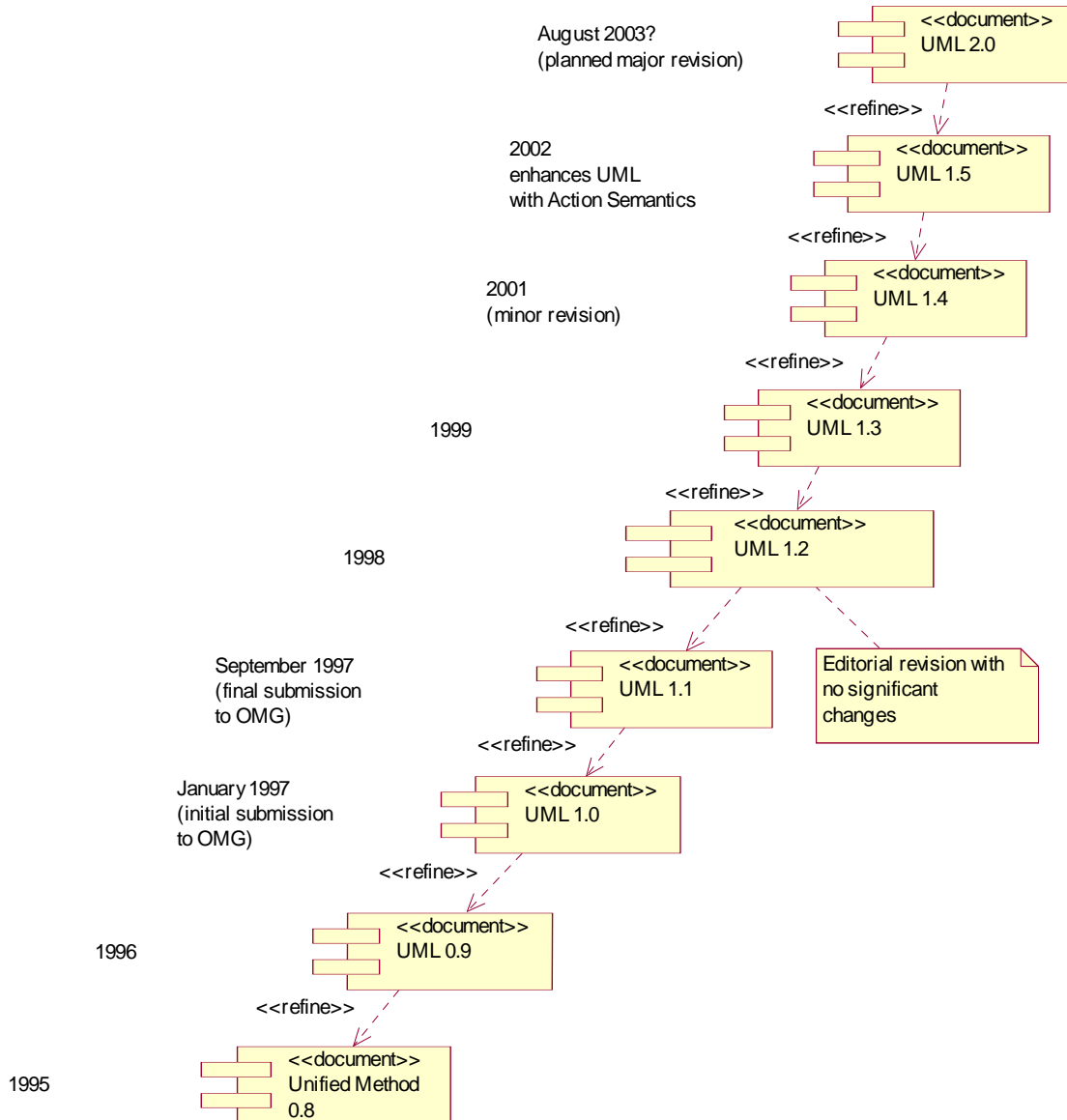


Figure 2.2: The origin and descent of UML, taken from [Kob99] and extended with the latest developments.

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 [HK99, UML03].

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, where the nodes are processors and their attached components and the connection of these elements in form of a network are the graph's edges.

The constructs of UML - like *class*, *association* or *attribute* - are defined in three packages, their interaction is shown in figure 2.3. Each of these packages consists of subpackages and/or classes, the detailed description of which lies beyond the scope of this introduction. [UML03] provides all the details.

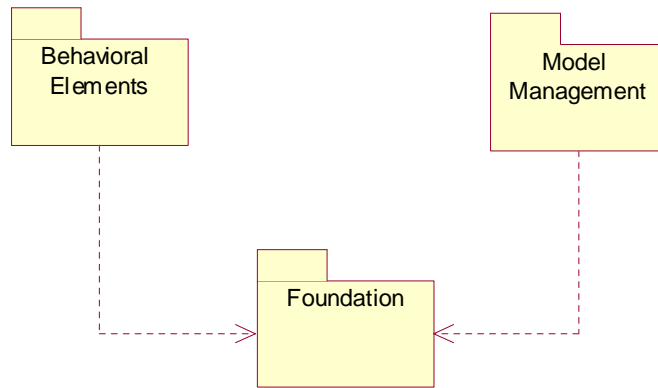


Figure 2.3: The packages of UML and their interdependencies [UML03].

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, according to [Kob99]:

- *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).

[Mel02] 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.

The UML2-Partners (U2P) Proposal: Evolution, not Revolution. What [SRK02] rank the highest, is a gradual evolution of UML and not a revolution to add a multitude of new and changed features – this is to secure the investments of the user base (both tool vendors and the individual developers), in the best case they should be able to *migrate* to UML 2.0 without even noticing. 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 infrastructure². 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 systems³ and more possibilities for modelling complex behaviour⁴ and a *formal graphical syntax and diagram interchange* on top of the current “model element” interchange format XMI.

¹This must include its runtime semantics to support the Model Driven Architecture (MDA) approach of the OMG.

²The infrastructure consists of UML abstractions like *class*, *association* or *Instance*

³This is necessary for modelling software architectures as for example CORBA components, EJB or .NET.

⁴This includes removing the tight constraints on the activity graphs and adding the possibility to model behaviour specifications in a hierarchical way.

The Distributed Systems Technology Center (DSTC) Proposal: UML2 must enable a family of languages. [Dud02] argues that with the UML based on the MOF in version 2.0 (which is mandated in the “Request For Process” (RFP) for UML 2.0) there is no reason for profiles anymore – instead the metamodel should define the basics for any modelling language and meta modelers should be able to add abstract syntaxes for new languages and the concrete graphical realization of them using the UML2 superstructure. So the different members of the *family of languages* could share semantics, structure, notation and - maybe the most important fact - the toolset available on the market. The process of adapting UML to new requirements of the market will be easier and faster to achieve with this approach.

The 2U Proposal: Make Models be Assets. In [Mel02], Mellor mandates executable, translatable and freely interchangeable models – 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 platform⁵.

The 3C Proposal: Be Clear, Clean and Concise. 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 [FT02].

The Object Process Methodology (OPM) Proposal: Why significant change is unlikely. [Dor02] discusses the problems of UML regarding simplicity. Dori states that the syntactical wealth of the various diagrams and far over a hundred concepts of

⁵This is comparable to what the OMG itself advocates in its model driven architecture (MDA) approach.

⁶These fifteen are individuals (objects, actions and associations), abstraction (combination and view), type, specification, role, template, time, place, possibility and change.

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.

Evaluation of the Proposals

According to [Mil02], three of the groups having proposed for the UML 2.0 specification have started two work together, namely the groups U2P, 2U and 3C.

The result might be the adoption of the outcome of this cooperation. So in the best case UML 2.0 will incorporate remedies to the biggest problems of the last time, will have a solid foundation using just fifteen primitive concepts and will be executable in a platform independent way (as is xUML already today). Part of the DSTC proposal is already mentioned in the U2P draft – enabling a family of language is an OMG’s mandatory request for the UML2 specification anyway, so this concept will be included. The revolutionary change suggested by the last proposal, made by [Dor02], means both developers and the tool industry would have to start from the base again.

2.3 The Executable Unified Modeling Language Profile

Executable UML is a profile⁷ 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* [MB02, MW99]. As [⇒xUML02, p. 9] mentions, 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 2.4). With

⁷A profile is a certain subset of the syntax of UML that specifies well-formedness rules beyond those in this subset (often in OCL), adds standard elements to the subset and specifies additional semantics in natural language. A profile can be both officially adopted by the OMG or can be an unofficial publication of independents, the latter is (still) the case with xUML.

the adoption of UML version 1.5 the situation has changed as the action semantics are now part 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.

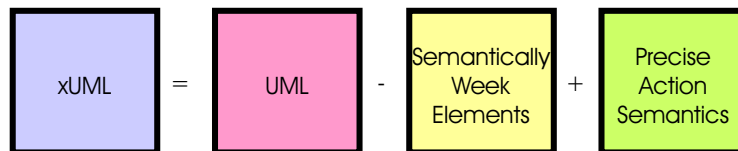


Figure 2.4: xUML is a subset and a superset of UML as of version 1.4 [\Rightarrow xUML02, p. 9].

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. The choice of programming language and platform is just restricted by the availability of model compilers for the target environment (or by the ability of the developer to write the model compiler he needs). Arguably, writing a model compiler for xUML is not easy, but the future will make these tools available for the most purposes ([MB02]), as can be seen in table 4.1.

The concepts used in an xUML model are, according to [MB02, p. 6], data and control elements as well as algorithms. An overview is provided in table 2.1.

Concept	Called	Modelled As	Expressed As
the world is full of things	data	classes attributes associations constraints	UML class diagram
things have lifecycles	control	states events transitions procedures	UML statechart diagram
things do things at each stage	algorithm	actions	action language

Table 2.1: The elements of xUML [MB02, p. 6]

2.3.1 The Maturity of xUML

One of the hardest critics of xUML is Ambler in [Amb02, p. 172]. He argues that xUML is far too early on the market to be of any use to today’s programmers. Additionally, he compares the beginning hype around this approach to the hype that accompanied the creation of CASE tools in the 1980’s. He bases this assumption on the lack of *user interface modelling* and *data modelling* in UML, being represented in xUML as well. So his advice is not to use xUML as of today. On the other hand, he is suggesting to use

UML in every project when the UML syntax can be applied to the problem without having to tweak it. According to him, for user interface description and data modelling other modelling languages should be used – even though there are proposals for a data modelling in UML like in [NM01] which are not yet standardized.

In his foreword to [MB02], Ivar Jacobson argues that the final goal of UML is to reach a state where it can replace nowadays high level programming languages. He predicted such a change far in the future (talking about 25 years from now) but with the progress xUML has made in the last time the goal could – according to him – be reached much earlier for at least a specified subset of software systems. Even if this might be the biased point of view of someone on staff in the leading UML toolset provider it is still clear that the direction of development will work for xUML as it provides us with a possibility to go to a higher level of abstraction; and this is what has happened since the beginnings of the software development.

2.3.2 Modelling in xUML

Executable UML tries to abstract from both specific programming languages and the decisions about how the software should be organized. An xUML 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 and
- *tested* by generating and running test cases for the model [Mel02].

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.

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...*”[MB02] 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 [MB02]. 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.

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.

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 [MB02].

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 [MB02]. 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 to tell 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 [Sta02, p. 141]. 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 3.5.2.

An additional artefact showing classes interacting with each other is the dynamical collaboration diagram - in [MB02] 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.

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

⁸n-ary associations are relationships in UML where more then three classes take part [UML03].

comprised of states, transitions and actions that are executed upon entry to a state. Examples for state diagrams will be given in section 3.5.2. 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 [UML03]. 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. Examples for this ASL syntax can again be found in section 3.5.2.

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 postconditions) [MB02]. 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*, as is defined in section 3.5.1.

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.

2.4 The Java Technology

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 [GM96]. 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-

systems⁹. 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*¹⁰.

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 – as can be seen in figure 2.5.

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 [CWH01]. 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.

⁹Any exception of this rule is generally considered to be a bug in Java. However, there is platform specific behaviour which is clearly marked as such in the documentation (e.g. the `System.exec()` functions - usage of which should therefore be avoided).

¹⁰Nowadays, Just-in-time (JIT) and hotspot interpreters are common when using Java. They help making the process of interpretation being a lot faster, not far off the direct execution of programs.

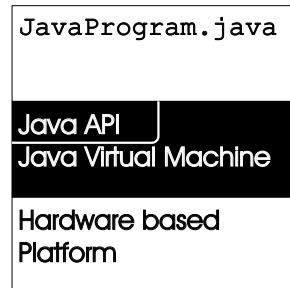


Figure 2.5: The Java platform consisting of Java VM and Java API [CWH01].

2.5 Patterns for Object-Oriented Software Engineering

A design pattern provides a solution for a problem that has occurred over and over in a certain field [GHJV94] – these fields can be as far apart as software engineering and architecture. Hence, when the problem is encountered another time, using a predefined pattern provides the solution much faster. The pattern facilitates *reuse of knowledge*. Additionally, the pattern helps in the communication among software developers – using the name of the pattern conveys a large amount of knowledge in a very dense way. A pattern is usually described in four components; these components explain what it is about and how it is to be used.

Pattern name: The pattern name is used to identify the pattern once it has been introduced. It is a way to communicate the pattern to other people and is therefore vital in spreading its reach, this is a fact mentioned in [GHJV94].

Problem description: In this section, the problem is described that is the reason for applying the pattern. It may be accompanied by a list of preconditions that must be fulfilled – only when these conditions are met, the pattern is applied.

Solution: Here the workings of the pattern are explained: which classes interact when and how these interactions are achieved. The description is on an abstract level to make sure that it can be applied in many situations [GHJV94].

Consequences: Here the effects of the pattern are explained. This might be both advantages and disadvantages of applying the pattern. The consequences are often related to the impact on flexibility, extensibility and portability the application of the pattern has [GHJV94].

Patterns can be used in any stage of software engineering, their applicability ranges from the analysis to the implementation phase. Examples for patterns will be given later, especially in the chapters 3 and 4. A discussion often arising with patterns is at what level of included knowledge a construct can be called a pattern. What is a pattern to one person, is a “primitive building block” ([GHJV94]) to another. However, it can not be considered bad practice to use the name *pattern* for any principle in software-engineering, as the developers who do not need and do not want to cope with that pattern can easily refrain from doing so. Patterns can be categorized, one way of achieving this categorization is provided by [BMR⁺96].

Architectural patterns: This type of patterns copes with the design at a high level and is used early on in the development process. An example is the *Layers* pattern, structuring the system into layers.

Design patterns: These are useful in the detailed design of the system, on a small to medium level. Examples are the *Facade* pattern, providing the interface from one layer of the system to the next.

Idioms: Idioms are used on a low level, and are very much oriented to the implementation of the system. An example is the *Enum* pattern for transferring the C++ construct of an enumeration to Java.

Chapter 3

Object-Oriented Analysis and Design Phase

Object-Oriented Analysis and Design (OOAD) cope with the analysis of the provided requirements and the subsequent design of a software system meeting the needs of the requirements. As this phase in software development is structured into a sequence of steps, the overall process providing the framework for this sequence shall be explained first. Next, the OOAD tools used in this thesis shall be introduced before the Object-Oriented Analysis and Object-Oriented Design steps are exemplified on the *MathTrainer* system. As a guideline for what to do in the distinct OOAD steps the recommendations by [Lar02] are followed.

3.1 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. The RUP is distributed as a collection of documents mainly in the HTML

format, including templates for the major artefacts being the outcome of applying the RUP.

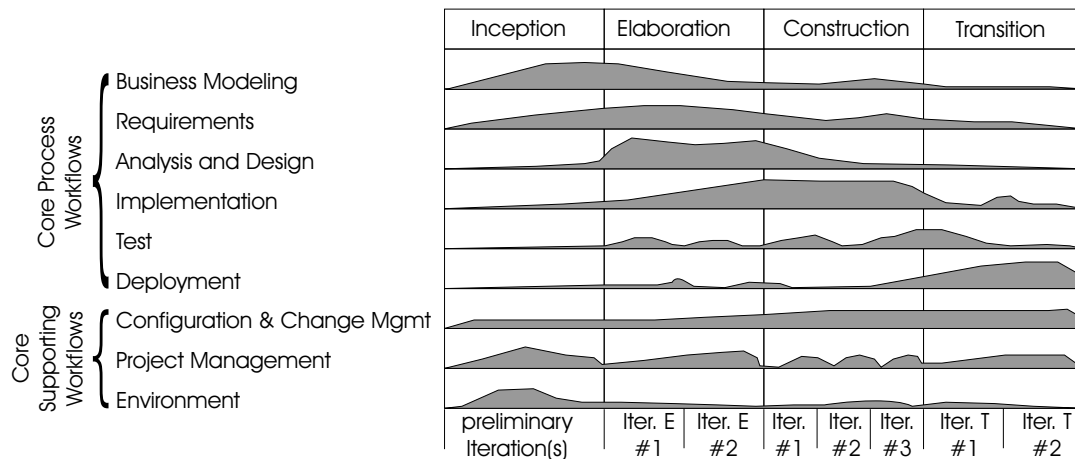


Figure 3.1: The phases of the Rational Unified Process, based on [Ros01].

The RUP is separated into four distinct phases, these being inception, elaboration, construction and transition. In each of these phases, a certain amount of effort is spent on the basic workflows business modelling, requirements, analysis and design, implementation, test, deployment, configuration and change management, project management and environment. However, the effort necessary for the different workflows changes over the lifetime of the project, as indicated in figure 3.1. For instance, the effort spent on deployment is very low in the beginning and grows to a large extent in the final part of the transition phase – as deployment is not a major workload in the beginning of the project. Hence, figure 3.1 provides the outline for planning a development project and deciding about what has to be done at which time. Fine planning is only done for one iteration in advance ([Ros01]), however, the overall time span of the project should be planned early. For each of the workflows represented in figure 3.1, the steps to be taken for the *MathTrainer* example are outlined on the base of the RUP.

Business Modelling: In this early stage, the current business is assessed as it is the environment of the software system in development. For the *MathTrainer* 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, as explained in section 3.4.1.

Analysis and Design: This workflow begins with planning the software architecture. For the *MathTrainer* 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 *MathTrainer* 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 *MathTrainer* are 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 and distributing it, or to provide a web-location for downloading the application. As the *MathTrainer* 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 *MathTrainer* 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 *MathTrainer* 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.

3.2 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 *Together ControlCenter*, mainly used in the OOD. Finally, the xUML tool *iUML* of Kennedy Carter is presented.

3.2.1 Rational Rose 2002

Rational Rose 2002 is one of the most widely used UML tools. The popularity of Rational Rose can be explained by Rational being the employer of the “three amigos”, Grady, Booch and Jacobson, and by Rational keeping close to the UML standard. Still, even Rational Rose does not support all UML constructs, for instance, conditional logic on sequence diagrams is not supported by the tool except by using notes.

The user interface of Rational Rose is easy to use, even though the complexity of the tool is sometimes overwhelming. Code generation from Rose is possible from static diagrams and to many target languages, with add-ins the functionality can be amended. Code is written outside of Rational Rose, but round-trip engineering subsequently updates model and source code, respectively. Rational Rose can cope with multiple users working on a model at the same time, but this functionality is not included in the standard tool. A version management tool like Rational ClearCase has to be purchased separately.

3.2.2 Together ControlCenter

Together ControlCenter has evolved from TogetherJ, providing more languages than its predecessor. The tool is sold by Borland; the inventor of TogetherJ is Peter Coad, well known in the field of OOP. Together ControlCenter is keeping as close to the UML standard as the Rational tool, but provides some additional constructs that can not be modelled with Rose. For instance, conditional logic is supported for sequence diagrams.

The user interface of Together ControlCenter is well structured and easy to use. An integrated development environment provides a way to simultaneously model and code without frictions between these two actions, round-trip engineering instantly updates the other view. Code generation is possible from static and dynamic diagrams and to many target languages. Still, automatic and instant round-trip engineering is only

available for static diagrams, for interaction diagrams this process has to be started manually. Together ControlCenter works with major Version Control Systems (VCS), also the freely available Concurrent Versions System (CVS), which enables multiple developers to work on a model.

3.2.3 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.

3.3 The Requirements

To get familiar with the example, first it is necessary to look at the *requirements* in figure 1.1. These are no formal, elaborated requirements, but this text might be a first draft of a requirement provided by the management as they declare to which capabilities and conditions the system will have to comply. In the case of the *MathTrainer* 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 ([Ros01]) 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.

The RUP supports the principle of *managing requirements*. This principle is contradictory to the waterfall attempt of having fully stabilized requirements before beginning with the development, but refers to constantly finding, evaluating, documenting and tracking the changing requirements ([Ros01])– management of change, in fact.

3.4 The Object-Oriented Analysis and Design Phase using UML

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.

3.4.1 The Object-Oriented Analysis

[Lar02] suggests, as a starting point to developing a system, to write *use cases* – stories about how the actors interact with the system to get value out of it. 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. It is possible to regard the requirements provided in figure 1.1 as one sentence use cases each.

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 [HK99, p. 166]. 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 figure 3.2.

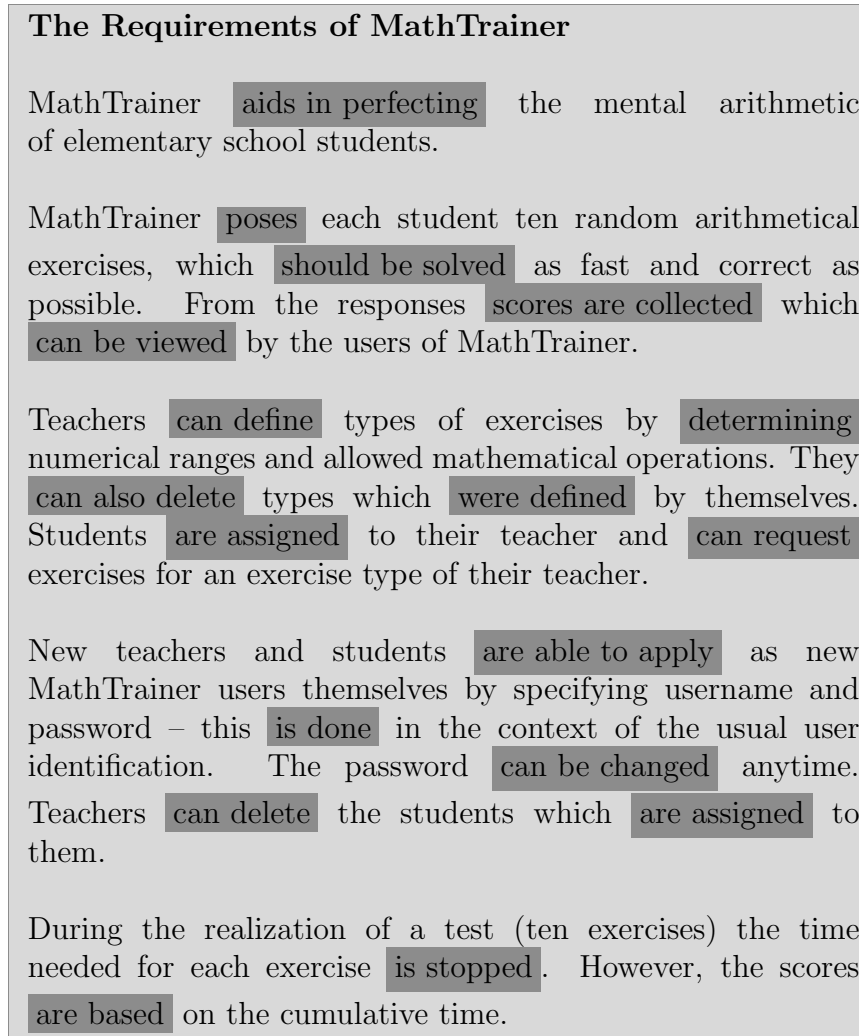


Figure 3.2: Highlighted verbs in the requirements of the MathTrainer - example

From this foundation on, the developer is to write down all verb phrases in a list and decides which verb phrases evaluate to a *use case* of the system. For the MathTrainer - example, this process is shown in table 3.1.

The found use cases may not be complete (for this example, an *Identify User* use

⁰For a discussion of goals, see [Lar02, p. 59]. Larman argues that use cases should be on the level of EBPs (elementary business processes). A common mistake is defining use cases at much too low levels, e.g. *adding a line item* instead of *taking an order*.

<i>Evaluation of the verb phrases</i>	
aids in perfecting	This is a mission statement - not a use case, it is too high in its goal to be one ² .
poses	Here the first use case candidate is found - MathTrainer poses student arithmetical exercises.
should be solved	The <i>student</i> solves exercises – this seems to be very close to the previous use case, so just one of these two use cases is necessary. The use case is chosen where the user takes the active step, so this one.
are collected	The <i>system</i> creates scores from the responses. If the system is doing some internal work, this work should not be reported as a use case. Use cases are about a black box view of the system.
can be viewed	The A user can view scores.
can define	Teachers can define exercise types.
determining	Teachers define exercise types by determining numerical ranges and allowed mathematical operations – this fact is refining and explaining the previous use case into more detail.
can also delete	Teachers can delete exercise types.
were defined	This is not a use case, rather a constraint on the previous one - only teachers defining an exercise type may delete this exercise type.
are assigned	A student is assigned to a teacher.
can request	A student can request exercises.
are able to apply	A user can apply as a new user providing username and password.
is done	This is a statement about where the above use case has to happen.
can be changed	A user can change the password anytime.
can delete	A teacher can delete his students.
are assigned	Again just a <i>constraint</i> for the previous use case.
is stopped	The <i>system</i> stops the time for each exercise. Again, reporting this fact as a use case would not be in compliance with the black box view.

<i>continued from previous page</i>	
are based	The score is based on the cumulative time.

Table 3.1: The evaluation of the verb phrases which were identified in figure 3.2

case has not yet been found but might be important to cope with the restrictions of user identification¹. 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 stand alone 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 MathTrainer system is shown in figure 3.3.

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 3.3 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. This assumption does not hold true for the use case diagrams in UML: the problem is the vague specification of the *extend* and *include* relationships.

¹[Lar02] argues against a user identification use case as it does not add any real business value. This being true, security is still something that is often very important to the business at a higher level and if this principle is not obeyed in the analysis and design phase, it is very hard to add it later on by changing the structure of the system (see the struggles of Microsoft in changing their policies to respect security to a higher extent).

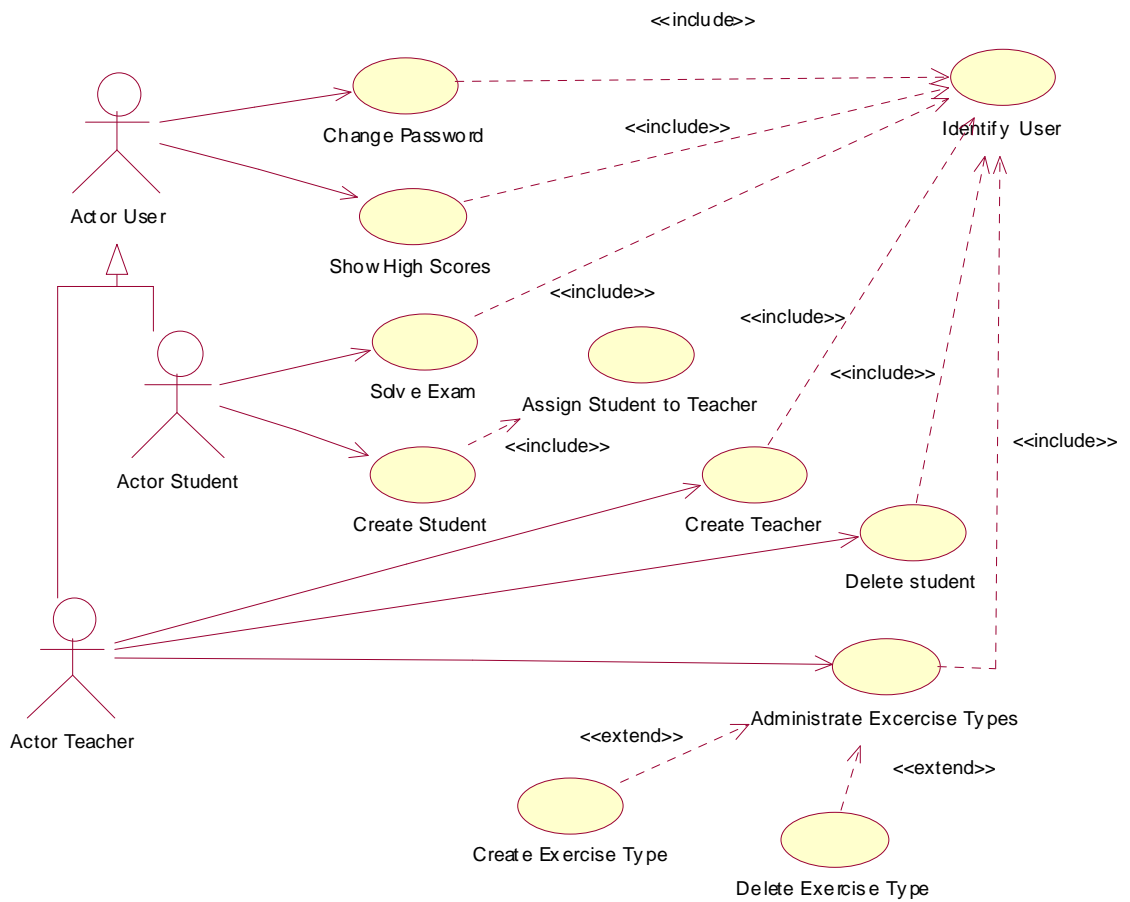


Figure 3.3: The use case diagram for the MathTrainer example.

According to [Qua03, p. 35] the include relationship is used whenever use cases share the same functionality. This functionality is then drawn out to a separate use case and connected to the use cases including its functionality with an *include* relationship. In figure 3.3 the *Identify User* use case is included in many of the others.

The *extend* relationship is used whenever optional behaviour or conditioned behaviour (e.g. behaviour which is only executed when a certain alarm has triggered) occur. Additionally, the *extend* relationship can be exploited to handle the case of an actor choosing one of several different flows of a use case. In figure 3.3 this behaviour can be seen with the *Administer Exercise Types* use case is extended by either a *Create Exercise Type* or *Delete Exercise Type* use case, depending on the choice of the actor *Teacher*.

Basing on the requirements provided for the *MathTrainer* example, the use cases are now converted into *written stories*. These stories explain what happens for the special use case under which circumstances and have a standardized layout. The use case for creating a student is presented in table 3.2.

To show the work-flow of a use case, an activity diagram can be used. However, it is not a replacement to the written stories accompanying a use case. An example for an *activity diagram* is shown in figure 3.4.

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*³.

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.

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. As a proposal, [Lar02, p. 138] suggests the creation of a conceptual

³As a second way, [Lar02] proposes the usage of a *conceptual class category* list where categories of objects like places, transactions, roles of people, events, organizations etc. are distinguished. For each of these categories classes are identified, e.g. by a brainstorming process.

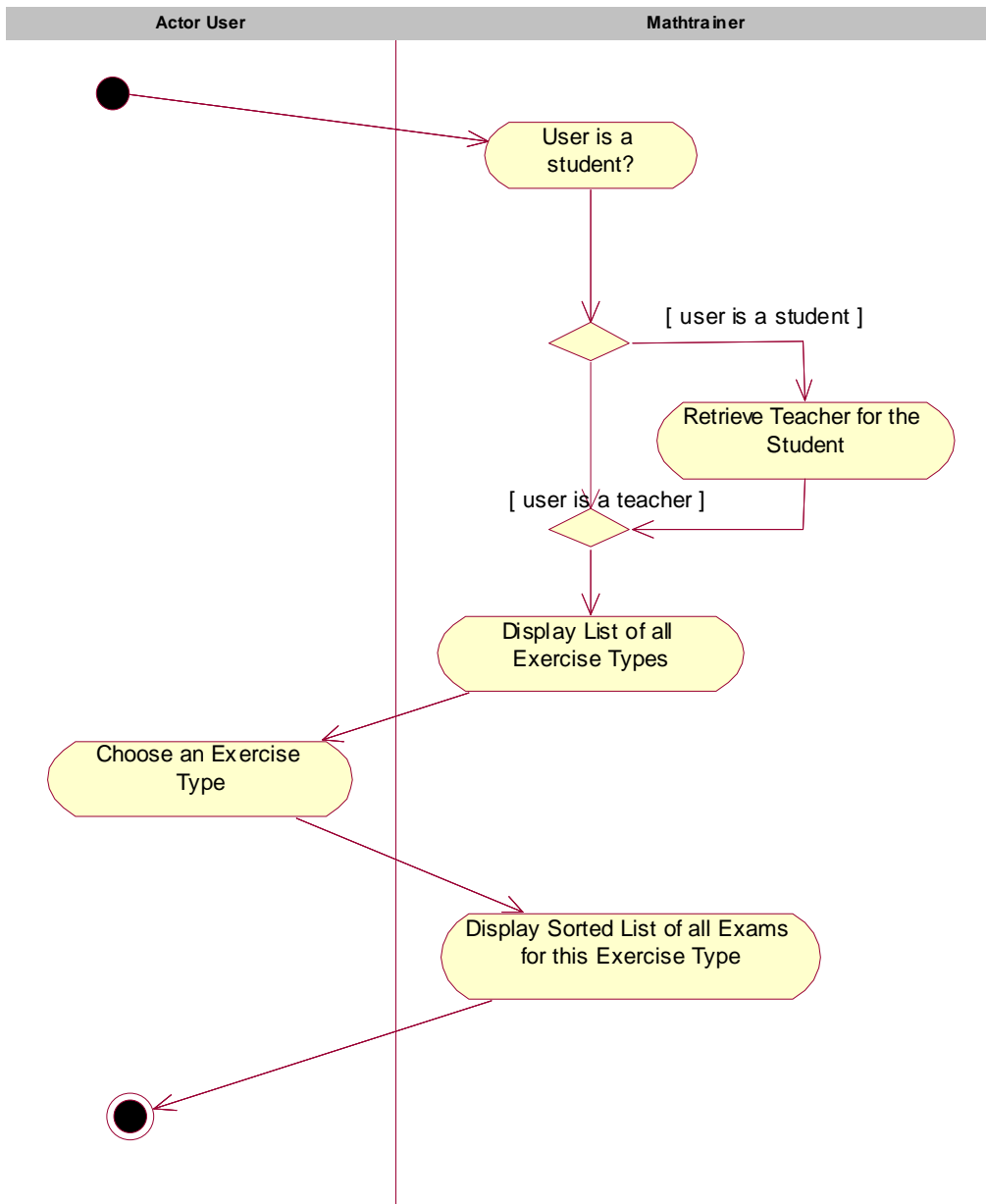


Figure 3.4: The *activity diagram* MathTrainer's *View Scores* use case.

Description	A student can be created by any user of the <i>MathTrainer</i> system. Necessary information are username and password which has to be entered twice. Additionally a teacher must be chosen.
Precondition	At least a teacher is known to the system. The user has access to the <i>Login</i> context of the <i>MathTrainer</i> system.
Postcondition	A new student has been created. The student possesses a unique username and a password and is associated to an existing teacher.
Error conditions	<ol style="list-style-type: none"> 1. The provided username is already given to a user of the <i>MathTrainer</i> system. 2. The entered passwords do not match.
Error postcondition	The student was not created.
Actors	User (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Username and password are entered, password twice. 2. The Teacher is chosen. 3. The username is not yet used by the system and the entered passwords match. 4. The student is created in the system. 5. The student is assigned to the chosen teacher.
Deviation 1	<ol style="list-style-type: none"> 3'. The username is already in use or the entered passwords do not match. 4'. An error is shown to the user and all system objects retain their state. 5'. The user is directed to the <i>Login</i> context.

Table 3.2: The written story for the *Create Student* use case.

class when in doubt. Doubt arises, when the noun in question is not merely a text or a number. When it is not just a number or text, it is intended to be a conceptual class of its own, even if it does not have any attributes. Another very common problem, which has not been arising in this example, is to forget about *specification or description classes*. These classes are necessary to describe other classes when crucial information is lost when all instances of the other class are deleted.

Imagine a university having *Lectures*, and each of them having an association with a *Professor*. Now, the summer holidays start and as no lectures are given anymore, the lecture-instances are deleted from the system. Also the information about which professor can hold which lecture is lost. So there has to be a *specification class* which tells us about which professor can hold which lecture.

<i>Evaluation of the nouns</i>	
--------------------------------	--

<i>continued from previous page</i>	
MathTrainer	This is the name of the system being built. It is generally not included in the domain model – the system is what is to be generated <i>starting off</i> from the domain model, it is not yet there.
mental arithmetic	This abstract noun is not to be excluded as it is abstract (there can be a lot of abstract concepts which have their place in domain modelling, e.g. the notion of “happiness” of a user in a Customer Relationship Management (CRM) system.), but as the sentence where it originated from was seen to be a higher mission statement, not an explanation of a business process the noun is not relevant to the domain model.
elementary school student	Definitely a class to be included in the model, it plays a rule in many use cases.
arithmetical exercise	Again a conceptual class.
response	This looks like a conceptual class, the question is if this should be modelled as an attribute or as a separate class. As the response to an arithmetical exercise is a number, a class is used.
high scores	Again a conceptual class.
user	Definitely a class being used to name the general user of the system.
teacher	Again a conceptual class.
type of exercise	It needs to be determined if the type of the exercise is to be modelled as an attribute or as a separate class. As it has two distinct properties (the next two nouns in the list) it can be seen as a conceptual class.
numerical range	Should be included in the model, the question is once more if it is an attribute or a class, when a range is thought as having limits, it is no plain number or text and should be a class.
mathematical operation	The operation, again, is no plain number or text.

<i>continued from previous page</i>	
username	The username is a text which can be included as an attribute to the user.
password	The password is a text which can be included as an attribute to the user.
context	Is nothing which is relevant for the problem domain.
user identification	Has been identified as a use case and will consist of collaboration of independent classes, so it is not a class of itself
realization	The realization of the test is again a use case (solve the exercises, solve the test) and is not a class of itself.
test	Modelled as a conceptual class.
time	The concept of time is here seen as modular as a number or text, it can be an attribute to the exercise.
cumulative time	Arguably, time is as modular as a number or text, leading to the conclusion of modelling the cumulative time as an attribute of the exam.

Table 3.3: The evaluation of the nouns which were identified in figure 3.5

Again, the linguistic analysis step is far from perfect and does not realistically show all the classes of the domain model – the developer has to do additional reasoning so as not to forget important aspects. When the conceptual classes have been identified, the next step is to *find associations*.

In this step, there no concept of linguistic analysis is suggested, and finding the correct associations is indeed not very easy. For the start, some associations which are very important in most software development projects and which can be useful in any domain model are introduced. These are, according to [Lar02]:

- A is a part of B, either physical or logical.
- A is contained in B, either physical or logical.
- A is recorded, logged or known in B.

The Requirements of MathTrainer

MathTrainer aids in perfecting the mental arithmetic of elementary school students.

MathTrainer 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 MathTrainer.

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 MathTrainer 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.

During the realization of a test (ten exercises) the time needed for each exercise is stopped. However, the scores are based on the cumulative time.

Figure 3.5: Highlighted nouns in the requirements of the MathTrainer - example

With the application of these types many of the associations of MathTrainer's domain model can be found: e.g. an exercise *is contained* in a test or a numerical range *is a part of* an exercise type.

Still, there is no restriction to the vocabulary given above, the more expressive the description of an association is, the better for the system's design. Additionally, it might be interesting to model *is-a* or *kind-of* relations by usage of an inheritance relationship. In the MathTrainer example, this is the case with the user-student and the user-teacher association, shown in 3.7.

There is a trade-off between inheritance and aggregation. Often, both can be used to model a specific association, the decision for one of them depends on other influences. In the MathTrainer example, the decision can clearly be made in the case of the user,

student and teacher relationship. A teacher *is-a* user, a student *is-a* user – an inheritance relationship should be used. While inheritance is very useful in such cases, it can be a problem in others. As an example, the case of a user being either a part-time or a full-time student is suggested. If a developer modelled this relationship in an inheritance, it would be hard to accommodate the fact of a student changing his time schedule. If he was a part-time student before, he would then be a full time student or the other way round. This would not be a problem in itself, but a problem might emerge in the implementation as there are not many programming languages supporting an object able to change its class during the lifetime. Hence, Quatrani mandates in [Qua03, p. 127] to refrain from inheritance when a change of the class is likely. Instead, the author recommends using a specification class as an attribute to the student. This specification class can then be subclassed in an inheritance relationship to accommodate the full-time and part-time specifications. The approach of xUML is here clearly different, as explained in section 2.3.2.

The final step for building the domain model is to identify the *attributes* of the conceptual classes. It is possible to start this analysis by referring to table 3.3 and marking the nouns being excluded from the list of conceptual classes. Additionally, reasoning has to be made about the important attributes by looking into the real world and by transferring the relevant attributes into the domain model.

Usually, the reasoning for the determination of conceptual classes should be reversed for class attributes. Everything that is a simple text or number (time, Boolean value and date may be included in this list) should be an attribute; not a conceptual class or an association. The term *association* has to be mentioned here as a common mistake is to include *referential attributes* in the domain model instead of modelling them as associations.

Still, the readability has to be balanced with the importance of the displayed details, as can be seen when comparing figure 3.6 to figure 3.7. In the first diagram, the choice was made to display the *Number* as a distinct conceptual class, even though it might fulfil the specification necessary for an attribute. The resulting diagram is rather complex to grasp, and does not have a great advantage over the second diagram, which displays numbers as an attribute.

After the domain model is created, the phase of OOA is finished and the phase of OOD can start. For a clarification, *finished* does not mean perfect or anywhere near it, finished means good enough to go ahead to the next step; there will always be the possibility to return to OOA in any of the next iterations of the development process.

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

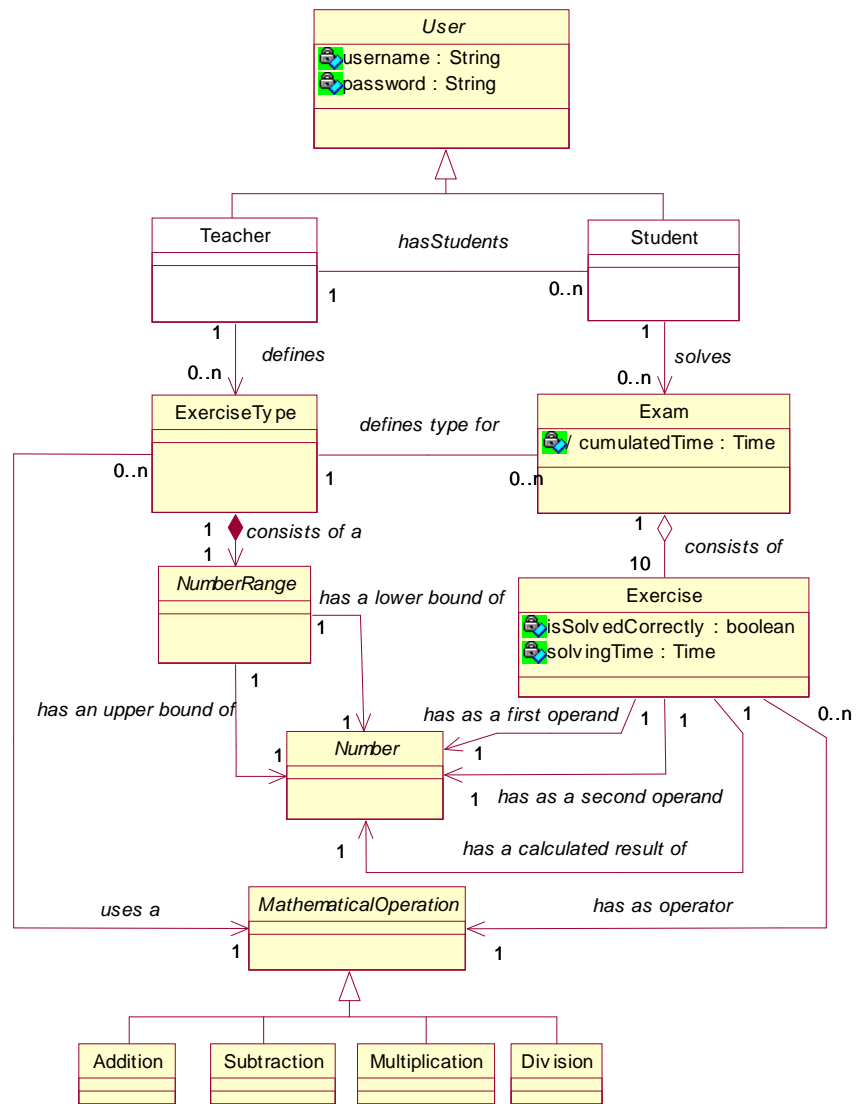


Figure 3.6: The domain model of the MathTrainer example, including numbers as a conceptual class.

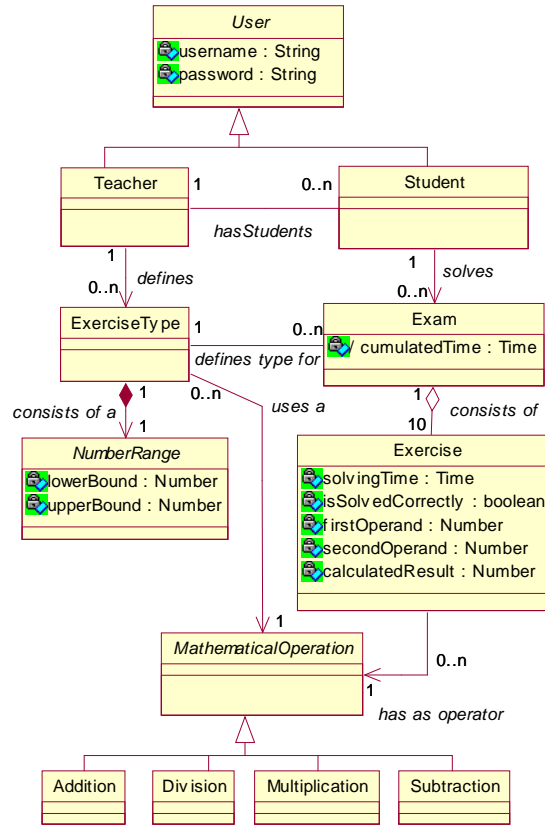


Figure 3.7: The domain model of the MathTrainer example.

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. *Rational Rose* has filtering capabilities which can handle this problem, but still it is preferable to design a distinct model.

3.4.2 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 *MathTrainer*. Complementary to the *design class diagrams*, the *interaction diagrams* are the other main model being built in the OOD phase. The two types of diagrams are not created sequentially, but rather parallel and in an iterative process. The outcome of the class diagram influences the outcome of the subsequent interaction diagrams and vice versa. Of great use in the OOD phase are patterns, as the *reuse of knowledge* they provide is aiding the developer in its attempt to cope with static and dynamic aspects of the software system. Some of the patterns widely used in the Object-Oriented Design shall be presented subsequently.

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 background for understanding patterns theoretically is provided in 2.5.

The Entity-Boundary-Controller Pattern: This very popular pattern has its roots in the Model-View-Controller (MVC) concept. The MVC concept advocates a separation of concerns, leading to a more understandable assignment of responsibilities. According to [Qua03, p. 57], 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 [Qua03], 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 world⁵.

Entity classes: This type of classes is relatively *stable* in its behaviour. These classes often reflect a concept of the real world, so they will often be found in the analysis

⁴The goal of this pattern is *not* to separate the data and the logic of an object. Those should both be included in the object when using OO paradigms. The code being separated is the *sequencing* – so *when* the logic takes place.

⁵Controller classes are seldom classes being inspired by real world behaviour.

phase and already be present in the domain model [Qua03]. Often this type of classes can be reused among applications when these applications have to model the same part of the real world.

Boundary classes: Boundary classes are the interface classes of the system. They are found by evaluating the use cases of the system; typically, each actor-use case pair needs one boundary class. Actors can be persons as well as other systems - in the first case, the boundary class often resembles a GUI, in the second case, the boundary class is often an interface class which defines a special protocol to be used to communicate with the system [Qua03].

Controller classes: Controller classes help with defining and controlling the flow of sequences to fulfil the action of a *use case*. Again, the required classes of this type are found by evaluating the actor-use case pairs – one pair, one controller class⁶. Important about defining controller classes is that they should not take on too many responsibilities [Qua03]. An example: in the domain diagram in figure 3.7 a *Student* of the MathTrainer-system has to be added to its *Teacher* upon creation⁷. In this case, the controller class for the teacher should know *when* to add the student. A possibility to recognize bad design would be if the controller class of the teacher knew *how* to add the new student.

In the basic EBC-pattern the boundary element may only communicate with the controller element, and does not have any links to the entity object. This is necessary to keep the coupling between boundary elements and entity elements low. It is possible to modify the basic pattern by letting the controller pass the entity to the boundary element – the boundary object can then use the methods of the entity element for data entry. Using this approach, the data structure modelled in the entity object can be reused. Still, the coupling is higher in this approach – when the entity object remains highly stable, the high coupling will not be a problem. For the *MathTrainer* example, the modified approach was used, as outlined in [HK99]. 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.

⁶Hence, very often one boundary class has one controller class accompanying it.

⁷As both the teacher knows its students and the students know their corresponding teacher, to avoid misinterpretations – this relationship is changed to a one-way relationship in the design phase.

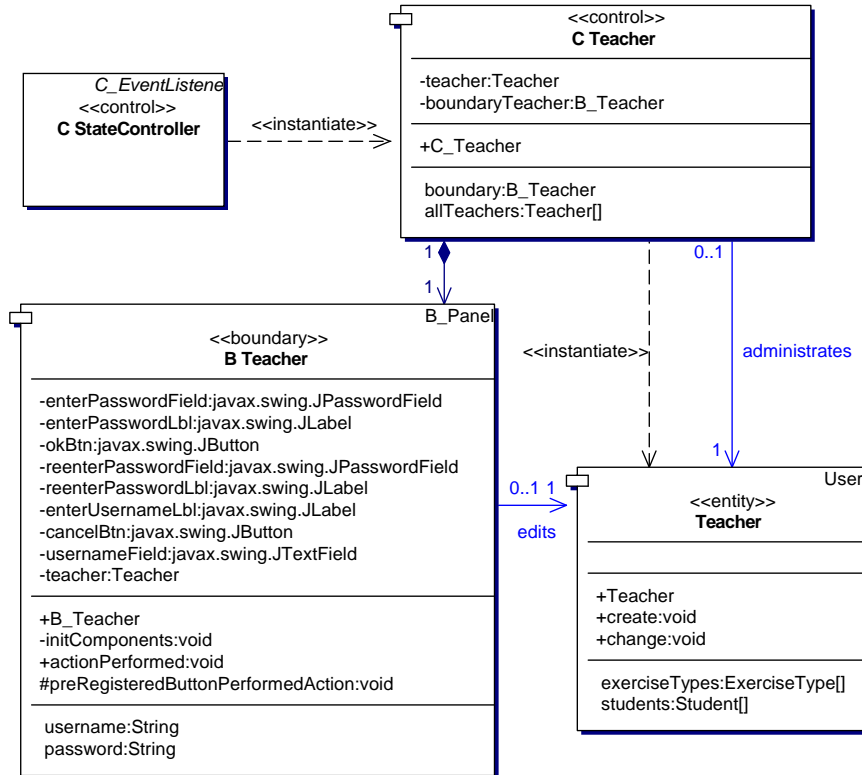


Figure 3.8: The static structure of the modified EBC pattern.

The dynamic behaviour of the modified EBC pattern is shown in figure 3.9 and figure 3.10. 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). Upon data entry the boundary element updates the entity element. Optionally, the boundary element may also tell the entity when it has to store its data persistently. Finally the boundary element is no longer of use and can be destroyed (or stored for a later usage), and either the boundary or the controller element can finish the creation process of the entity element using the necessary instructions – which might be to save the element in a repository.

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, introduced by [Lar02]. GRASP means *General Responsibility Assignment Patterns* and consists of the patterns aiding a developer in handling this important task. In these patterns, re-

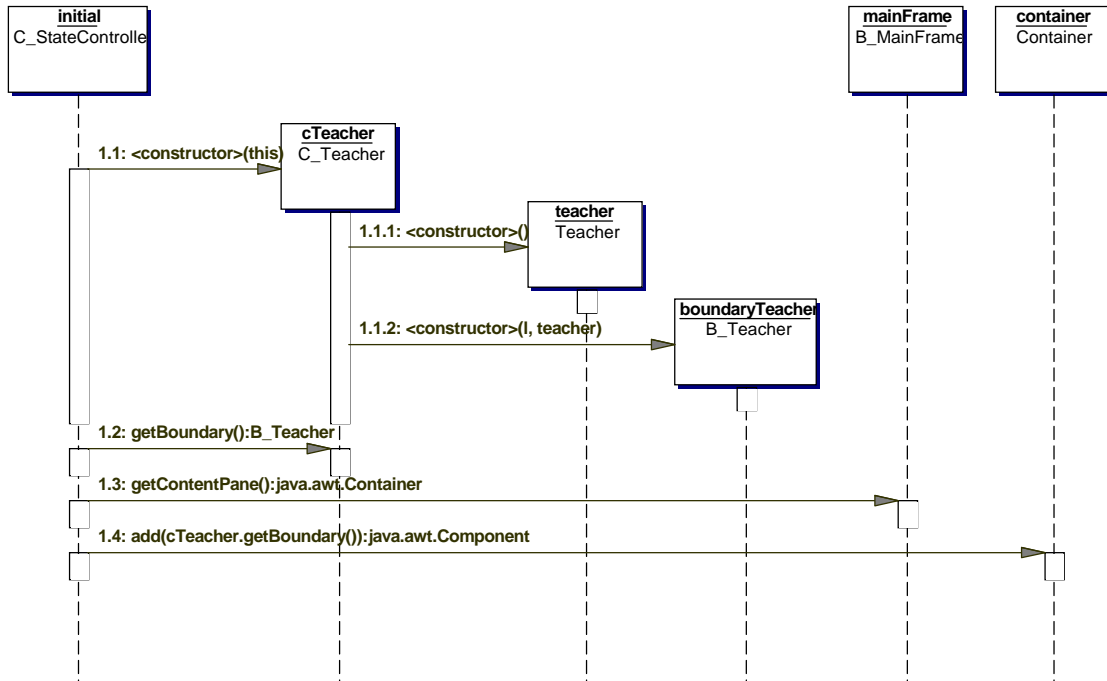


Figure 3.9: The construction phase of the modified EBC pattern’s dynamic behaviour.

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*, but the part of who receives the message can be cleared up by using the five patterns explained below.

Information Expert: This pattern assigns responsibilities to the class that *has the necessary information* to cope with the responsibility.

The information is provided by either the attributes it possesses or by the associations it is connected to. An example: the cumulative time for solving the exam should be provided by the *Exam* class as it is *associated* with all the exercises that each know their solving time. So when the design choice has to be made to add a method `getCumulativeSolvingTime()`, the class *Exam* will receive it.

There are some cases where this pattern should be applied, an example would be

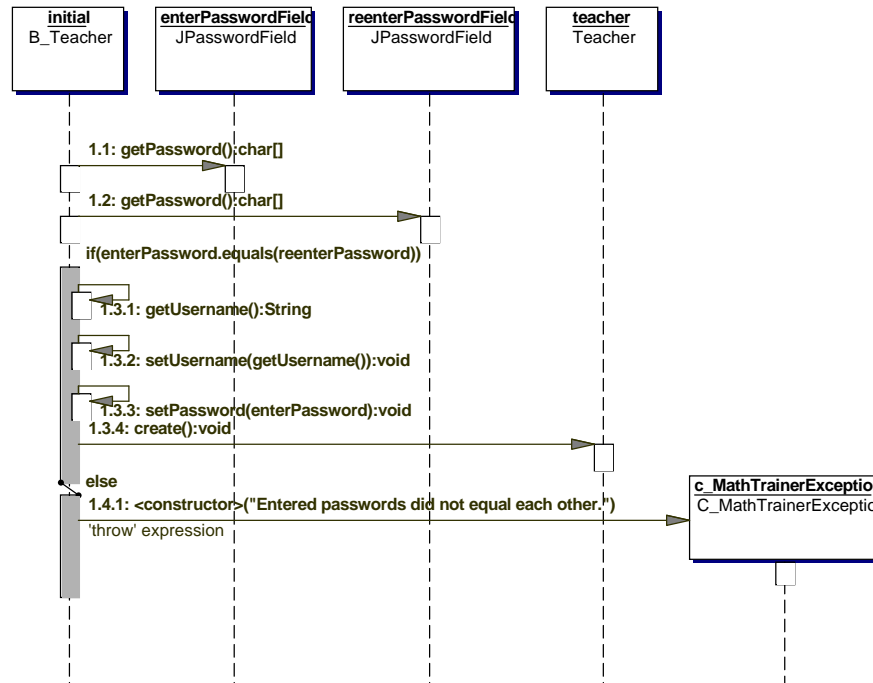


Figure 3.10: The data entry phase of the modified EBC pattern’s dynamic behaviour.

database access; e.g. via Java Database Connectivity (JDBC). In this case, a lot of source code will be duplicated if each class knows how to handle these connections, so it will be harder to maintain and comprehend the code. This fact mandates a *separation of concerns* and separating the connection code to a common place.

Creator: 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*. An example: As the *Exam* class aggregates *Exercise* instances, the exam should be *creator* for the exercise class. Another pattern dealing with creation is the *factory pattern*.

Low Coupling To explain this pattern, first the term coupling has to be defined:

“Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements [...]. These elements include classes, subsystems, systems, and so on.” [Lar02, p. 229]

The design principle of this pattern is to assign the responsibilities so that coupling remains low. This principle can often be broken down to knowledge: when classes can exist without having knowledge of each other, this knowledge should not be added. Low coupling is especially important with unstable objects. It should not be a problem to couple with highly stable business objects which are at the core of an application or with the Java API, but it can be a problem to couple with objects of the GUI which might change anytime.

High Cohesion At first the explanation of the term cohesion is provided:

“Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. These elements include classes, subsystems, and so on.” [Lar02, p. 232]

It is undesirable in OOD and OOP to have classes which have hundreds of methods and attributes - for the sake of maintainability and usability it is necessary to partition the responsibilities of this class functionally and draw them out to different classes.

This problem often arises with *factory classes*, being designed for the creation of objects. When the types of objects get numerous in a system, the factory class can consist of a collection of a hundred and more methods. The class becomes incomprehensible and unmaintainable. The solution is to partition of the factory class, in the same way as the objects being created are partitioned.

Cohesion and coupling often depend on each other. When one is made worse, the other is made worse as well. So to prevent these problems from arising, it is necessary to pay attention to both principles at the same time.

Controller: When handling *system events* being the input from a user or another system, *controller* classes come in handy. These classes have either the responsibility of handling events systemwide (*facade controller*) or handle just one use case in the system (*use case controller*). In the second case, they are often named `<UseCaseName>Handler`.

Controllers themselves are not user interface classes, but they often handle the interaction with the GUI. It is sensible to put in a layer between the GUI and the business objects as the GUI can change anytime due to technical improvements and additional requirements – business objects are rather seldom changed, on the other hand. The responsibilities of the controller are to be the intermediary between GUI objects and business objects and often to provide a sequencing function in determining which actions are executed. In the pure controller pattern, the GUI object may only communicate with the controller object, it is not allowed to access the business object directly. Often, a less purist approach is taken and the GUI object can obtain the business object from the controller object and can directly access its methods for display and data-entry. After it finishes these tasks, it returns the business object to the controller object for the administrative work [HK99]. This approach helps in reducing code overhead as the data model of the business object does not have to be replicated in the boundary object, but it performs worse with respect to *low coupling*. It depends on the stability of the underlying business objects if this approach can be taken, coupling to highly stable objects is generally not a problem.

By the information provided above, the GRASP patterns can be seen as highly interconnected. If a slight change occurs in the assignment of responsibility, all of these patterns are affected. Generally, the better the software development, the better is the overall fulfilment of the principles the GRASP patterns propose.

The Singleton Pattern: This pattern describes a way to create a single instance of a class and to prevent more than one instance from being created. Such behaviour is often necessary with architecture classes that should be instantiated once and only once. Of course, the same effect would be achieved with providing static methods to the class, but this approach is not open to software reuse (as most programming language preclude the user from overriding static methods). Additionally, if once the necessity arises to have more than one instance available, the code can easier be changed when using the singleton pattern.

Java example 3.1 shows the application of the singleton pattern in a factory class, using lazy initialization⁸ of the single instance.

⁸Lazy initialization means that the instance is only created when a request really happens, this helps to save resources and is preferable over immediate initialization.

The Factory Pattern: The *factory pattern* mandates a separation of concerns – business logic elements are there to cope with the business logic, and for the creational logic a new type of element is defined: the factory. The advantages of the introduction of pure fabrication objects, called factories, are:

Separation of concerns: The complex responsibility of creating objects is put into helper classes.

Hidden creation logic: The creation logic is hidden from the other parts of the system, they communicate with the factory only.

Performance enhancements: As the creation logic is hidden, it is possible to introduce performance enhancements, as caching of objects or instances.

An example for the factory pattern can be seen in figure 3.11, where it is used, along with the *Singleton* pattern, to generate random numbers using a `RandomNumberGenerator` class. The factory and the singleton pattern are natural companions as often exactly one instance of the factory has to be created.

```
public class ToolFactory
{
    private static ToolFactory _toolFactory = null;

    private ToolFactory()
    {
        _toolFactory = this;
    }

    public Tool createTool(String name)
    {
        //Create and return a tool depending
        //on the name

        return tool;
    }

    public static ToolFactory instance()
    {
        if(_toolFactory==null)
            ToolFactory();

        return _toolFactory;
    }
}
```

Example 3.1: How the factory pattern could be implemented in Java code.

The Enum Pattern: A very useful pattern in designing programs for the Java programming language is the Enum pattern. It is necessary as Enums are not supported by Java (contrary to C++), even if this construct can be utile in many situations. Often constants are used to get around this restriction, but public constants are far from helping with reusability as they cannot be overridden as well as extended. The basic structure of the Enum pattern is to define a class which has the desired properties as public members. To prevent others from adding properties to the list, the constructor is made private. Example 3.2 provides a colour class with the three properties Red, Green and Blue.

```
public class Color
{
    public static Color RED = new Color("Red");
    public static Color GREEN = new Color("Green");
    public static Color BLUE = new Color("Blue");

    private String _name = null;

    private Color(String name)
    {
        _name = name;
    }

    public String toString()
    {
        return _name;
    }
}
```

Example 3.2: How the Enum pattern could be implemented in Java code.

With all these patterns in place, it is easier to solve complex problems in OOP – still, it is not necessary to know all the patterns to find a good solution to problems arising in this field. And the wrong application of patterns can often lead to code that is even more unmaintainable than without them. Hence, it is necessary to check if a pattern can usefully be applied in the context. Hints for the application of patterns were given above and additional examples will be provided in the following sections, with respect to the models of the *MathTrainer* system.

The Design Class Diagram

The *design class diagram* has its roots in the *domain model* of the OOA phase, but transfers the concepts found in the real world into the context of the software system yet to be built. The *MathTrainer*'s class diagram is shown in figure 3.11. The transition's first step is executed by evaluating classes, associations, and attributes still being necessary for the design phase. In the case of the *MathTrainer*, most of the constructs

make their way from the analysis to the design phase. Exceptions are the subclasses of the *MathematicalOperation*: They cease to be important as the Enum-pattern is applied to model the *MathematicalOperation* concept. Each of the different subclasses is mapped to a public instance of the *MathematicalOperation* superclass, and both the subclasses as the generalization relationship are removed. The second step adds concepts not important in the analysis phase. For instance, a factory is added to the *MathTrainer* system to generate random integers; an additional *StopWatch* aids in measuring the time needed to solve the exam. After the transition has completed, the class diagram is further detailed by applying the patterns of the design phase.

The application of patterns can be seen in many places throughout this diagram. Generally, the *information expert* pattern has been applied for all classes shown on the diagram and is therefore responsible for the way how operations have been partitioned to the major classes. As this pattern interacts with the *low coupling* and the *high cohesion* patterns, these three patterns have interacted providing the solution shown. The factory pattern is used to generate numbers in the *RandomNumberGenerator*. The singleton pattern is also used in this context. The Enum pattern is used to define the different *mathematical operations*. This pattern is very often utilized to get away from using constants as they are generally not very reusable.

Interaction Diagrams

The dynamical view of a system is represented by interaction diagrams. Interaction diagrams can be both *collaboration diagrams* and *sequence diagrams*.

The syntax of a sequence diagram is easily understood: Instances are laid out at the top end of the diagram, and messages pass from one instance to the other. The currently active instance has a thread⁹ associated with it. The description of the messages is not generally defined by the UML, often method names are used to underline the existence of a mapping between the message and the operation. The message description can also contain the operation signature, which means the parameters and an eventual return value. Optionally return messages are drawn which themselves can be labelled with *return types* and *return values* [UML03].

The UML standard has defined ways to notate conditional logic on sequence diagrams. The conditional messages are visualized by enclosing the condition in square braces and putting the condition before the signature of the operation. Loops are defined as rectangular boundaries around the messages affected, where the loop condition is put somewhere into this rectangular space. However, not many tools support this kind of

⁹The active thread is usually denoted by a rectangle.

visualization, which makes it hard to describe the method bodies thoroughly enough to generate code from them.

An example for a sequence diagram is figure 3.12. The same diagram in collaboration notation is shown in figure 3.13.

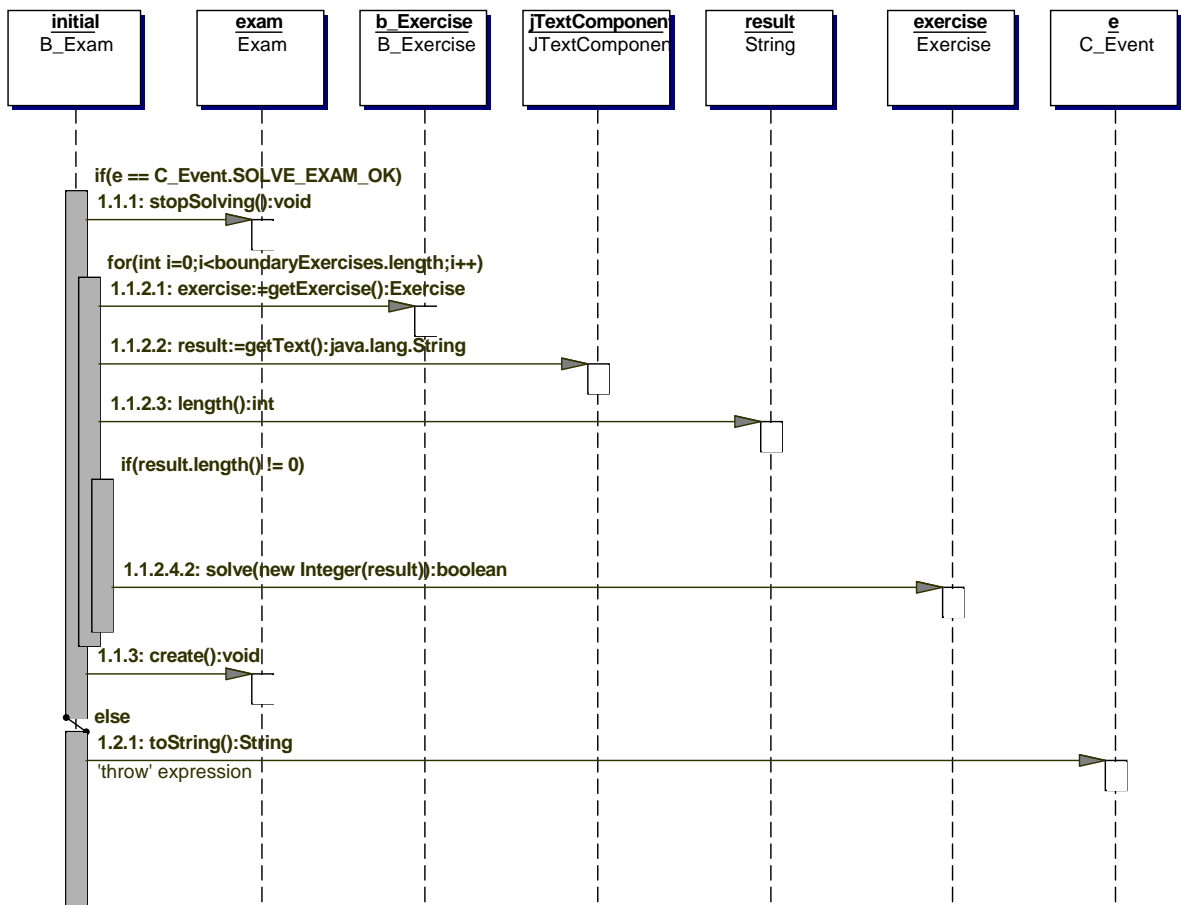


Figure 3.12: The sequence diagram for solving an exam in the MathTrainer example.

Collaboration diagrams essentially show the same information as sequence diagrams, but in a different representation. What is missing in collaboration diagrams is the *timeline*, the information represented by this element is conveyed by the sequential numbering of messages on collaboration diagrams [UML03]. This type of diagram resembles a class diagram when looking at it first, but the classes shown in this model are actually instances of the classes, and the arrows in between are not associations but the messages passing from one instance to another [UML03]. Conditional logic is here more

often supported by tools, also loops can generally be shown on collaboration diagrams.

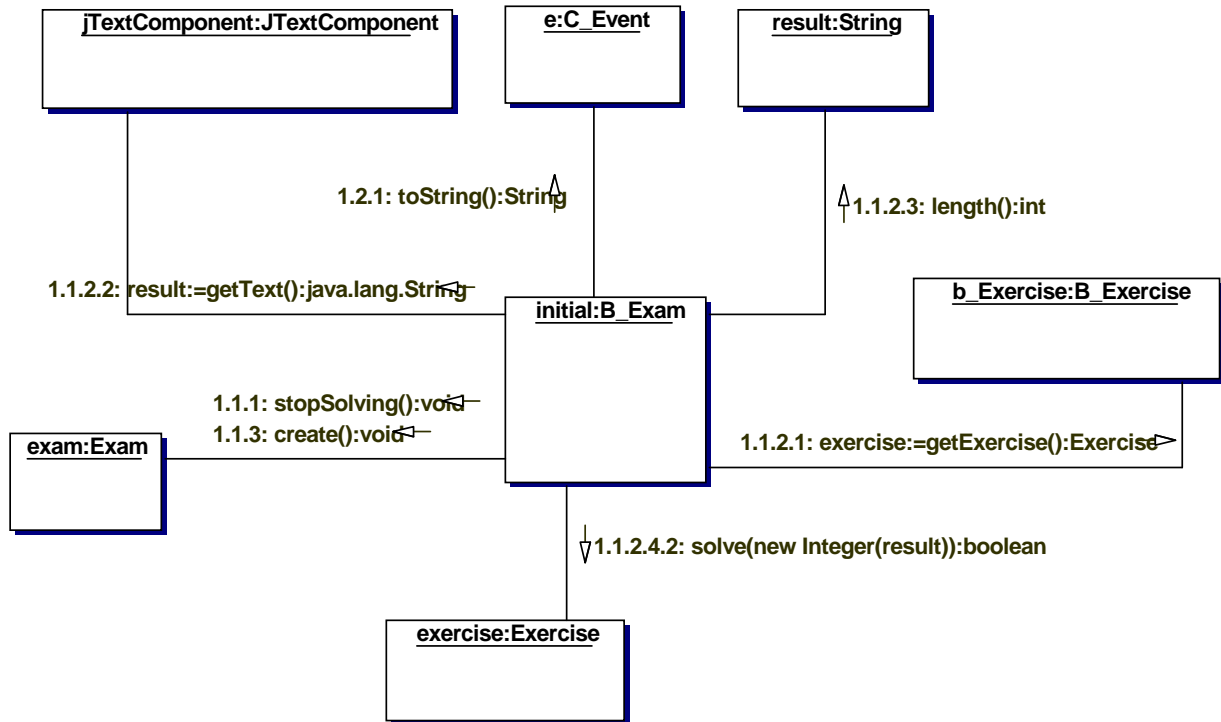


Figure 3.13: The collaboration diagram for solving an exam in the MathTrainer example.

Tooling - Conditional Logic in Sequence Diagrams

The UML defines a way to cope with decisions and loops in sequence diagrams, Rational Rose 2002 does not. To work around this fact and still model the conditional logic arising from requirements [Qua03, p. 81] suggests to use notes and scripts on a single diagram when the logic is simple. When the logic comprises more than some messages she mandates the usage of two diagrams, one for the “*if*” and one for the “*else*” case. To allow the user to easily navigate to other diagrams, Rational Rose offers the ability of *linking* – done by specially marked entries in plain *notes*.

This is a rather unsophisticated way of coping with this problem for the leading UML tool on the market, even if the notion of simplicity is called upon for mandating the exclusion of conditional logic from sequence diagrams [Qua03].

Tooling - Layouting Class Diagrams

When class diagrams become larger, it gets more difficult to analyze and comprehend them. However, the usage of the EBC -pattern, explained in section 3.4.2 leads to an explosion of the number of classes necessary to model the system. This explosion induces class diagrams which can by hardly be understood. A possible workaround is to continue the separation of concerns on the layer of packages and introduce packages for each of the three class types: entity, boundary and controller. Problematic is the high interconnection among the three classes making up one instance of the EBC-pattern. Hence, the structure can not easily be conceived if the classes are spread on several diagrams. So it is necessary to include at least one class diagram in the documentation which shows the structure of the EBC-pattern used in the system. Even better is the inclusion of a sequence or collaboration diagram showing the dynamic interaction.

Apart from those additional diagrams, the main class diagram of the system should not show boundary or controller classes, except if they are of a great importance for understanding the system.

Tooling - Using the Object Constraint Language in Rational Rose

OCL, the OMG's object constraint language, helps formulating restrictions on what can happen (or what can exist). These constraints are then visualized on the diagrams. The language exists as an alternative to natural language constraints and is – maybe due to its alternative state – neither widely known by programmers nor implemented by tool vendors.

Rational Rose 2002 is not an exception. The only way to notate OCL expressions on diagrams is by attaching notes to diagram elements, so there is no syntax check and no immediate utility of writing constraints in OCL. Generally, it is not recommended to use OCL as a constraint language as long as this sad state of affairs is not changed.

3.5 The Object-Oriented Analysis and Design Phase using xUML

The following explanation of the OOAD phase in xUML is based on the UML approach shown in the previous section. Major differences are outlined and the steps are compared to those taken with the UML.

3.5.1 The Object-Oriented Analysis

Using xUML, the phase of OOA comprises the same steps as in using the “ordinary” UML. An interesting difference is about nomenclature: the *domain model* used to show the conceptual classes found in the real world in UML is the normal *class model* in xUML.

The *domain model* in xUML, on the other hand, is something like a model of the main packages (called domains in xUML) of a system, including their interaction. This is, as mentioned in 2.3.2 comparable to a *class diagram* in UML which contains only packages (which themselves contain the classes of the system).

The Domain Model in xUML

The domain model of the MathTrainer example is shown in figure 3.14; 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. A domain can also be seen as lying outside of the scope of the system, then it is not handled in the development process – except for the interfaces being necessary to connect to the domains within the system.

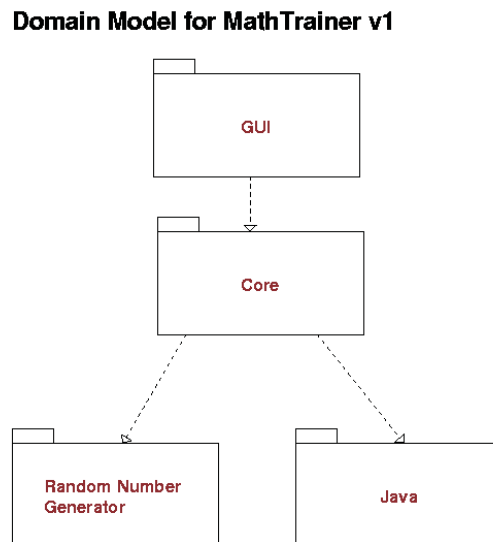


Figure 3.14: The xUML *domain model* of the MathTrainer example.

Use Case Diagrams in xUML

The second step of OOA is in xUML as in UML the generation of use cases and use case diagrams. The generation and visualization of these diagrams resembles very much those generated by the application of UML and are therefore not replicated here.

A problem in the OOAD process using xUML is the even less distinct separation between OOA and OOD. Using xUML the creation of only one diagram is recommended [MB02, Sta02] – this diagram should be closer to the *analysis* of the real world than the *design class diagram* in UML, but still reflects some design decisions. As an example, the addition of *operations* to the classes in this diagram is mentioned.

Class Diagrams in xUML

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 MathTrainer example, is shown in figure 3.15. As mentioned in section 2.3.2, the xUML class model resembles to the class models built in UML. The differences between the models shall be outlined in the following paragraphs.

Notation of Associations: One of the major differences in the visualization of associations between xUML and UML are the *multiplicity expressions*, which are explained in 3.5.1. Additionally, UML offers displaying the notion of *navigability*, which is not shown in xUML¹⁰. Furthermore, the xUML subset defines a unique name for associations (an R followed by a unique identifier, e.g. R1, R2,... as explained in section 2.3.2). Finally, the role specifiers of UML take over the role of describing the associations. Or, to put it more correctly: The description of the association is put to the place where the role names reside in UML. To read an xUML association, first the class name is read, than the far end description and then the other class name (e.g. *Exam* has exercises of type *Exercise Type*). In UML, a class is described by the role name on the near-end of the association [Qua03, p. 94]. This is clearly different and can lead to misinterpretations of xUML diagrams which should be avoided.

¹⁰This is the reason for the fact that no arrows - except the generalization arrows - are displayed on xUML class diagrams.

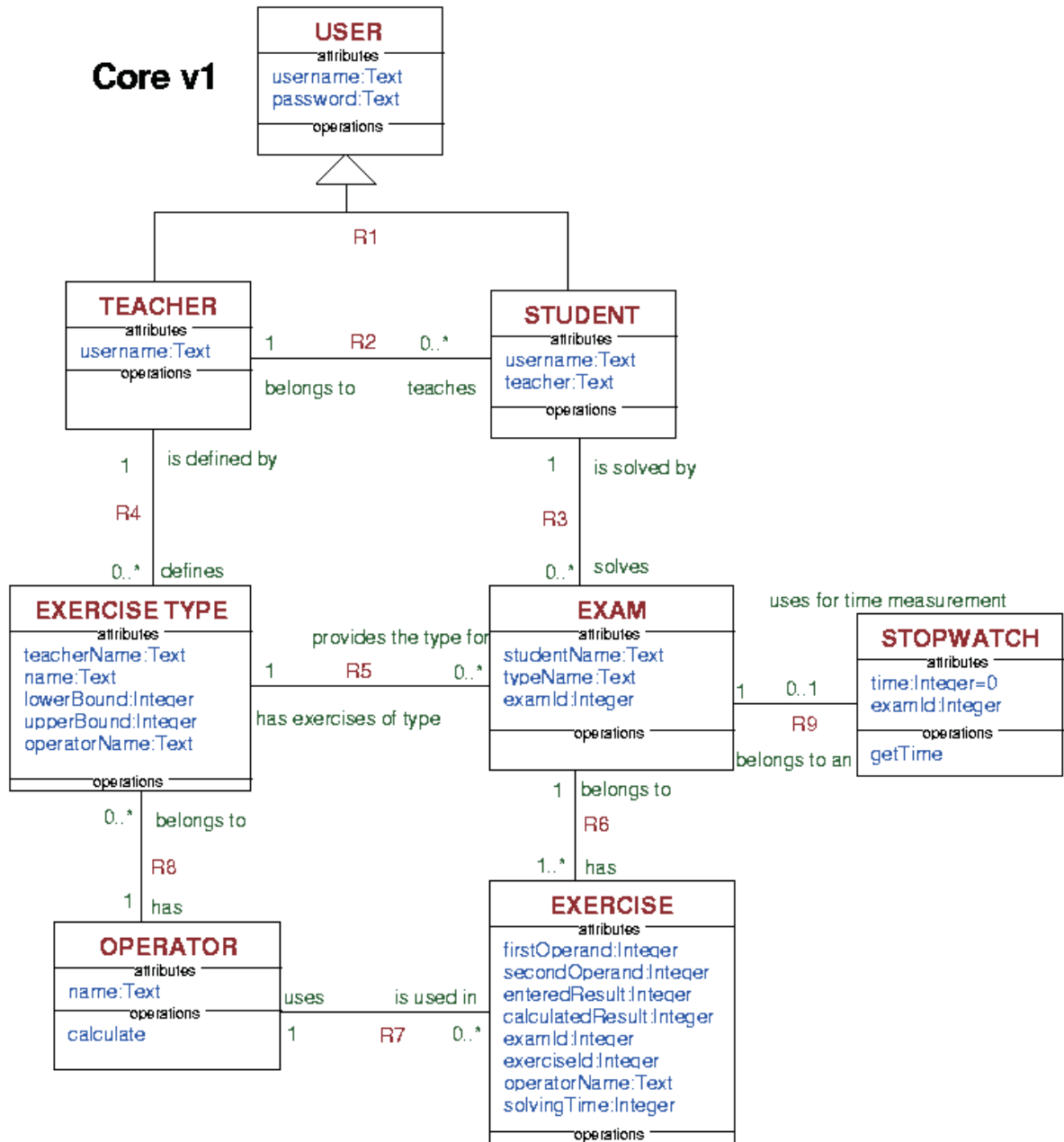


Figure 3.15: The xUML class diagram of MathTrainer's Core domain.

Multiplicity Expressions in xUML: According to [Sta02, p. 303] xUML allows to specify the allowed multiplicity in a class model in the form of

- 0..1,
- 1,
- 0..* and
- 1..*.

From the requirements analysis of the MathTrainer example follows that *one* exam consists of *ten* exercises. Modelling this fact would be very complicated when using the xUML profile – in UML a multiplicity like the one given above can be defined. Starr suggests in [Sta02] as a remedy to reconsider the *probability of change* of this multiplicity. This can be achieved by first posing the question if the relationship being modelled

- might change anytime or
- is something that is a fundamental law of a science like mathematics, geometry or physics.

In the *second case*, the behaviour should be modelled explicitly. This is easy with small multiplicities (as an example, a flat surface has a backside and a frontside), but can be impossible with larger multiplicities – which are seldom to find in fundamental laws anyway¹¹. An example of this would be the definition of a simple arithmetic problem in the mathematics: there are always exactly two numbers which are joined by the operator and on which the operator is executed. A class diagram showing this relationship can be found in figure 3.16. For the *first case*, the constraint should not be included in the model: if a company has 150 customers and the 151st comes along, nothing should change in the behaviour of the system – the code should work with 151 customers as good as with 150. The place to put this information is the *colouring*¹² of the xUML-model: the developer instructs the compiler to consider this relationship to have a multiplicity of ten, and the compiler ought to find a viable solution with a small footprint and a good execution time [Sta02].

¹¹An exception would e.g. be the amount of protons in an atom of a given element. It is hard to distinguish different protons though, so they might be modelled as a count-attribute of the atom class.

¹²*Colouring* is the process of putting a kind of acetate sheet over the model - without changing it - to instruct the model compiler to treat parts of the model differently (e.g. a part of the model is to be executed on a special processor) [Sta02, p. 19]

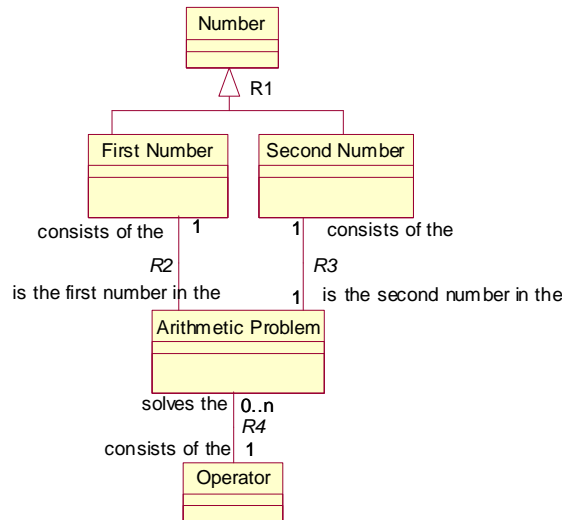


Figure 3.16: How an arithmetical exercise, its operator and the operands interact.

Referential Attributes in xUML: An interesting addition of the xUML profile to the UML standard is the introduction of *identifiers* [Sta02, p. 57]. These identifiers are – as in the *relational data model* – the key to one special instance of the class. Identifiers can be *implicit* or *explicit*. An implicit identifier is an identifier a class gets if there is no other identifier defined. This identifier is provided automatically, by the underlying software layers, to be able to distinguish among instances – it might be done by using keys, pointers, indices, handles, memory co-location or any other software architecture related mechanism. When – later in the process – statements in the action specification language are written, the implicit identifiers are used when a link is set or removed, or if a link is followed from one class to another and a single instance or a set of instances is received – no explicit identifier is necessary. Still, if bridging to another domain in which a GUI resides is necessary, and the instance data shell be presented to the user, this user might expect the identification to be of a special format, e.g. an item identification number, a car plate number or a social security number. In such cases, it is better to model an identifier explicitly¹³ [Sta02]. As in the relational data model, the identifier can consist of one attribute or more than one. If more than one is used, the identifier is termed *compound*. When designing xUML classes, their attributes and identifiers, the relational principle of the first normal form, second normal form and third normal form should be obeyed – making the close coupling between xUML and

¹³In iUML by Kennedy Carter every class has to have its explicit identifier - hence the restriction is even higher.

relational principle even more obvious. An introductory explanation of these concepts can be found in [⇒Normalization], and in [Sta02, p. 61ff] they are applied to xUML; although not explicitly mentioned.

3.5.2 The Object-Oriented Design

The OOD phase using xUML begins by adding a collaboration diagram to each domain – as the major parts of the class diagram have already been designed in the analysis phase. It should be noted that the class diagram might not be finished yet, it is possible to return and refine it during any of the next steps. Additionally to the attributes and associations, it might be necessary to add operations to the class diagram. A major difference to the design phase using UML is that the concepts *information hiding* should not lead to wrapper methods, even if the target language mandates them. These wrapper methods will be created automatically when the model is compiled. For creating other necessary operations the usage of the GRASP pattern – explained in section 3.4.2 – is advocated. The EBC-pattern – explained in section 3.4.2 – does not aid in designing xUML models as the controller is here often the *state machine* of the class. Additionally, the boundary class is seldom included in the model as the GUI is often in a completely separated domain, connected by a *bridge*.

The xUML Collaboration Diagram

This collaboration diagram shows essentially the same classes as the *class diagram*, but carries additional information about their interaction and is therefore a dynamical diagram [MB02]. For the MathTrainer’s core domain, this diagram is shown in figure 3.17.

Three types of classes are displayed on an xUML collaboration diagram, the type is expressed by the stereotype provided in the first compartment of the class symbol.

Class: The stereotype class describes ordinary classes which are defined on the class diagram and do not have any special meaning.

State machine: The term *state machine* denoted classes which have a state machine, expressed in a state chart diagram, associated with them. The step of determining which classes should become state machines and how to associate them with state charts will be explained later on.

Terminator: Finally, the stereotype *terminator* marks classes which are outside of this domain and interact with it (they are interface classes). Terminators can

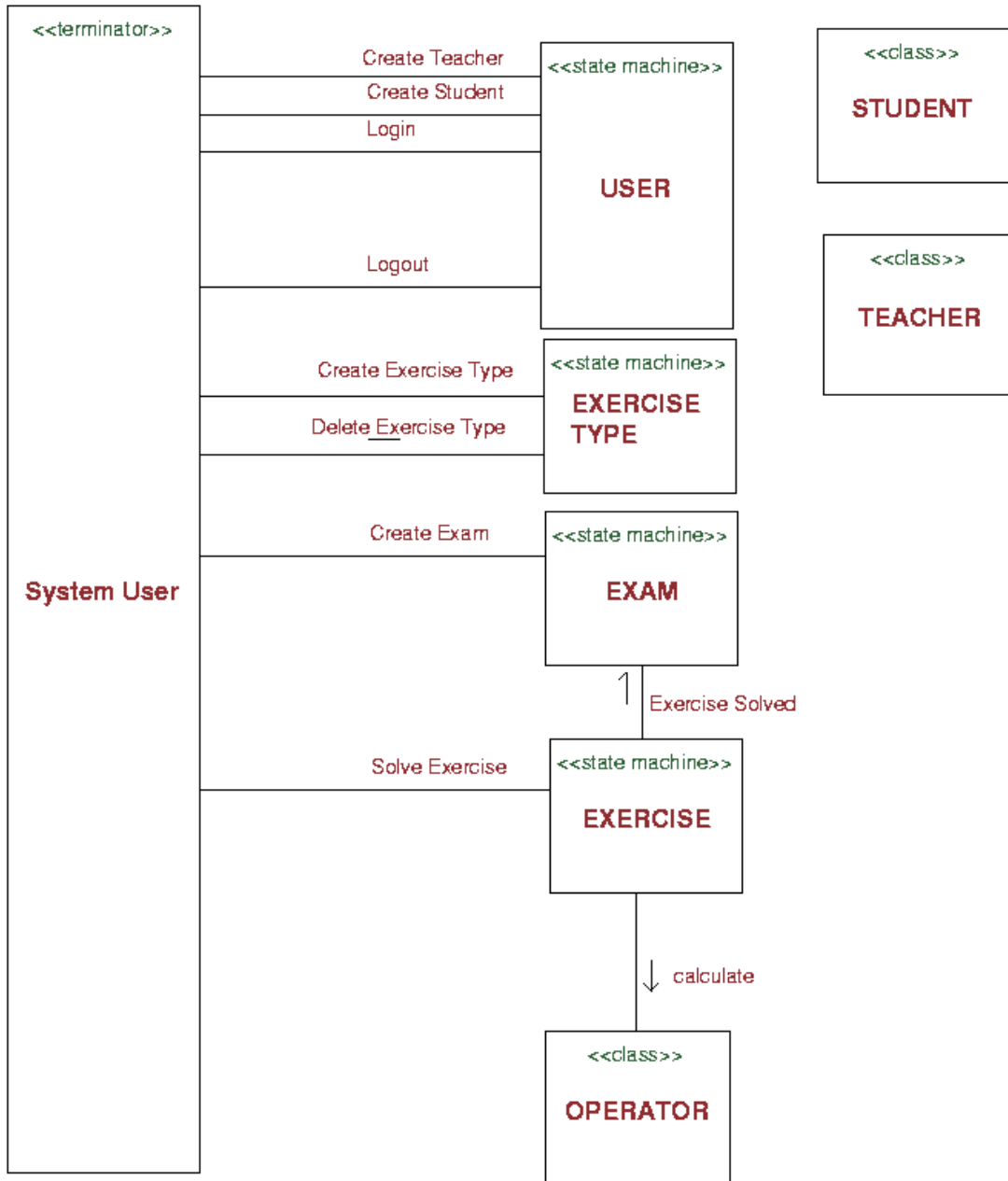


Figure 3.17: The xUML collaboration diagram for the MathTrainer's *Core* domain

have signals and method calls associated with them which are activated by the classes of the domain, terminators themselves can send signals to the domain and call operations of the domain.

State machines and State Chart Diagrams

Starting off from the class diagram of the domain, the next step is to think about which of the classes shown on this diagram are *state machines* [MB02]. Generally, almost any object can be considered as being a *state machine* as almost all objects start off being in a special state and going through a lifecycle in which they traverse a sequence of states. By determining which state machines are important to adequately model the behaviour of the domain, a correctly working system can be achieved.

For the MathTrainer example, most classes have been identified as state machines. One exception are the *Student* and the *Teacher* class. As they are subclasses of the *User* class, a separate state machine for them is not recommended [MB02]. Another is the *Operator* class for which the state information is not necessary as objects of this class are created and do not react to any transition. An example for a state chart diagram is represented in figure 3.18, exemplifying the state chart for the *User* class.

This figure 3.18 consists of the essential components of such a state chart diagram: Each chart is composed by representing the different *states* the object can assume, and the *signals* which cause the transition to another state. These signals are labelled with a unique identifier followed by a colon, then a name which is given by the developer and finally by the parameters accompanying them. The action which is taken upon entry into the state is shown in the state's box and is denoted in an action specification language (in [UML03] the semantics, but not the syntax of such a language is standardized). This language depends on the manufacturer of the tool - the language used by Kennedy Carter in their tool iUML shall be presented (and evaluated with respect to [UML03]) in the next section.

The Action Specification Language by Kennedy Carter

In [WKC⁺03], the ASL is explained thoroughly. As the language is situated in the public domain, it has been implemented several times independently [UML03].

The ASL provides the user with syntax to denote *sequential logic*, to access *data items*, to manipulate *classes and objects*, to handle *associations and generalizations*, to generate *signals*, to execute *arithmetic and logical operators* on data, to call *operations*

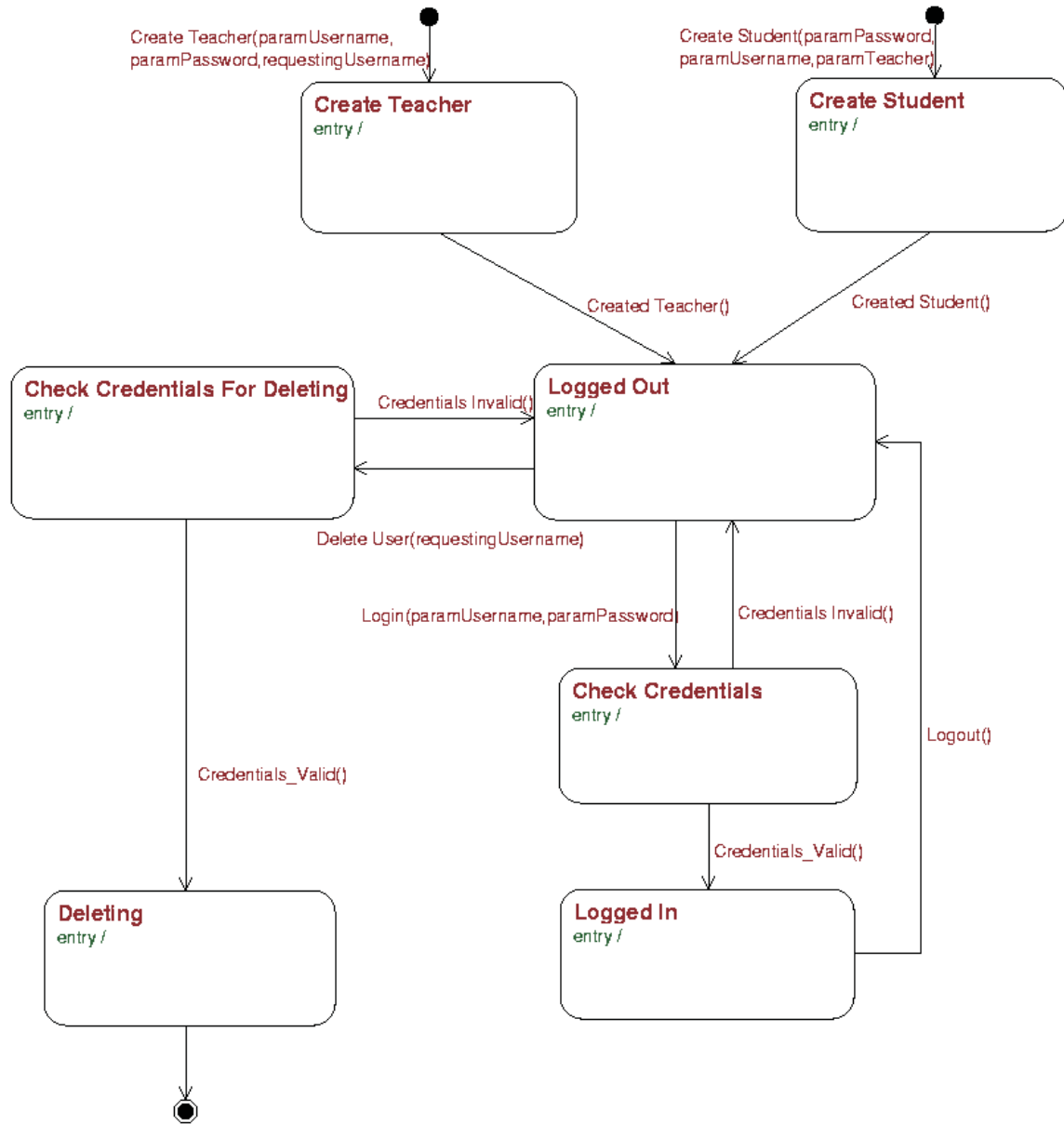


Figure 3.18: xUML state chart of the User class.

and to handle *sets* of objects. Additionally, a *timer* is provided to implement time based functionality [WKC⁺03].

Some of these possibilities shall be explored with an explanation and a short example.

Sequential Logic Sequential logic can be represented in the ASL by using any of the following statements.

- `switch`
- `if`
- `for .. loop`
- `loop`

The statements can also be nested, if this is necessary [WKC⁺03]. The example 3.3 shows an `if` statement in ASL.

```
if countSolvedExercises = 10 then
  generate EXM3:Exam_Solved() to this
else
  generate EXM5:Exam_Not_Yet_Solved() to this
endif
```

Example 3.3: An `if` statement in ASL.

Handling associations and generalizations: An interesting feature of the ASL is to handle associations and - in the same way - generalizations. It is possible to navigate to the other end of associations, to establish a link between to classes and to delete this link again [WKC⁺03]. Example 3.4 shows these possibilities.

```
#type and mathOp are both local variables
#which have been defined before

link type R8 mathOp

newMathOp = type->R8

unlink type R8 mathOp
```

Example 3.4: Establishing a link, navigating along it and deleting it, written in ASL.

Sending signals: Out of ASL code, it is possible to send signals to other classes and terminators [WKC⁺03]. The signal sending syntax is relatively easy, as can be seen in example 3.5.

```
#USR ....short name of the state machine
#1,2 ....unique index of the signal
#Create_Teacher, Create_Student .... name of the signal
#in the brackets the parameter are provided

generate USR1:Create_Teacher("teacher","teacher","admin")

generate USR2:Create_Student("student","student","teacher")
```

Example 3.5: Syntax for sending signals in ASL.

Handling Sets: *Sets* are generated in ASL by following a link to an association with a multiplicity of higher than one or by executing a *find* statement. On collections, many operations are defined. For example, it is possible to iterate through the set or count the instances comprising the set [WKC⁺03]. Example 3.6 shows the details.

```
local_exam = find-only EXAM where studentName="student" & \
                Current_State = 'Unsolved'

#Set obtained by navigating along an association
#Sets are denoted by {} in ASL
{exercises} = local_exam->R6

#Iterate through the set and send each instance a signal
for exercise in {exercises} do
    generate EXC1:Solve_Exercise(20) to exercise
endfor
```

Example 3.6: Handling of sets in a test method written in ASL.

Calling Operations: The example 3.7 shows how a method with one parameter and two return values is called.

```
#has two return values (both the teacher and its user object)
#the difference to signals are the notation with square brackets
[teacher, user] = USR5:GetTeacherAndUser["username"]
```

Example 3.7: Calling an operation in ASL.

Handling Objects and Classes: The final example 3.8 shows how objects and classes are handled. This is done by showing a test case where a *Student* and a *User* are created, they are connected to create a generalization relationship and then the *Student* is deleted and the *User* instance is connected to a teacher – thus showing that an object can change its generalization relationships as easy as its associations.

```

#Create a user
tmpUser = Create USER with \
  username = paramUsername & \
  password = paramPassword

#Create a student
tmpStudent = Create STUDENT with \
  username = tmpUser.username & \
  teacher = paramTeacher

#look up the user that belongs to the teacher
tmpTeacherUser = find-only USER where username = paramTeacher

#navigate to the teacher itself
tmpTeacher = tmpTeacherUser->R1.TEACHER

#link the student and the teacher
link tmpStudent R2 tmpTeacher

#link the user and the student objects to
#create a generalization relationship
link tmpUser R1 tmpStudent

#student creation is finished – now a change: the
#created user object is to be a teacher
#so we reset the links and delete the student
unlink tmpUser R1 tmpStudent
unlink tmpStudent R2 tmpTeacher
delete tmpStudent

#Create a teacher
tmpNewTeacher = Create TEACHER with \
  username = tmpUser.username

#link teacher and its user object
link tmpNewTeacher R1 tmpUser

generate USR4:Created_Teacher() to tmpUser

```

Example 3.8: Creating and deleting instances in ASL and changing the position in a generalization relationship.

Model Simulation

The implementation phase of the xUML OOAD process would be the transformation of the designed model to an executable one on the desired platform. A very interesting and useful aid when designing models is that it is possible to use *model simulators*. These

are tools automatically executing the model, but on the special platform of a simulator so that the model can be debugged before it is finally executed [MB02]. In a strict sense, this is a form of implementing an xUML model, still it is used in both the analysis and mostly the design step to *do the right things right*.

This step aids in designing models as the viability of the model can be ensured by simulating the instantiated model, providing a set of input stimuli and checking the postconditions. If the model was designed to build a calculator, the input conditions “3” and “5” should lead to the postcondition “8”, assuming the model was initiated to add the numbers. Example 3.6 shows a test method that is executed upon simulation of the model and triggers some behaviour of the objects. In this case, the exam is solved by sending signals to the exam’s exercises to communicate when – and with which value – they are solved.

3.6 Summary

The phase of Object-Oriented Analysis and Design (OOAD) was exemplified on the *MathTrainer* system using both UML and xUML. Analyzing in UML leads to the *use case diagram*, which was further explained by written use case stories and an *activity diagram* for each use case. The real world was – in its relevant properties – captured in the classes, attributes and associations of the *domain model*. By both enhancing and reducing the concepts of the domain model the transition of OOA to OOD leads to the class model of the design phase. In an iterative process, both interaction diagrams and the class model evolve and influence each other. In designing interaction diagrams, special effort has to be made to capture all interesting aspects of the system – these diagrams are the basis for the later code generation of the dynamical aspect in this approach. State chart diagrams could additionally be employed to further clarify the life cycles of objects, but were not designed as they were used in the xUML approach.

Using xUML, the main constructs of the UML analysis phase can be reused. Both the use case diagram and the activity diagram provide the information necessary to model an xUML diagram. The xUML domain model is different from the UML domain model as the first depicts the top level interconnection of main packages and the second shows all conceptual classes, attributes and their interactions. The xUML class model has its roots in the analysis phase of the OOAD as this model is more a picture of the real world than visualizing the software system yet to be built. Essentially, it uses the main constructs as its UML equivalent, some differences can be found in the notation for associations, in multiplicity expressions and in the referential attributes used in xUML. An xUML collaboration diagram was designed to visualize the dynamical interconnection of classes. The final artefact of designing in xUML were state chart diagrams which were used to

illustrate the lifecycle of objects where necessary. The states of these diagrams are further refined with action specifications, these being executed when the object enters the state. An example for a notation of such actions was provided by introducing the Action Specification Language (ASL). The state charts created in this phase will be the basis for the code generation of the dynamical aspect in the xUML approach.

Both the UML and the xUML approach can be used to model a software system in great detail. Whereas the emphasis in terms of time and commitment has to be on the interaction diagrams in the UML approach, it must be on the state chart diagrams in the xUML approach. The better and the more detailed the models of the design phase, the more code can be generated in the implementation phase. A major difference between the UML and the xUML approach is that with xUML, the design phase marks the final point of the classical software system development. Afterwards, the new tasks of colouring the model and configuring the model compiler take over. In UML, the distinction between design and implementation phase is not so clear and several iterations will take place, slowly shifting the emphasis from design to implementation. In both approaches, it is always possible to return to the design phase and clarify the information the model contains, when code generation is employed, these clarifications can subsequently be conveyed to the implementation phase.

Table 3.4 shows what times were necessary to actually analyze and design the system. Learning phases have been subtracted from the amount of time given in this table. In effect, the analysis phase took the same amount of time for both approaches, the design phase took longer with the xUML approach due to the detailed state chart diagrams necessary.

Phase	UML	xUML
Analysis	7 days	7 days
Transition from Analysis to Design	1 day	1/2 day
Design	10 days	15 days (due to the detailed state chart diagrams necessary)

Table 3.4: An overview of the time necessary for the Object-Oriented Analysis and Design phase.

Chapter 4

Implementation Phase

The description of this phase shows how the output of the previous steps has to be augmented to reach an executable software system.

4.1 Implementation on the Basis of the UML model

The first step in converting the model into an *executable* application was done by choosing Java as a programming language. Successively, the GUI prototype – which was originally written in Java – was connected to the generated classes of the business layer and finally persistence code was added to these classes. This process will be explained in the following sections.

4.1.1 Code Generation

Most available tools generate code from UML class diagrams – *Rational Rose 2002* also provides these capabilities. However, the code generated is not more than a static structure of the classes, and associations are not handled in a very detailed way [BKL⁺01]. Generating code from interaction diagrams is not common for tools dealing with UML, even though some do it, as will be evaluated later on with the case of *ControlCenter*.

Manual Mapping

Hence, if automatic generation of code is not fully possible, [HK99] suggests to handle the process of converting UML models into code in a rule-based way instead of handling

each construct separately.

Such rules must be defined for the mapping of

1. *classes*, of
2. *attributes*, of
3. *operations* and of course,
4. *associations*.

As an example, a rule-based approach for associations is very easy to accomplish with 1:n multiplicities with a navigability in just one direction, it gets harder if navigability in both directions is desired. In the *MathTrainer* example, the work was done without using bidirectional navigability, if this construct is necessary, [HK99, p. 271] provides a mapping.

An example of such a mapping to Java source code is shown in figure 4.1. Here the easy case of an association with navigability in only one direction is treated. This approach is taken by *Rational Rose* as well when generating code from the class diagram. Hence, this approach is also reflected in the *MathTrainer* source code.

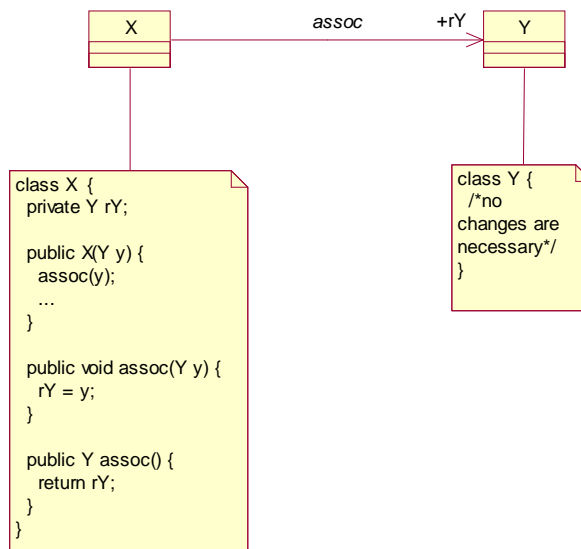


Figure 4.1: How an unidirectional 1:1 association can be mapped to Java source code [HK99, p. 270]

Automatic Mapping

Further than *Rational Rose*, the *Together ControlCenter* goes in its code generation capability. ControlCenter does not only offer code generation from class diagrams, but also from sequence diagrams – this gets very close to the desired full generation of code from UML models.

Example 4.1 is code that is totally generated from the sequence diagram of 4.2, except for the class statement itself which was generated from a class diagram. This automatic code generation works very well. There is just one problem with the mapping of diagrams to code (and back again): the constructs of the language which do *not comprise a message call* will not be reflected on the sequence diagram.

In Java, this is the case with the instantiation of *primitive types* like `int` or `float` and the assignment of a result of *mathematical* and *logical* operations to them. Additionally, the assignment of one object reference to another can not be represented on the sequence diagram and therefore not be generated. Furthermore, constructs such as the `throw <exception>;` statement can be reverse engineered, but not forward engineered using ControlCenter (`try` and `catch` statements can be represented). Finally, any comments in the code are lost after a round-trip engineering has finished.

The problems are illustrated by example 4.2 and figure 4.3. They show the loss of information that occurs when doing reverse engineering – additionally, this information loss prevents this kind of code from being forward engineered. Due to these flaws, it is not 100% viable to generate source code from sequence diagrams with *Together ControlCenter* – though it must be stressed that the UML standard provides no way to accommodate these constructs in a sequence diagram. The essential problem is a sequence diagram being object-oriented and a language like Java having constructs not being object-oriented.

When looking on the xUML approach, it is possible to learn how these problematic code lines could be accommodated in the sequence diagram: there, the action specification is added to the symbol for the state in the state chart diagram. Following this logic, the code lines which are not directly representable could determine the size of the *activation bar* in the sequence diagram. While the activation bar is not large enough to display these code segments to the user, it could show a text editor to the user in which he could enter these code lines, including comment lines. Finally, the length of the activation bar would then be stretched or reduced, to graphically show how much logic has been accommodated in this segment. Additional comment lines (which can not be accommodated in the diagram) would be added as notes or in the description of the sequence diagram's other elements. To add graphical interaction with the user, the code segments could be presented as a *tool tip* to the user. The advantage of this

```

public class HelloWorld {
    public void helloWorld(Integer numberOfDisplays){
        int count = numberOfDisplays.intValue();
        for (int i = 0; i < count; i++) {
            if (i == count - 1) {
                // message #1.1.1.1 to this:HelloWorld
                this.displayHelloWorld(true);
            }
            else {
                // message #1.1.2.1 to this:HelloWorld
                this.displayHelloWorld(false);
            }
        }
    }

    public void displayHelloWorld(boolean isLast){
        if (isLast) {
            // message #1.1.1.1.1 to <unnamed>:javax.swing.JOptionPane
            JOptionPane.showMessageDialog(null, "Last_Hello_World");
        }
        else {
            // message #1.1.1.1.2.1 to <unnamed>:javax.swing.JOptionPane
            JOptionPane.showMessageDialog(null, "Hello_World");
        }
    }
}

```

Example 4.1: The code generated by the sequence diagram of figure 4.2.

approach would be the possibility of full code generation and no violation of the UML standard whatsoever, paired with the ability to repeat round-trip engineering as often as necessary.

```

public void problematicOp() {
    //This operation does not make any sense
    /*But the constructs used here
    do prevent full round-trip engineering
    to be possible in ControlCenter*/
    int i=5;
    i=i+10;

    if(i==16)
        throw new IllegalStateException("Eventually_arriving_there.");

    String str = "abc";
    String str2 = "def";

    String str3 = str+str2;

    System.out.println(str3);
}

```

Example 4.2: The code being the base for the sequence diagram of figure 4.3.

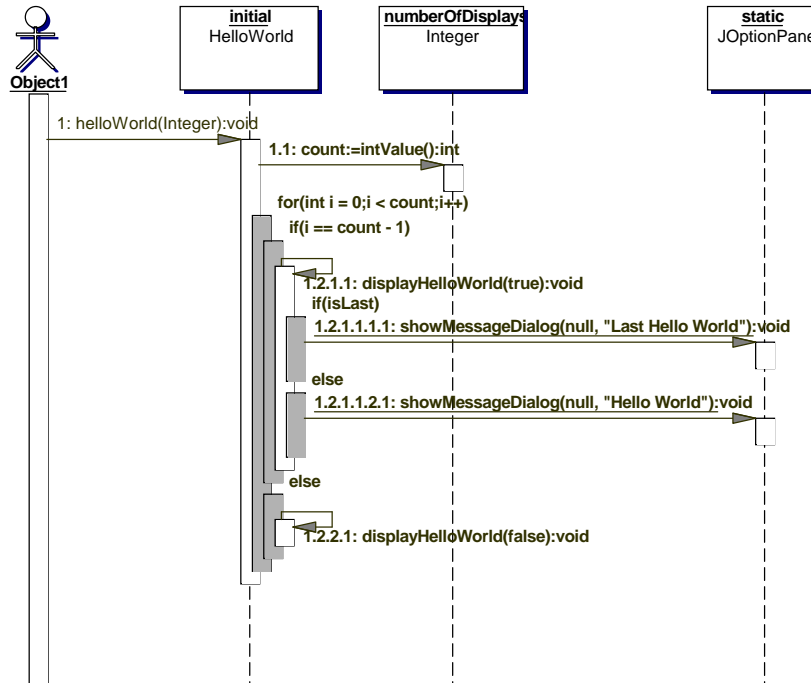


Figure 4.2: A sequence diagram automatically being mapped to method bodies.

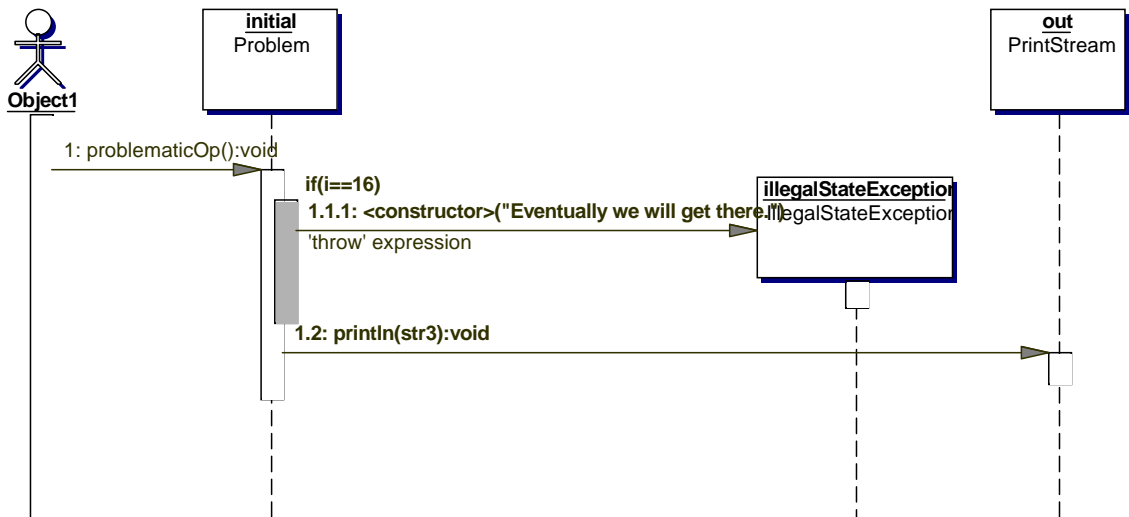


Figure 4.3: A sequence diagram that was automatically created by reverse-engineering from example 4.2.

Tooling: Code Generation with Rational Rose

Code generation with *Rational Rose 2002* is relatively easily accomplished. It is necessary to select the classes being forward-engineered and to set the relevant classpath settings. Code is generated only from the class diagrams: the sequence diagrams, state diagrams, use cases, deployment diagrams and so forth are totally left out. So what a developer gets when executing the code generation feature is the static structure of the system. Another fact is that constraint statements written in OCL are not considered when the code is generated - even though they could be used in specifying pre- and postconditions for testing code or could supply restrictions on associations. It is therefore recommended (see section 3.4.2) to use natural language for constraint statements, OCL is not widely understood by developers.

If this state of affairs is unsatisfactory for the developer, tools are available which generate code from the *state chart diagrams* and the *actions* being defined by them¹, or changing the modelling environment, a possibility would be to use *Together Control-Center*. In [Lar02, p. 574], Larman argues that the availability of code generation from *sequence diagrams* is one of the key functionalities users can expect from their UML CASE tool.

4.1.2 The Graphic User Interface Layer

Building the user interface lies almost completely out of the scope of UML diagrams. They can hardly show more than a glimpse on the static structure of the classes of the GUI and some overview of the behavioural aspects. What makes it so hard for the UML to be of use when designing the GUI is the fact that the framework such a GUI consists of is often a very complex conglomerate of classes. Additionally, the user interface itself has as its goal a graphic output to the user - investigating this output is the easiest way to understand how the user interface works. Diagramming does rather hide the interesting points than showing them to the developer who desires to understand the user interface part of the program. Nevertheless, the GUI is a major part of any program, and the code necessary to construct it can be as much as half of the overall code. Building the GUI is therefore also an expensive part of the development process and it should be coped with rather early in the sequence.

So the best way to design a user interface is to build a prototype in some GUI builder early – producing code which can later be connected to the classes generated from the UML models. A program creating a GUI from the UML models automatically is

¹A plug-in is available from *BridgePoint* software enabling the developer to model xUML in Rational Rose.

Janus (developed by oTRIs) – this program uses C++ and also connects to a Microsoft Access database to ensure persistency [Bal00, p. 15]. Later, by reverse engineering, the interesting aspects about the GUI can be visualized in the form of some diagram; especially the connection of the user interface with the *application layer* is of interest here (section 3.4.2).

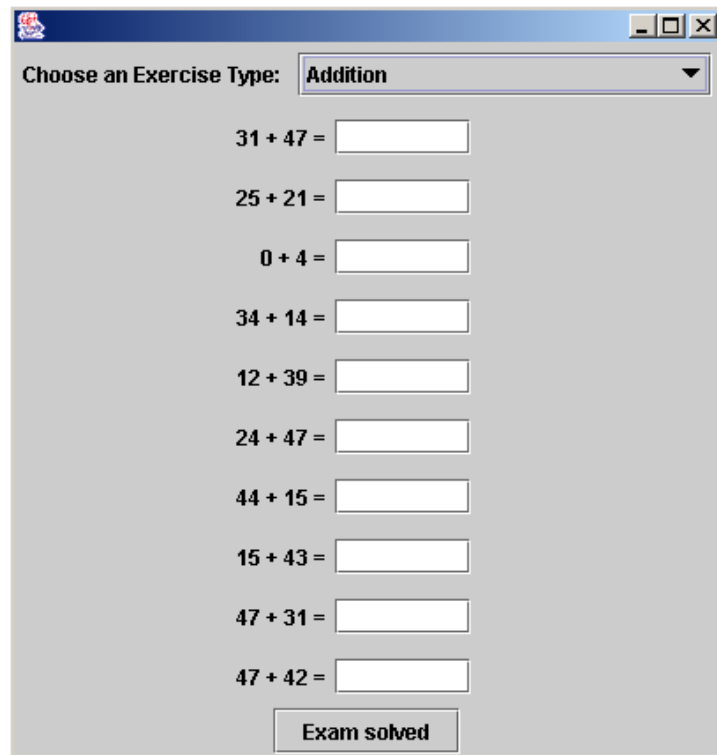


Figure 4.4: The Swing-window for solving an exam.

When implementing a prototype, it has to be decided how detailed this should be. As a recommendation, it is not necessary to be able to interact with the GUI in this first step. As an example, the GUI prototype for solving an exam is provided in figure 4.4.

Tooling - Netbeans and Rational Rose

When the design of a GUI prototype and the reverse engineering of this prototype with *Rational Rose* are discussed, severe incompatibilities with Netbeans, the open source IDE of Sun, have to be mentioned. When the user interface is laid out in the GUI editor in Netbeans, special comment lines are added to the source code. When reverse engineering these classes into Rational Rose and doing a forward engineering

again, these comments are lost as they are situated *outside of method bodies* somewhere between methods – *Rational Rose* can not handle these comments properly. Trying to start the design editor of Netbeans again it is impossible, because these comment lines are missing. The only remedy to this problem is not to use Rational Rose as a modelling tool and Netbeans as a GUI editor together.

4.1.3 The Persistence Layer

For the persistence layer, a relational database was chosen, namely MySQL. The reason for this choice lies in the properties of this product; it is both open source and has a wide community of developers. Other open source products might provide more functionality than MySQL, but generally do not offer the speed of this database system and the amount of documentation available.

Database Design

Designing a database based on the information provided by the UML model is not always straightforward. As often the object-oriented approach used in the model has to be mapped to a relational database in the back-end there has to be some *object-relational mapping*. Descriptions of this mapping can be found in [Bal00]. Basically, the pattern works in mapping the classes to tables; the attributes of classes are mapped to columns of the tables and the associations are mapped to either foreign key columns in the respective tables or to tables of their own. In the case of the *MathTrainer* example, the mapping has been less strict for the sake of simplicity.

MySQL offers different types of tables depending on the purpose for which the database table is going to be used, as transaction handling should be provided by the persistence layer, the table type *InnoDB* was necessary. The create script for the table *user* is shown in example 4.3.

Database Access

Arguably, the professional application developer should not struggle with designing the database access himself. There are well-prepared, full blown persistence architecture available which can readily be used in project. Possibilities for achieving persistence in Java include all tools based on the JDO standard and the open source Castor persistence layer which is freely available (some trade off regarding the functionality has to be mentioned, at least as of current versions below 1.0). For the example used along this

```
drop table user;

create table user (
  username varchar(100) not null,
  primary key(username),
  password varchar(100),
  parent_username varchar(100),
  index par_ind (parent_username)
) type=InnoDB;

alter table user
  add constraint foreign key (parent_username) references user (username);

insert into user values ('teacher','teacher',null); insert into
user values ('student','student','teacher');
```

Example 4.3: An example SQL statement for the creation of the user table in the MySQL system.

project, a sophisticated persistence layer like proposed above would be helpful, but would also add complexity to the source code, so a simple approach using JDBC was used. According to the *information expert* pattern the functionality to make an object persistent should be in the object itself, but as explained before, this approach could lead to *low cohesion* and therefore violate another pattern of good software development.

The approach taken for this project is somewhere in the middle: There is a class named `C_DatabaseManager` which provides the handling of database connections (see example 4.4) and executing statements (see example 4.5, as well as handling any exceptions happening on the way. The objects themselves are still responsible of handling the methods associated with persistence, like `save()`, and generating the SQL strings being associated with these methods. Additionally, the *observer* pattern is used to let these objects handle the result sets – which are generated by executing the queries – themselves by implementing the `C_DatabaseResultSetHandler`-interface. By using this pattern, the `C_DatabaseManager` class can define generic code being the same for all the entities, and code reuse is increased.

Example 4.6, extracted from the `User` class, shows how persistence is handled in the objects themselves: a SQL statement is generated for calling the method `change()` – which is converted into an `UPDATE` in SQL.

```

/**
 * Creates a new instance of DatabaseManager
 * and connects to the database.
 * <p>
 * If connecting is not possible, an exception
 * is thrown.
 *
 * @throws C_MathTrainerException
 * @roseuid 3E2DC97C002A
 */
private C_DatabaseManager() throws C_MathTrainerException
{
    try
    {
        Class.forName(MTProperties.getString(
            "C_DatabaseManager.jdbc_driver")).newInstance();

        conn = java.sql.DriverManager.getConnection(
            MTProperties.getString("C_DatabaseManager.connect_string"));
    }
    catch(Exception ex)
    {
        C_MathTrainerException mtEx = new C_MathTrainerException(
            C_MathTrainerException.DATABASE_CONNECTION_FAILED);
        mtEx.initCause(ex);
        throw mtEx;
    }
}

```

Example 4.4: The part of the `C_DatabaseManager`-class handling the connection.

4.2 Implementation on the Basis of the xUML Model

The implementation on the basis of an xUML model uses the automatic code generation features of xUML model compilers. The model compiler evaluates the constructs used in the model and transforms them to executable constructs in the target language, respectively.

4.2.1 Code generation

An xUML model is implemented by executing two steps:

Colouring consists of preparing the model for compiling by marking the xUML artefacts as having to be implemented in a special way. This might not be necessary for all components, but is surely necessary for some items which need to be treated specially.

```

/** Execute an SQL query against the database. After executing
 * the query, the method <code>handleResultSet()</code> of the
 * <code>C_DatabaseResultSetHandler</code> object is called, this can be
 * used to handle the <code>ResultSet</code>.
 * @param query The string used as a query.
 * @param handler The interface object being called for handling the
 * <code>ResultSet</code>.
 * @throws C_MathTrainerException
 * @roseuid 3E2DC97C00D4
 */
public static void executeQuery(String query,
                                C_DatabaseResultSetHandler handler)
    throws C_MathTrainerException
{
    try
    {
        if (manager == null)
            manager = new C_DatabaseManager();

        Statement stmt = manager.getConnection().createStatement();

        if (stmt.execute(query))
        {
            ResultSet rs = stmt.getResultSet();
            handler.handleResultSet(rs);
            rs.close();
        }
        stmt.close();
    }
    catch (Exception ex)
    {
        C_MathTrainerException e = new C_MathTrainerException(query);
        e.initCause(ex);
        throw e;
    }
}

```

Example 4.5: The part of the `C_DatabaseManager`-class executing a query.

Compiling is done by taking the model as provided by some repository, reading in the colouring information and transforming the model to source code being platform specific or platform independent. In turn, that source code can then be transformed to binary code (by a standard compiler for the language the intermediary source code output was generated in) which is executable, or e.g. in the case of Java, interpretable.

The process of colouring shall not be explained in detail here as it is different for each of the modelling tools. The process of compiling a model is rather straightforward – the outcome should be generated in a short time. The base for model compiling (what is generated as an intermediate outcome from the xUML model) is not standardized, thus different for each tool. Today, the model compilers work with just one tool, and even worse, there is no way to interchange models among the tools. In [MB02, p. 10], Mellor

```

protected void change(String parent_username) throws C_MathTrainerException
{
    StringBuffer query = new StringBuffer();
    query.append("UPDATE user SET username=");
    query.append(getUsername());
    query.append(" ', password=");
    query.append(password);
    query.append(" ', parent_username=");
    if (parent_username != null)
    {
        query.append(" ");
        query.append(parent_username);
        query.append(" ");
    }
    else
    {
        query.append(" "+null);
    }
    query.append(" _where_ username=_");
    query.append(getUsername());
    query.append(" ");
    C_DatabaseManager.executeQuery(query.toString(), null);
}

```

Example 4.6: The part of the `User`-class committing an UPDATE to the database.

names possible model compilers which are either available or could be designed shortly, but he notes the Enterprise Java Beans compiler as being unavailable (see table 4.1).

<i>Description of the xUML model compiler</i>
Multi-tasking C++ optimized for embedded systems, targeting Windows, Solaris and various real-time operating systems.
Multi-processing C++ with transaction safety and rollback.
Fault-tolerant, multi processing C++ with persistence supporting three processor types and two operating systems.
C, straight on to an embedded system, with no operating system.
C++, widely distributed discrete-event simulation, Windows and UNIX
Java byte code for single-tasking Java with EJB session beans and XML interfaces
Handel-C and C++ for system-level hardware and software development
A directly executing xUML virtual machine.

Table 4.1: Possible model compilers for xUML (not all of them are currently available)[MB02, p. 10]

As no Java code could be created from the model, C++ code was generated instead. This was done using a C++ code generator supplied by Kennedy Carter Ltd., a company specialized on xUML tools. The used compiler supports no persistency. Hence, the database access is not shown in the generated code. The basic principle for this code generation is to evaluate every single construct of the ASL and appropriately transform it to the target language. Additionally to the pure transformation, some directives

provide the framework for the execution of the code. Example 4.7 shows an extract from the output being generated, where the corresponding ASL-statements are included as comments.

```

void
D_D3_COR::SCN_1()
{
    //Local variable
    OLD3_COR_O8_OP_IH VAR_op_plus;
    /* ASL 2 : op_plus = create OPERATOR with name = "+"
    */
    { VAR_op_plus = D3_COR_O8_OP->create_instance( "+" ); }
    /* ASL 7 : local_admin_user = \
    create USER with username = "admin" & \
    password = "admin" & \
    Current_State = 'Logged_Out'
    */
    { VAR_local_admin_user = D3_COR_O2_USR->create_instance(
    "admin" );
    VAR_local_admin_user->password_write("admin");
    VAR_local_admin_user->current_state = 3;
    }
    /* ASL 16 : link
    local_admin_user R1 local_admin_teacher
    */
    VAR_local_admin_user->R1.link_super_sub( VAR_local_admin_user ,
    VAR_local_admin_teacher);

    /* ASL 20 : generate USR2: Create_Student("student","student",
    "teacher")
    */
    Q->generate(new E_D3_COR_O2_USR_Create_Student(D3_COR_O2_USR, NULL, 2, 2,
    "student", "student",
    "teacher"));
}
// Non-Counterpart Terminator Services
// Terminator Events

```

Example 4.7: An extract of the generated code being the output of an xUML model compiler. The corresponding ASL statements are shown as comments.

4.2.2 The Graphic User Interface Layer

Connecting this code to a GUI is not difficult and could be achieved using any major C++ API. This assumption is based on the fact that the bridge to the GUI was designed in a straightforward way and decoupling of the user interface and the business objects was achieved. An example provided by Kennedy Carter Ltd. worked in cooperation with Microsoft Visual C++ and the MFC classes². Even connecting this C++ code

²This example was obtained in written communication with Kennedy Carter Ltd.

to a Java GUI would be possible, using the JNI (Java Native Interface) as the inter-connection between the application and the user interface. Still, this would lead to a mixture of programming languages and technologies being hard to maintain and is not recommended. Waiting until a Java model compiler is commercially available will be necessary to do serious work generating Java code from small xUML models.

4.2.3 The Persistence Layer

Persistency is a question of how the model compiler is configured. It depends on which form of persistence is used (e.g. an object-oriented database or a relational database) and, additionally, when the data has to be persisted. This could be

1. after each attribute update,
2. at the end of each action occurring in a state or
3. at the end of each executing thread.

Then the code generator would have to be configured to insert the respective method calls to the persistency mechanism at the points being defined³. There are model compilers available being configured to add persistency, the one used for creating the source code for the *MathTrainer* example did not offer such possibilities. Therefore the source code shown in 4.7 does not contain calls to the persistency layer.

4.3 Summary

The implementation based on the UML approach led to a slowly shifting focus from the design to the implementation phase. By iteratively generating code from interaction diagrams, further enhancing this code and reverse engineering it to the diagrams modelling and coding went step by step. However, in none of the available tools full code generation from interaction diagrams is achieved. Hence, after each iterative cycle the programming lines which were added so far have to be re-entered. Future tools will solve this problem and enable full code generation providing a smooth transition between the design and the implementation phase.

Not all of the software system was modelled in UML, the part of the GUI and the persistence layer was added by external means. In the first case this was done by using

³Based on written communication with Kennedy Carter Ltd, February 19th, 2003.

a GUI builder, in the second case by manual programming. As these two components can add up to substantial parts of the programming code and the code generation is restricted to the components modelled in UML, the benefits of code generation for the implementation depend on the relation of GUI/persistency code to the rest. However, the seamless integration of the GUI part and the persistency code with the other parts make it possible to visualize these components as well, providing benefits for both documentation and maintenance.

The xUML approach started its implementation phase with the colouring of the class model emerging from the design phase. Then the model compiler was configured and transformed the model into executable code, in this case based on C as a programming language. The problem for small projects is that the model compiler has to be readily available, which was not the case with a Java model compiler; else the targeted language can not be generated. The integration with a Java GUI would work using the Java Native Interface, but would possibly lead to a fragile architecture. Integrating with a C++ GUI is possible, but was not undertaken for the *MathTrainer* example. The unavailable Java model compiler makes it hard to interconnect GUI and business layer for the example. The persistence layer is created by properly configuring the model compiler which was – again – not readily available, the compiler subsequently adds the desired statements at the indicated locations. These locations could be after each action specification, when the information of an attribute of the object has changed or when a thread execution is finished. As the unavailability of a model compiler lead to code being connected neither to the GUI nor to the persistence layer, the model was executed in a model simulator. This model simulator allows to define test sequences and to run the model through these, providing a way to correct and verify the model.

Both approaches have their advantages and disadvantages in the implementation phase. Table 4.2 shows how much time each of the approaches took in the implementation phase and how much of the code could be generated for the different parts of the system.

	UML	xUML
Time necessary	5 days	1/2 days
Amount of code generated for GUI	0%	no GUI used
Amount of code generated for persistence layer	0%	no persistence layer used
Amount of code generated for business layer	80%	100%

Table 4.2: An overview of the time necessary and the amount of generated code for the implementation phase.

Chapter 5

Summary, Results and Future Work

To conclude the project, the following sections provide a summary and an outlook on both other projects and future work to be done.

The project used a *MathTrainer* system for elementary school students to illustrate the phase of Object-Oriented Analysis and Design in the development process and subsequently, to show the possibility of generating substantial amounts of source code from UML models. In the OOAD phase, both the UML and the xUML profile were used to model the software system – exemplifying the similarities and differences of the two approaches. Whereas use case diagrams, written use cases, activity diagrams and class diagrams were used in both approaches, the dynamic aspect of the system was for the UML approach modelled in interaction diagrams, particularly sequence diagrams, for the xUML approach in state chart diagrams. These state chart diagrams were designed for the classes having a lifecycle with interesting aspects for the software development. The notation of xUML diagrams differs not too much from the one of the UML standard and developers accustomed to the UML can easily model systems in xUML, as well. Finally, the xUML state charts were further refined using action specification statements. These action specifications document the behaviour of the object upon entry in a state and are used to fully generate the dynamical aspect of the code. For specifying these actions, a special notation termed ASL was used, there is no standardized language for the action specification.

In the implementation phase, the code generation approaches were applied to the designed models. For the UML approach, the code was then connected to a Java GUI and to a relational database and a Java application was developed. For the xUML approach, the code could not be generated in Java and the connection to the relational database was not possible, the target language had to be changed to C++ and no persistency code was generated. The reason for this was the unavailability of a pre-

configured model compiler able to achieve the desired code generation. Hence, the model was connected neither to the GUI nor to the relational database. However, it was executed in a model simulator allowing the definition of test sequences and driving the model through these test sequences. By executing the model in the simulator, the model was iteratively corrected and verified.

5.1 Results

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. As Java is not a very uncommon programming language as of today the unavailability of an out-of-the box Java solution for xUML can be interpreted as a major weakness of this approach, at least for small projects¹. When the necessary model compiler is available, the approach of xUML is promising. The inherent program language and platform independency of this form of software development might provide exactly the additional step of abstraction necessary to drive the IT industry to a higher layer. Still, the xUML advocates mandate not to change compiled code ([MB02]), so many software developers good in writing program

¹Java is not generally unavailable for the xUML approach as sophisticated code generators exist on the market providing the generation of any language, even Java. Problematic are the expenses needed for the configuration of such a code generator. For the size of the project used as the example in this thesis they would be unaffordable – so the rule in this comparison must be that the code generator ought to be readily available, or it cannot be used. For an IT company having dozens of projects of a larger size running, the obstacle of configuring a code generator to produce Java code might be a minor one.

code today would have to learn something totally new – a change which might not happen for years.

5.2 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. [SFL98] 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.

In [EHSW99], an approach to enhance the semantics of collaboration diagrams beyond the standard of UML is presented. This approach restricts itself to collaboration diagrams; furthermore, the authors argue that code generation from sequence diagrams is not viable. As the availability of *ControlCenter* is the proof for working code generation from sequence diagrams, this point of view has to be reconsidered.

Finally, the documentation delivered with Together ControlCenter provides a practical overview about how the code generation from sequence diagrams works with this tool [CCG02]. Together is very much in scope with the UML standard with the graphical notation used for the sequence diagrams and provides many of the constructs other tools do not support, making code generation from sequence diagrams possible. A small example of code generation from sequence diagrams using *ControlCenter* is also provided by [Lar02, p. 572].

For the second topic, [MB02] provides a good insight to modelling using xUML (and also an example using xUML, but no comparison to the same example solved by using UML). Still, the code generation using xUML is not very widespread, so it might be not so easy building a code generator for the desired project.

5.3 Future Work

Four proposals shall be made for future work on the presented topics.

Some of the most significant advantages introduced by xUML are, according to [MB02]:

1. platform independency and
2. language independency.

All of the action languages (or action specification languages) available today in tools for modelling with xUML are proprietary developments by the providers of these tools. Hence, what a developer has to do first when starting to model in xUML is to learn the syntax of this language.

The advantage of using Java as an action specification language would be to remove the obstacle of learning the syntax of a new language (like ASL). Additionally, the Java API could be used, which is arguably one of the best available today. The disadvantages would be that Java might first not be the best approach to write the necessary constructs being provided by an action language². Furthermore, the notion of *language independency* would fall³. For the community of Java developers this disadvantage could possibly be neglected.

For the xUML to be a success, it is necessary to standardize the output generated from models⁴. With output not the compiled output is meant, but the intermediate output used as a basis for the compilation. From this intermediate, standardized output a model compiler could generate the executable code in any desired language.

As Java would be a great destination language due to its platform independency and its interaction capabilities with many components available, an open source EJB model compiler would bring a leap forward for both the acceptability and usability of xUML.

Another leap forward could be made by the inclusion of complete code generation in an open source UML diagramming tool. There are such tools available today, but they generally lack code generation capabilities beyond the static structure generated from a *class diagram*. Enhancing such a tool, an example might be the Java ArgoUML tool, would be a great way to increase the usage of code generation from UML diagrams.

²An example would be the `link`-construct which would have to be mapped to a method call, as well as the signal generation would have to map to a method call, e.g. using a `appendSignal()`-syntax.

³As it might be hard to compile Java code into another language, even if this could be achieved.

⁴According to [MB02], the tool industry tries hard to accomplish that .

This could happen for both approaches presented above, either for code generation from sequence diagrams or for code generation using the xUML profile.

Both approaches to complete code generation presented above have their advantages and disadvantages: combining the approaches would enable a new way of modelling and automatic code generation and visualization so far not been possible. As an example, the often very algorithmic approach to *action languages* means that the actions written in these languages could also be represented in *sequence diagrams*. This is certainly true with the ASL language used in iUML, but also with many others. An automatic mapping between these two ways of defining actions is further simplified by the strict object-orientation of many of the action languages. Such a mapping would increase the readability and comprehensibility of action definitions by far. Additionally, the pressure of learning such a language could be eased for the developer as the visualized syntax would be easier to understand.

Appendix A

Additional Use Case Descriptions

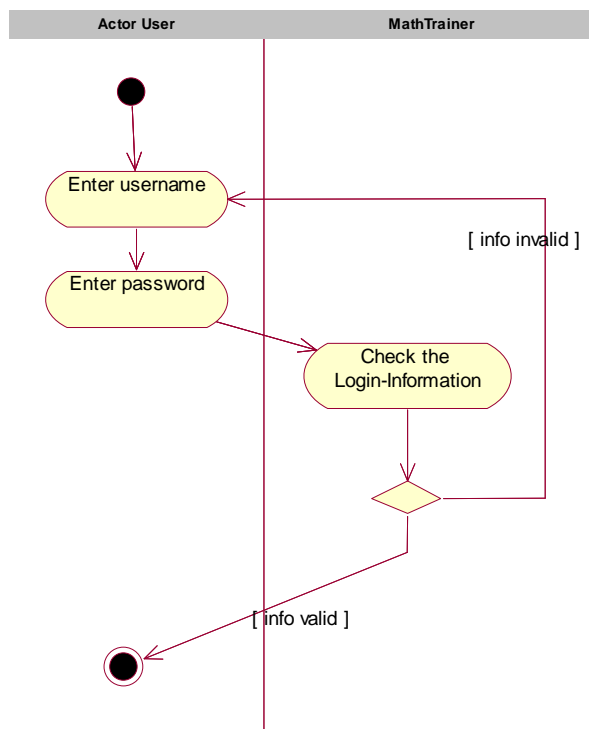


Figure A.1: Activity diagram for the *Identify User* use case.

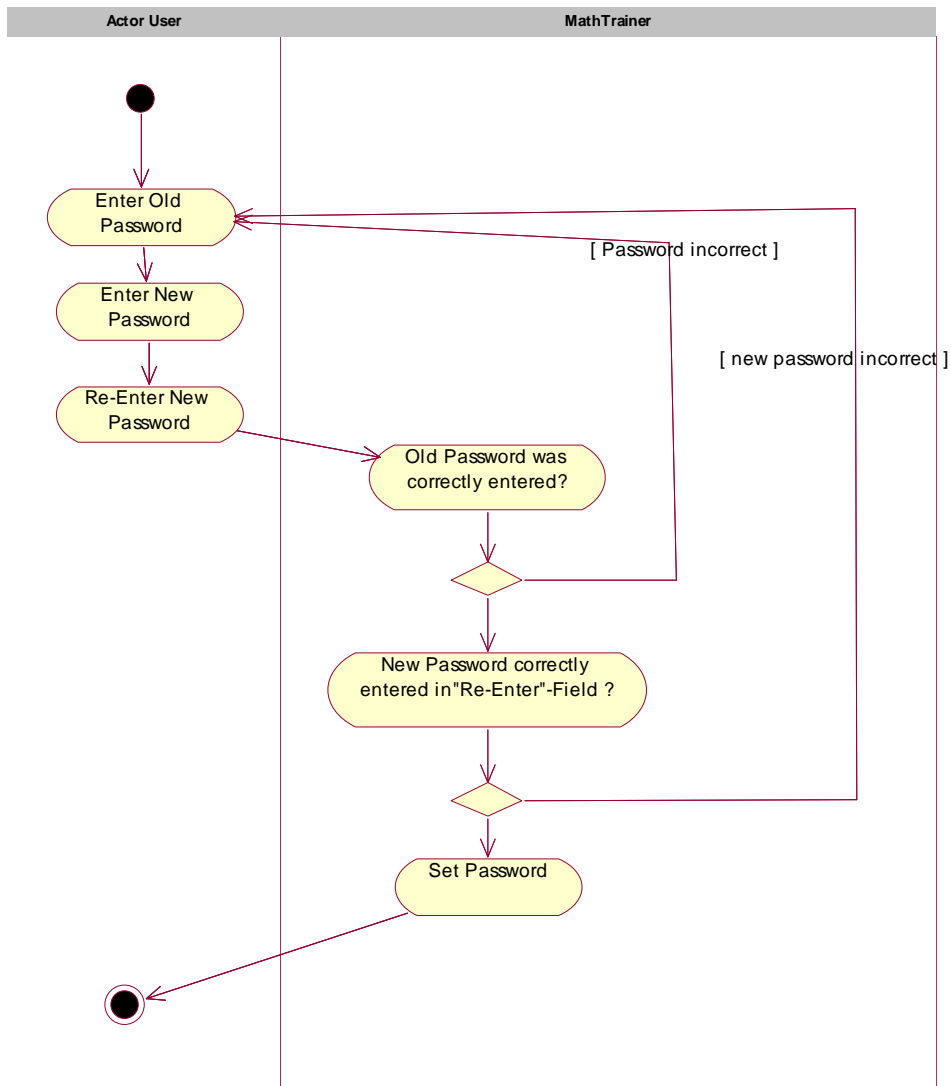


Figure A.2: Activity diagram for the *Change password* use case.

Description	Before any action can be taken in the <i>MathTrainer</i> system, the users have to be logged in. This is done by providing username and according password.
Precondition	The user has been created in the <i>MathTrainer</i> system and has access to the <i>Login</i> context.
Postcondition	The user's state has changed to being logged in, he can set desired actions.
Error conditions	<ol style="list-style-type: none"> 1. The username is unknown to the system 2. The password is wrongly entered.
Error postcondition	The user is not logged in.
Actors	User (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Username and password are entered. 2. The username is known to the system and the password is correct. 3. The user's state is changed to logged-in. 4. Possible actions are presented to the user.
Deviation 1	<ol style="list-style-type: none"> 2'. The username is not known or the entered password is not correct. 3'. An error is shown to the user and all system objects retain their state. 4'. The user is directed to the <i>Login</i> context.

Table A.1: The written story for the *Identify User* use case.

Description	The user's password can be changed at any time. This is done by entering the old password and the new password twice.
Precondition	The user is logged-in.
Postcondition	The password of the user has changed.
Error conditions	<ol style="list-style-type: none">1. The entered old password was wrong.2. The entered new passwords do not match.
Error postcondition	The password was not changed
Actors	User (primary actor)
Standard procedure	<ol style="list-style-type: none">1. Old password and new password are entered, new password twice.2. The entered old password matches the user's password and the entered new passwords match.3. The password is changed.
Deviation 1	<ol style="list-style-type: none">2'. The entered old password do not match the user's password or the entered new passwords do not match.3'. An error is shown to the user and all system objects retain their state.

Table A.2: The written story for the *Change Password* use case.

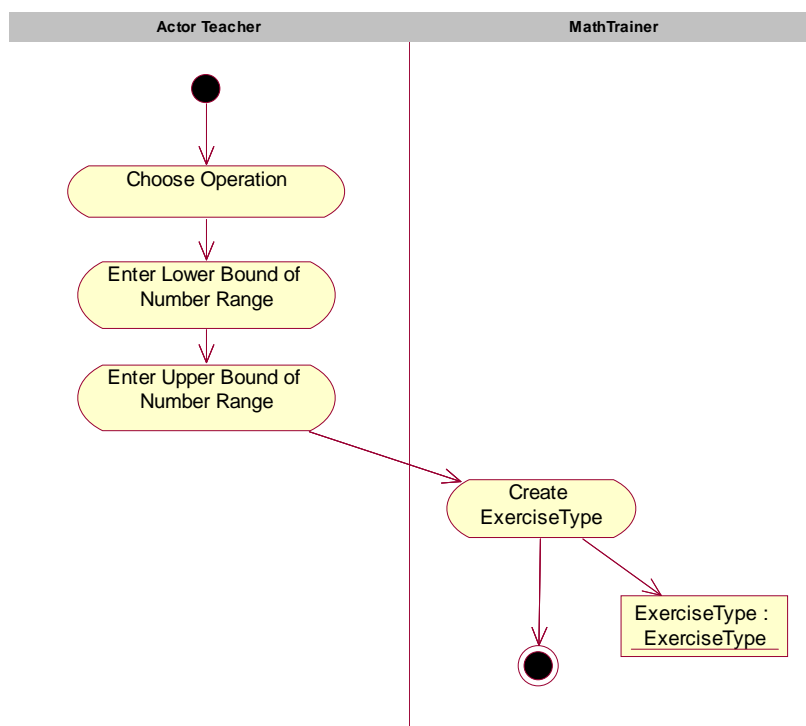


Figure A.3: Activity diagram for the *Create Exercise Type* use case.

Description	A teacher can create exercise types. These types then define what the exercises look like when a student solves an exam.
Precondition	The teacher is known to the system and logged-in.
Postcondition	A new exercise type has been created. The exercise type possesses a unique name, an upper and a lower bound for its range of numbers and at least one mathematical operation.
Error conditions	The name is already used by an existing exercise type.
Error postcondition	The exercise type was not created.
Actors	Teacher (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Upper and lower bound for the number range are chosen. 2. Mathematical operations are chosen, can be happen for each available mathematical operation consecutively. 3. The exercise type is created in the system.
Deviation 1	<ol style="list-style-type: none"> 2'. The name of the exercise type is already used. 3'. An error is shown to the teacher and all system objects retain their state.

Table A.3: The written story for the *Create Exercise Type* use case.

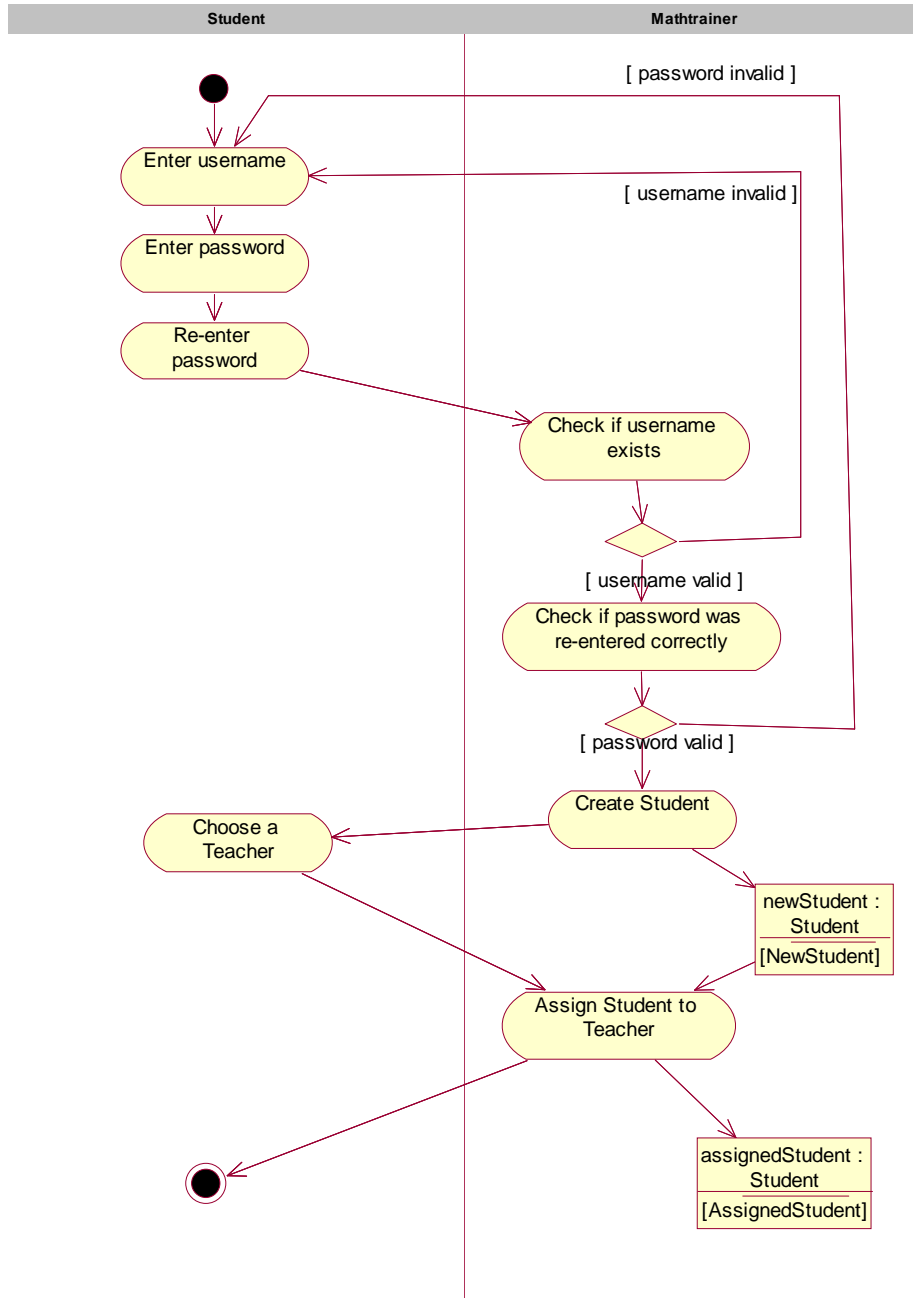


Figure A.4: Activity diagram for the *Create Student* use case.

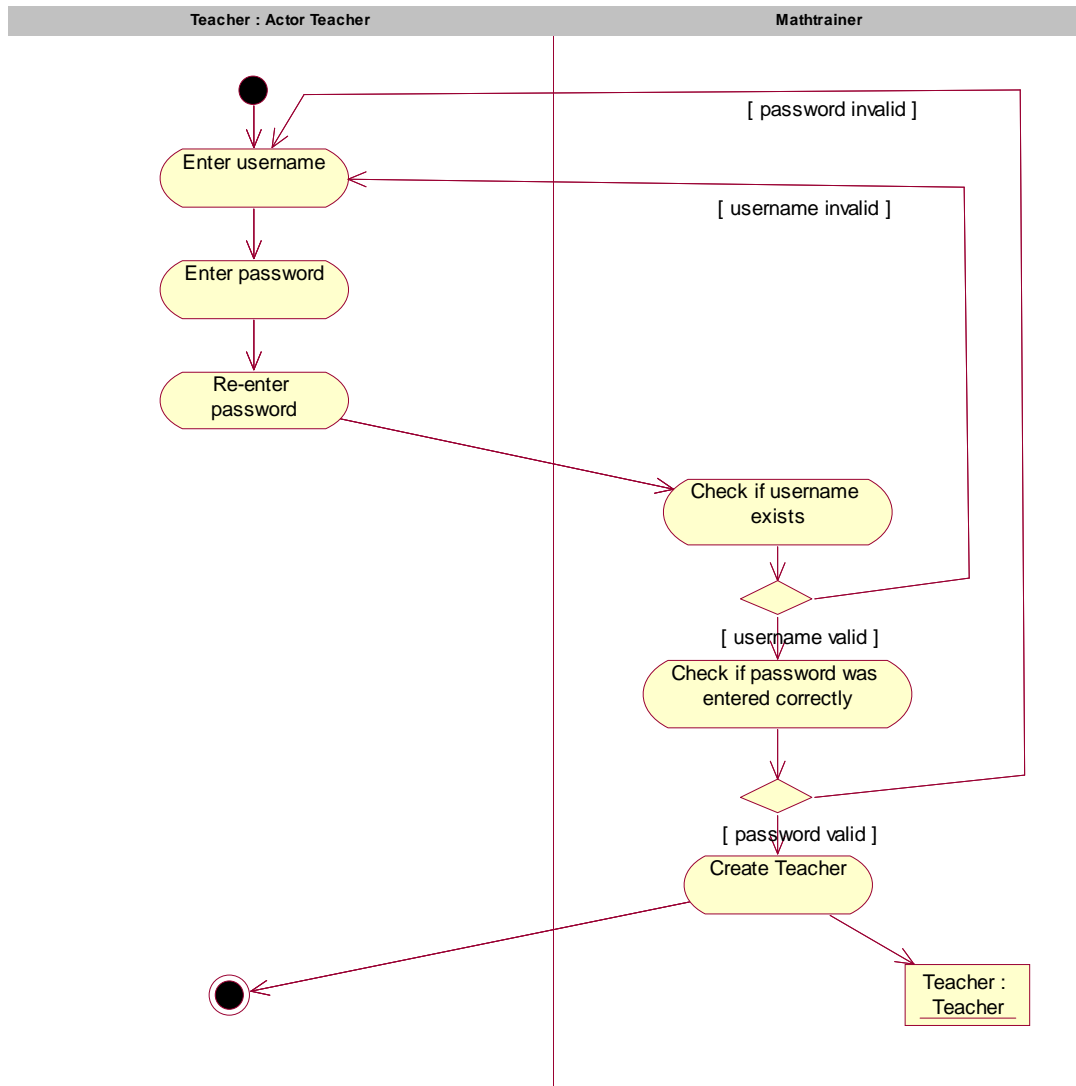


Figure A.5: Activity diagram for the *Create Teacher* use case.

Description	A teacher can only be created by an existing teacher of the <i>MathTrainer</i> system. Necessary information are username and password which has to be entered twice.
Precondition	The existing teacher is known to the system and logged-in.
Postcondition	A new teacher has been created. The teacher possesses a unique username and a password.
Error conditions	<ol style="list-style-type: none"> 1. The provided username is already given to a user of the <i>MathTrainer</i> system. 2. The entered passwords do not match.
Error postcondition	The teacher was not created.
Actors	Teacher (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Username and password are entered, password twice. 2. The username is not yet used by the system and the entered passwords match. 3. The teacher is created in the system.
Deviation 1	<ol style="list-style-type: none"> 2'. The username is already in use or the entered passwords do not match. 3'. An error is shown to the existing teacher and all system objects retain their state.

Table A.4: The written story for the *Create Teacher* use case.

Description	An exercise type is deleted.
Precondition	A teacher is logged-in to the system.
Postcondition	An exercise type object is deleted.
Error conditions	none
Error postcondition	none
Actors	Teacher (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Exercise Types of the current teacher are shown. 2. Teacher chooses one of the types. 3. The chosen type is deleted.

Table A.5: The written story for the *Delete Exercise Type* use case.

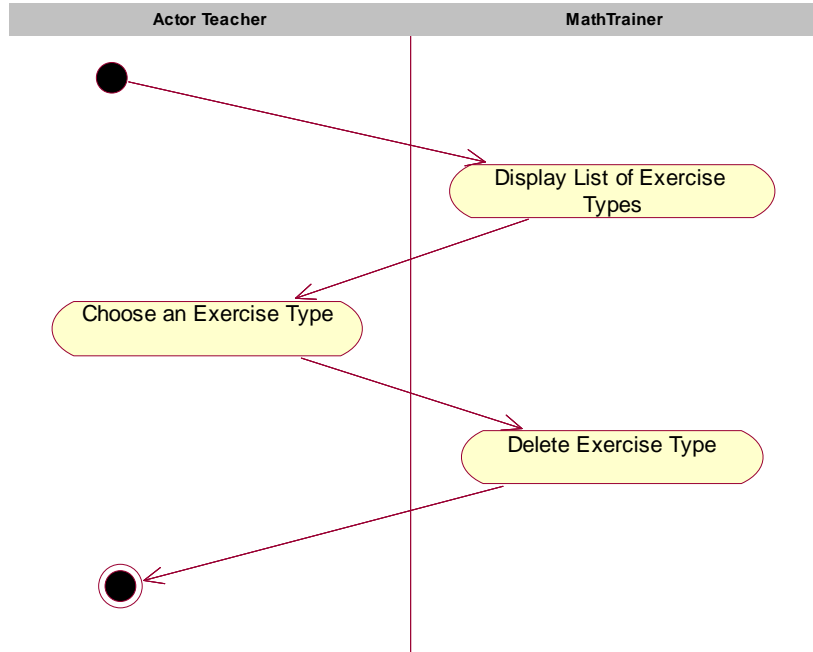


Figure A.6: Activity diagram for the *Delete Exercise Type* use case.

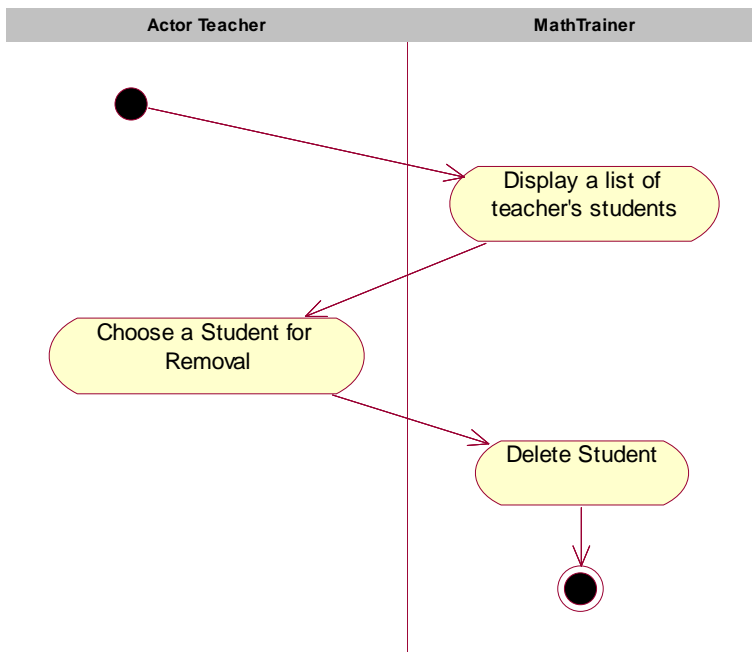


Figure A.7: Activity diagram for the *Delete Student* use case.

Description	A student is deleted.
Precondition	A teacher is logged-in to the system.
Postcondition	An student object is deleted.
Error conditions	none
Error postcondition	none
Actors	Teacher (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. Students of the current teacher are shown. 2. Teacher chooses one of the students. 3. The chosen student is deleted.

Table A.6: The written story for the *Delete Student* use case.

Description	Exams are created upon request of the student; the student requests by choosing an exercise type. They consist of ten randomly created exercises.
Precondition	The student is known to the system and logged-in.
Postcondition	A new exam has been created and solved by the student.
Error conditions	The student has entered non-numerical values for the solutions.
Error postcondition	The exam object was created and subsequently deleted.
Actors	Student (primary actor)
Standard procedure	<ol style="list-style-type: none"> 1. A list of exercise types is shown to the student 2. The student chooses one of them. 3. An exam and its ten random exercises are created. 4. The student solves the ten exercises subsequently, the system measures the time. 5. The exercises were all solved by entering numerical values. 6. The exam object is retained.
Deviation 1	<ol style="list-style-type: none"> 5'. One of the exercises was solved by entering a non-numerical value. 6'. An error is shown to the student and all system objects retain their state. 7'. The exam object is discarded.

Table A.7: The written story for the *Solve Exam* use case.

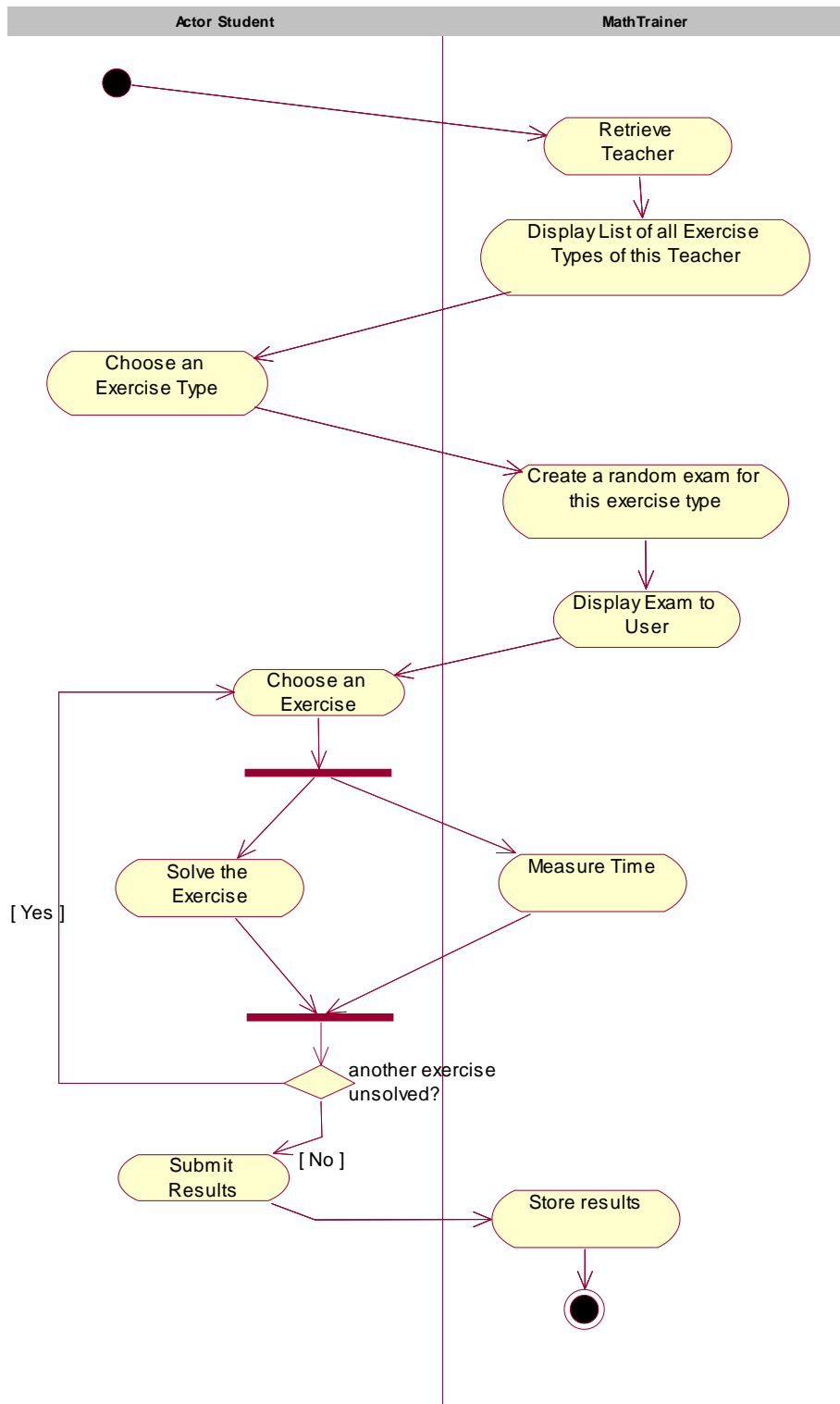


Figure A.8: Activity diagram for the *Solve Exam* use case.

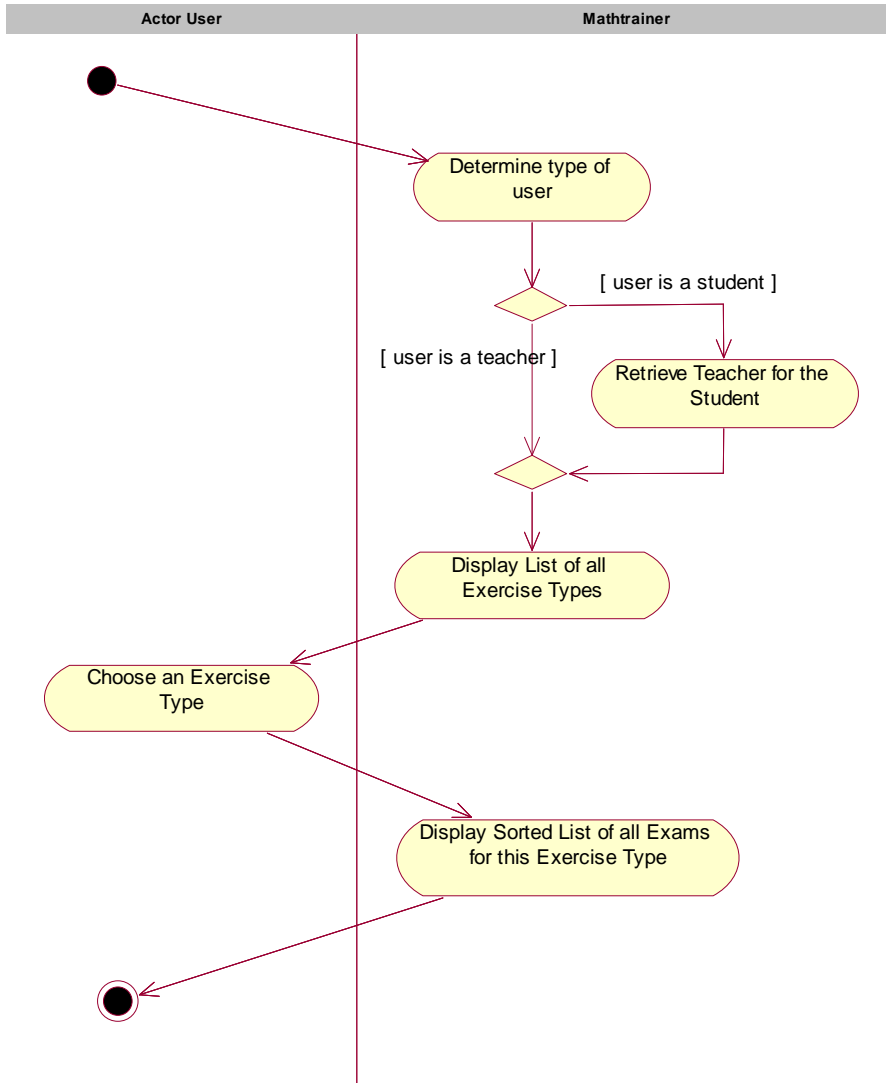


Figure A.9: Activity diagram for the *Show Score* use case.

Description	Users can request to view high scores for a special exercise type.
Precondition	The user is known to the system and logged-in.
Postcondition	The high score has been shown, all system objects retained their state.
Error conditions	none
Error postcondition	none
Actors	User (primary actor)
Standard procedure	<ol style="list-style-type: none">1. A list of exercise types is shown to the user.2. The user chooses one of them.3. A score table is created and shown to the user.

Table A.8: The written story for the *Show Score* use case.

Appendix B

Additional Sequence Diagrams

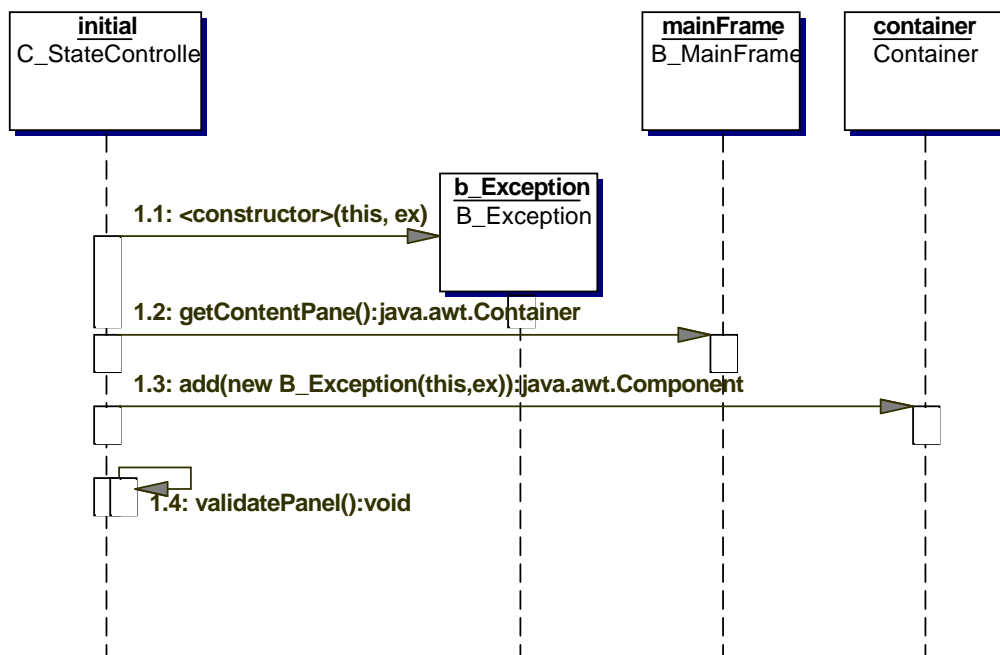


Figure B.1: Sequence diagram of showing an exception in the GUI by calling `C_StateController.showThrowable()`.

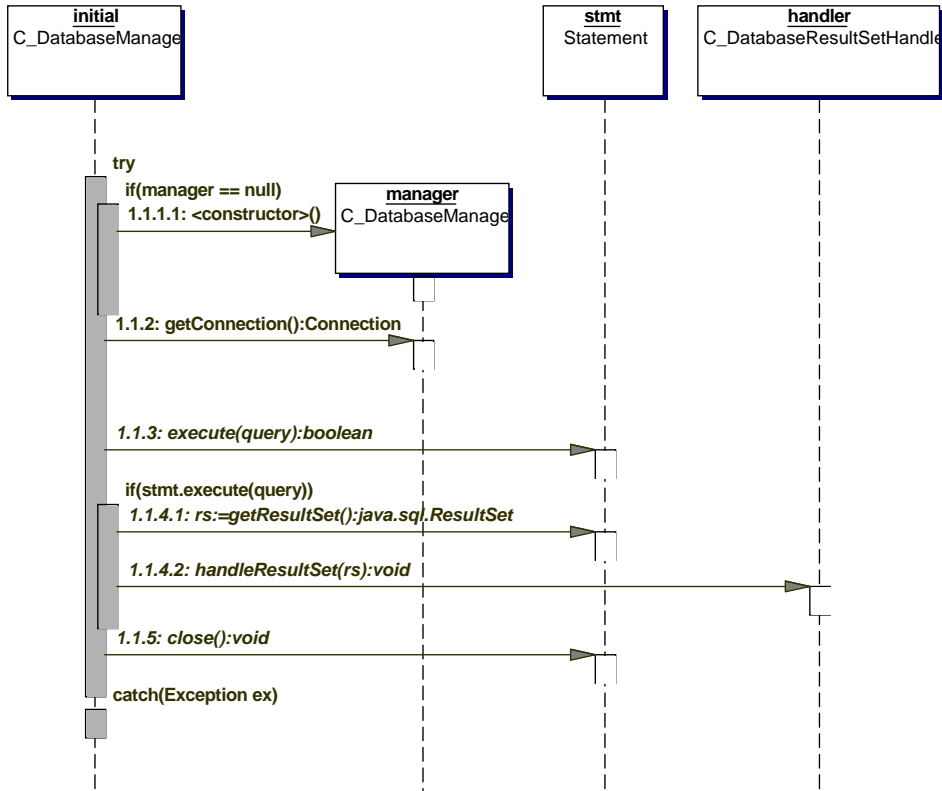


Figure B.2: Sequence diagram of executing a query by calling DatabaseManager.executeQuery().

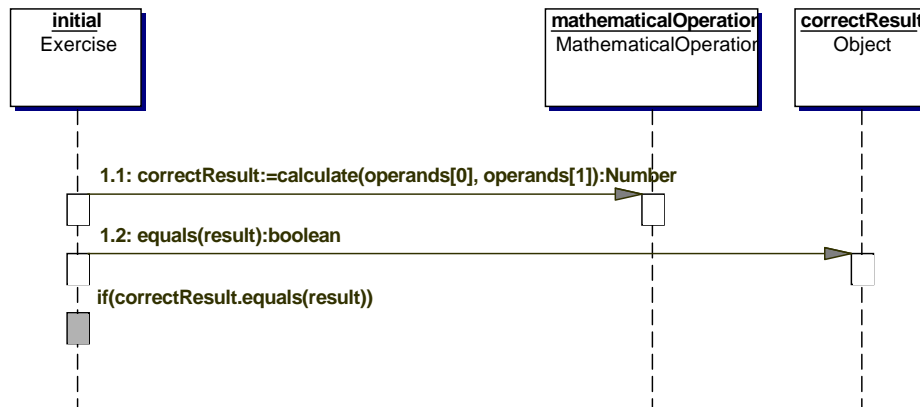


Figure B.3: Sequence diagram of solving an exercise by calling Exercise.solve().

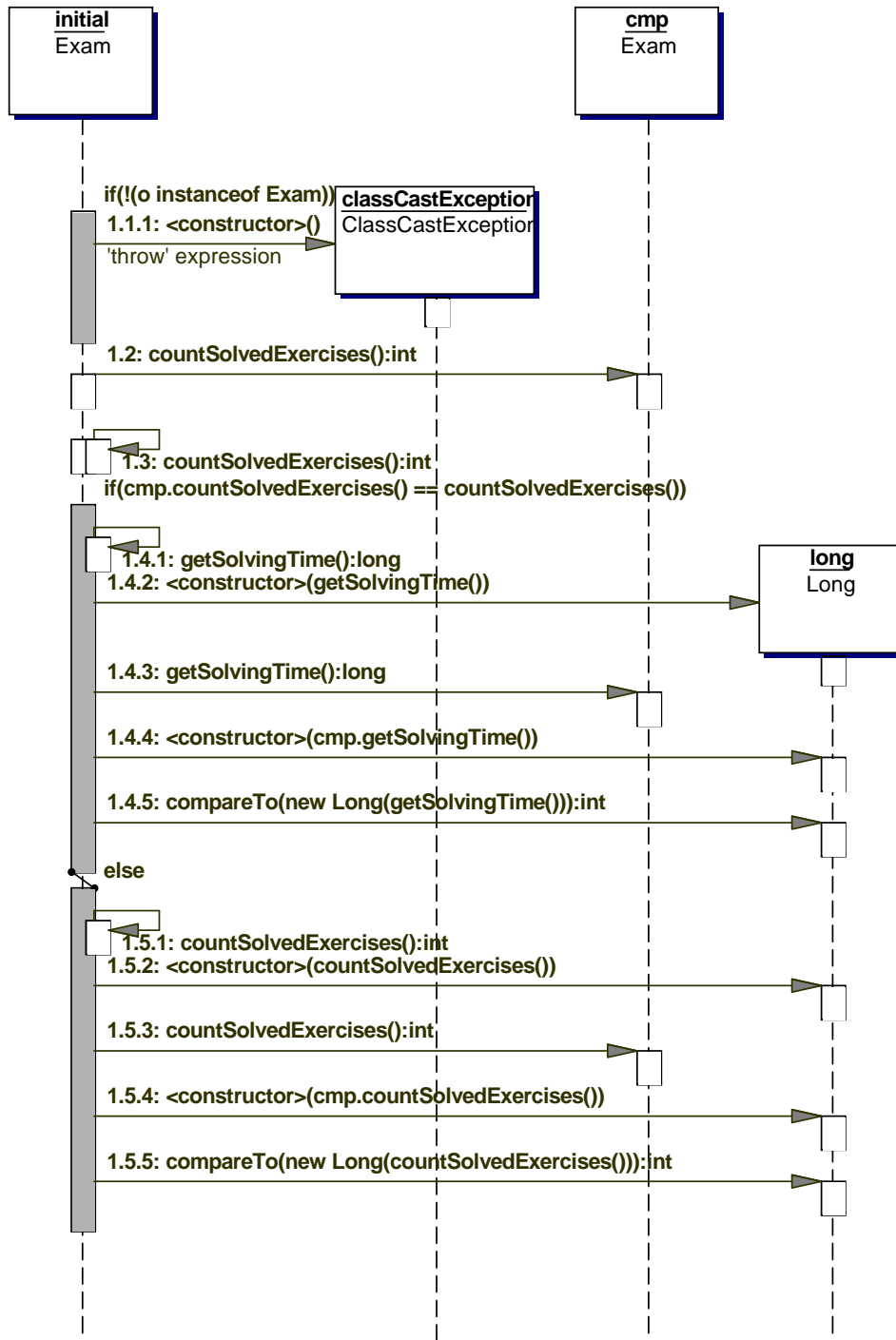


Figure B.4: Sequence diagram of comparing exams by calling Exam.compareTo().

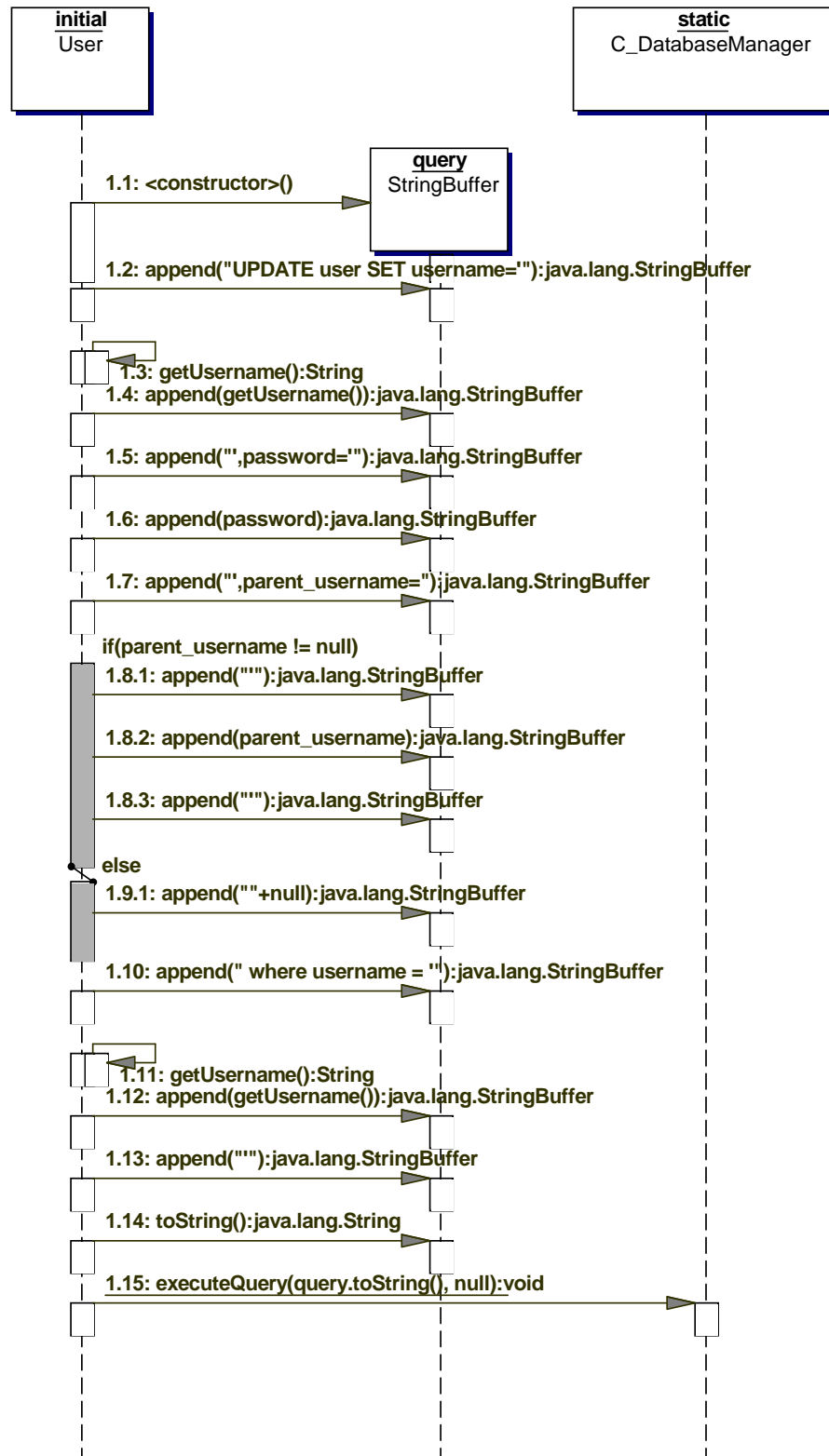


Figure B.5: Sequence diagram of changing a user by calling `User.change()`.

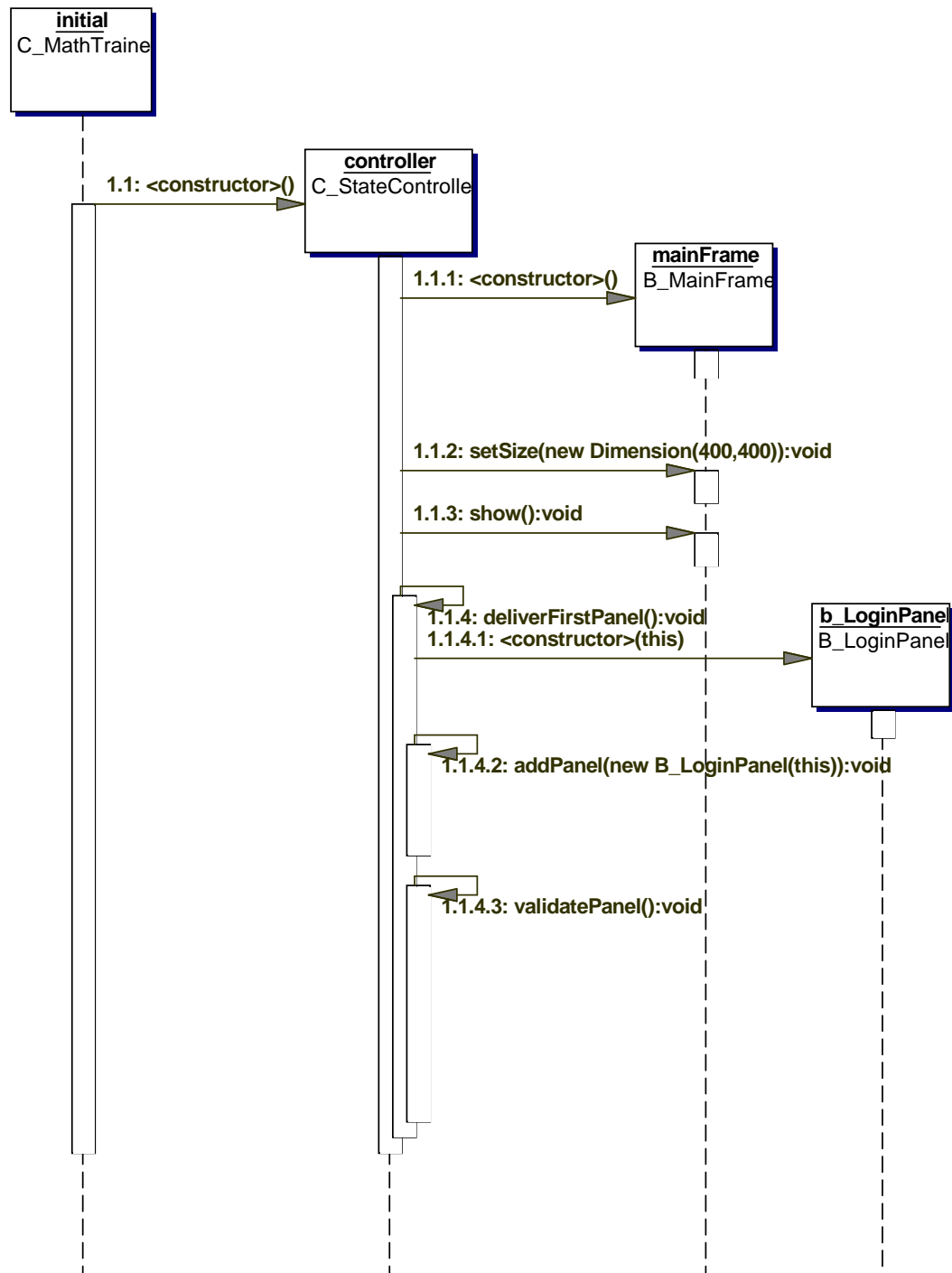


Figure B.6: Sequence diagram of the `startInstance()` method of the class `C_MathTraine`.

Appendix C

Additional State-Chart Diagrams

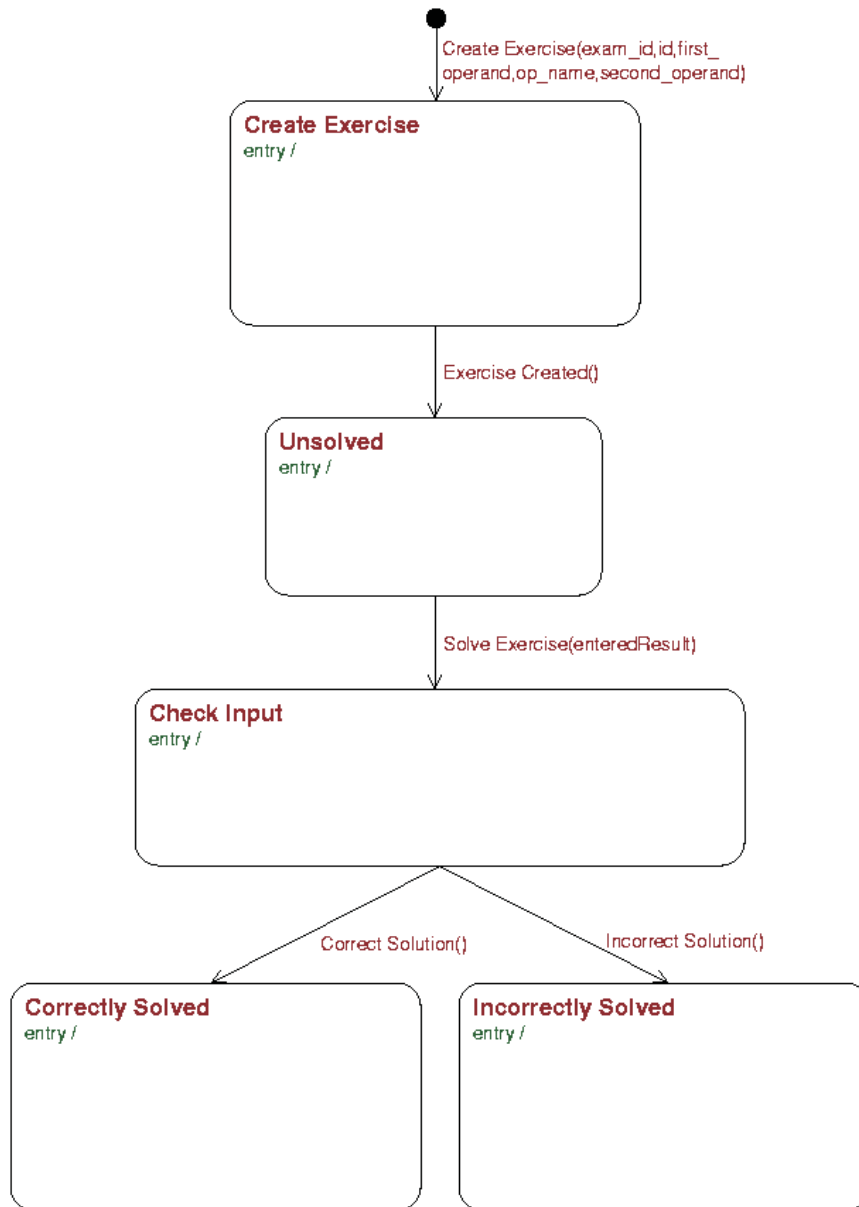


Figure C.1: xUML state chart of the Exercise class.

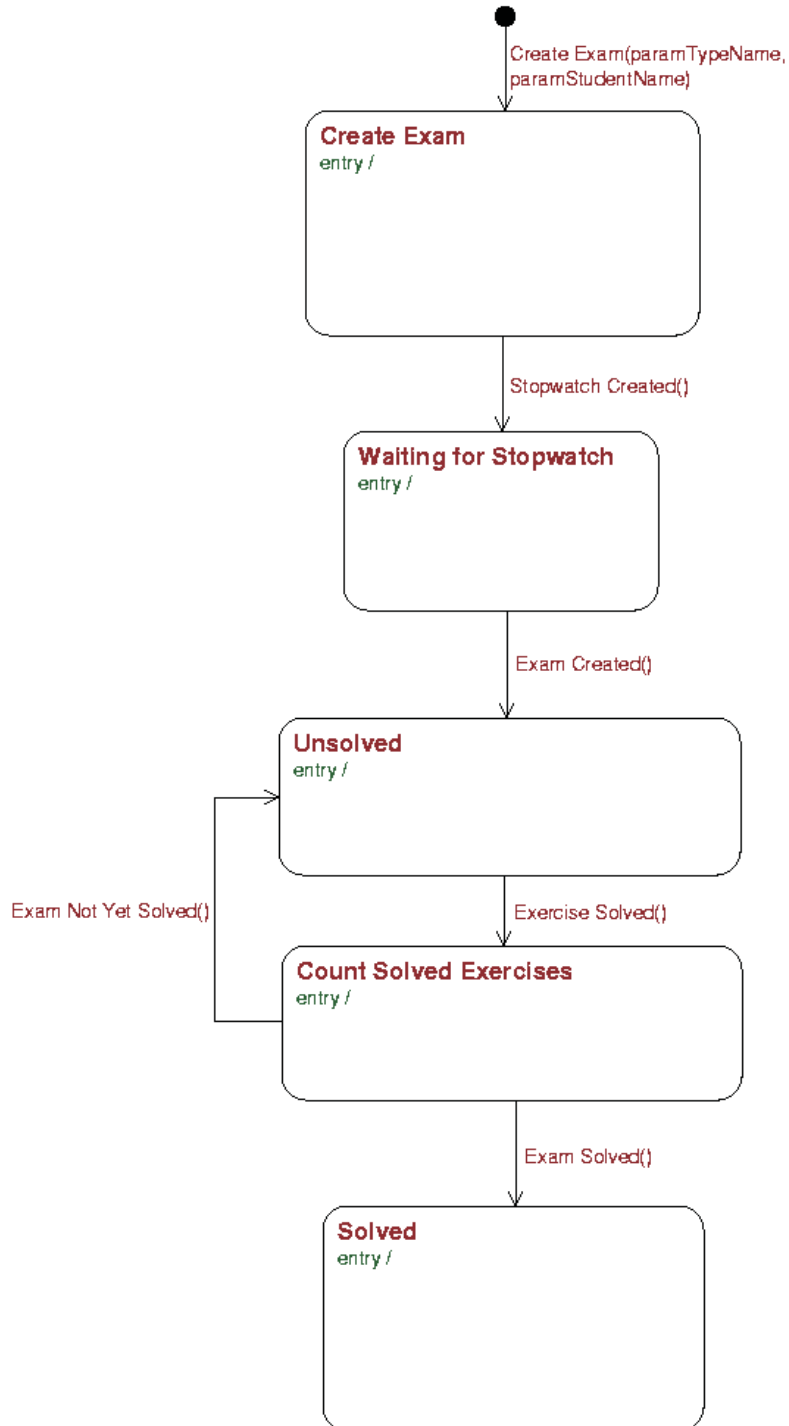


Figure C.2: xUML state chart of the Exam class.

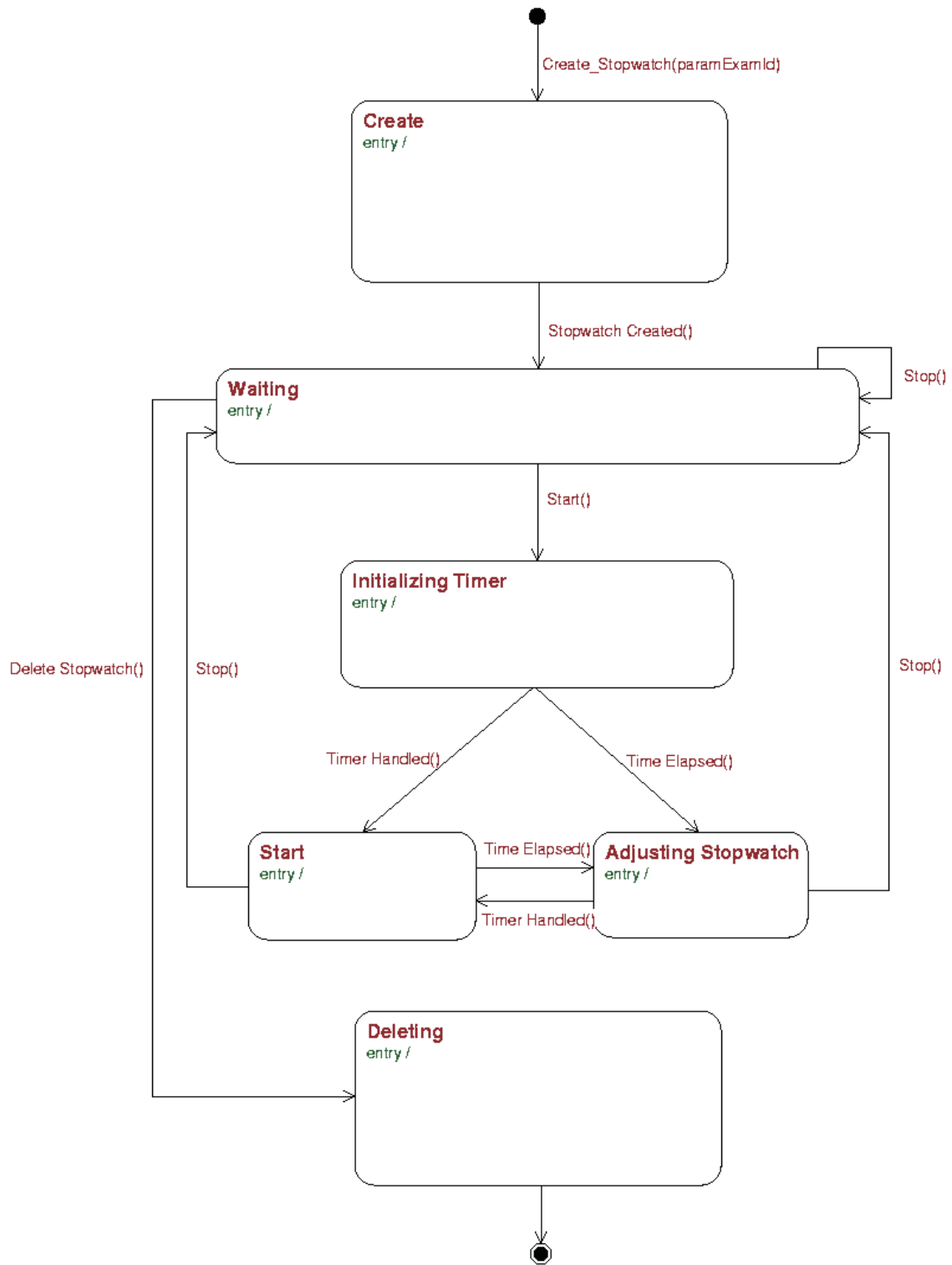


Figure C.3: xUML state chart of the Stopwatch class.

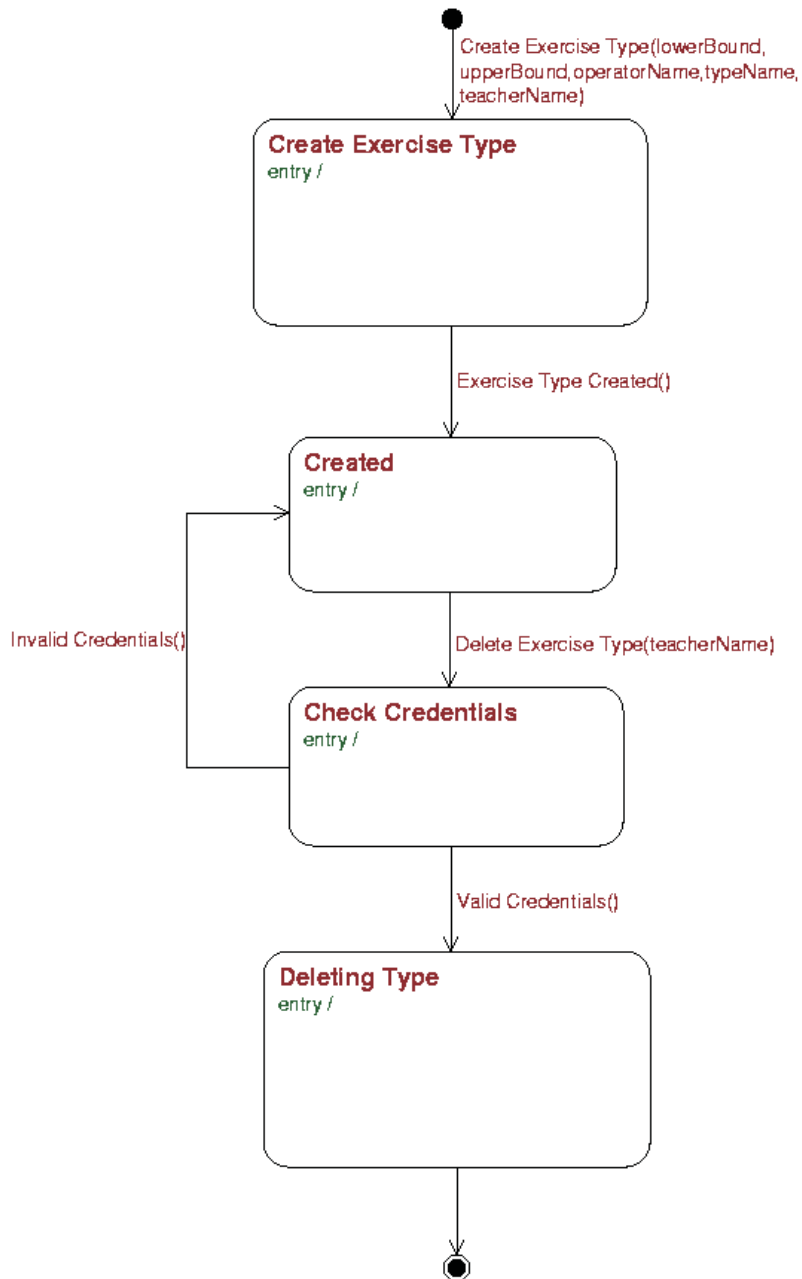


Figure C.4: xUML state chart of the ExerciseType class.

List of Figures

1.1	The requirements for the MathTrainer - example	3
2.1	The phases of OOAD and how the transition from OOA to OOD works – based on the insights of [Kai99].	7
2.2	The origin and descent of UML, taken from [Kob99] and extended with the latest developments.	9
2.3	The packages of UML and their interdependencies [UML03].	11
2.4	xUML is a subset and a superset of UML as of version 1.4 [\Rightarrow xUML02, p. 9].	15
2.5	The Java platform consisting of Java VM and Java API [CWH01].	22
3.1	The phases of the Rational Unified Process, based on [Ros01].	25
3.2	Highlighted verbs in the requirements of the MathTrainer - example . . .	30
3.3	The <i>use case</i> diagram for the MathTrainer example.	33
3.4	The <i>activity diagram</i> MathTrainer’s <i>View Scores</i> use case.	35
3.5	Highlighted nouns in the requirements of the MathTrainer - example . . .	39
3.6	The domain model of the MathTrainer example, including numbers as a conceptual class.	41
3.7	The domain model of the MathTrainer example.	42
3.8	The static structure of the modified EBC pattern.	45
3.9	The construction phase of the modified EBC pattern’s dynamic behaviour.	46
3.10	The data entry phase of the modified EBC pattern’s dynamic behaviour.	47

3.11	The design class diagram of the MathTrainer example.	53
3.12	The sequence diagram for solving an exam in the MathTrainer example. .	54
3.13	The collaboration diagram for solving an exam in the MathTrainer example.	55
3.14	The xUML <i>domain model</i> of the MathTrainer example.	57
3.15	The xUML <i>class diagram</i> of MathTrainer's <i>Core</i> domain.	59
3.16	How an arithmetical exercise, its operator and the operands interact. . .	61
3.17	The xUML collaboration diagram for the MathTrainer's <i>Core</i> domain . .	63
3.18	xUML state chart of the <code>User</code> class.	65
4.1	How an unidirectional <code>?:1</code> association can be mapped to Java source code [HK99, p. 270]	72
4.2	A sequence diagram automatically being mapped to method bodies. . . .	75
4.3	A sequence diagram that was automatically created by reverse- engineering from example 4.2.	75
4.4	The Swing-window for solving an exam.	77
A.1	Activity diagram for the <i>Identify User</i> use case.	91
A.2	Activity diagram for the <i>Change password</i> use case.	92
A.3	Activity diagram for the <i>Create Exercise Type</i> use case.	95
A.4	Activity diagram for the <i>Create Student</i> use case.	97
A.5	Activity diagram for the <i>Create Teacher</i> use case.	98
A.6	Activity diagram for the <i>Delete Exercise Type</i> use case.	100
A.7	Activity diagram for the <i>Delete Student</i> use case.	100
A.8	Activity diagram for the <i>Solve Exam</i> use case.	102
A.9	Activity diagram for the <i>Show Score</i> use case.	103
B.1	Sequence diagram of showing an exception in the GUI by calling <code>C_StateController.showThrowable()</code>	105

B.2	Sequence diagram of executing a query by calling <code>DatabaseManager.executeQuery()</code>	106
B.3	Sequence diagram of solving an exercise by calling <code>Exercise.solve()</code>	106
B.4	Sequence diagram of comparing exams by calling <code>Exam.compareTo()</code>	107
B.5	Sequence diagram of changing a user by calling <code>User.change()</code>	108
B.6	Sequence diagram of the <code>startInstance()</code> method of the class <code>C.MathTrainer</code>	109
C.1	xUML state chart of the <code>Exercise</code> class.	111
C.2	xUML state chart of the <code>Exam</code> class.	112
C.3	xUML state chart of the <code>StopWatch</code> class.	113
C.4	xUML state chart of the <code>ExerciseType</code> class.	114

List of Tables

2.1	The elements of xUML [MB02, p. 6]	15
3.1	The evaluation of the verb phrases which were identified in figure 3.2	32
3.2	The written story for the <i>Create Student</i> use case.	36
3.3	The evaluation of the nouns which were identified in figure 3.5	38
3.4	An overview of the time necessary for the Object-Oriented Analysis and Design phase.	70
4.1	Possible model compilers for xUML (not all of them are currently available)[MB02, p. 10]	82
4.2	An overview of the time necessary and the amount of generated code for the implementation phase.	85
A.1	The written story for the <i>Identify User</i> use case.	93
A.2	The written story for the <i>Change Password</i> use case.	94
A.3	The written story for the <i>Create Exercise Type</i> use case.	96
A.4	The written story for the <i>Create Teacher</i> use case.	99
A.5	The written story for the <i>Delete Exercise Type</i> use case.	99
A.6	The written story for the <i>Delete Student</i> use case.	101
A.7	The written story for the <i>Solve Exam</i> use case.	101
A.8	The written story for the <i>Show Score</i> use case.	104

List of Examples

3.1	How the factory pattern could be implemented in Java code.	50
3.2	How the Enum pattern could be implemented in Java code.	51
3.3	An if statement in ASL.	66
3.4	Establishing a link, navigating along it and deleting it, written in ASL. .	66
3.5	Syntax for sending signals in ASL.	67
3.6	Handling of sets in a test method written in ASL.	67
3.7	Calling an operation in ASL.	67
3.8	Creating and deleting instances in ASL and changing the position in a generalization relationship.	68
4.1	The code generated by the sequence diagram of figure 4.2.	74
4.2	The code being the base for the sequence diagram of figure 4.3.	74
4.3	An example SQL statement for the creation of the user table in the MySQL system.	79
4.4	The part of the <code>C_DatabaseManager</code> -class handling the connection. . . .	80
4.5	The part of the <code>C_DatabaseManager</code> -class executing a query.	81
4.6	The part of the <code>User</code> -class committing an UPDATE to the database. . . .	82
4.7	An extract of the generated code being the output of an xUML model compiler. The corresponding ASL statements are shown as comments. . .	83

Bibliography

- [Amb02] Scott W. Ambler. *agile modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, New York, first edition, 2002.
- [Bal00] Heide Balzert. *Objektorientierung in 7 Tagen: Vom UML Modell zur fertigen Web-Anwendung*. Spektrum Akademischer Verlag, Heidelberg, first edition, 2000.
- [BKL⁺01] Alf Borrmann, Stefan Komnick, Gunnar Landgrebe, Jan Matèrne, Manfred Rätzmann, and Jörg Sauer. *Rational Rose und UML*. Galileo Computing, Bonn, first edition, 2001.
- [BMR⁺96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, West Sussex, England, first edition, 1996.
- [Boe88] Barry W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.
- [CWH01] 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, 2001.
- [CWHT01] Mary Campione, Kathy Walrath, Alison Huml, and Tutorial Team. *The Java Tutorial Continued: The Rest of the JDK*. Pearson Education, Upper Saddle River, first edition, 2001.
- [Dor02] Dov Dori. Why significant UML change is unlikely. *Communications of the ACM*, 45(11):82–85, 2002.
- [Dud02] Keith Duddy. UML must enable a family of languages. *Communications of the ACM*, 45(11):73–75, 2002.

- [EHSW99] G. Engels, R. Hücking, St. Sauer, and A. Wagner. UML collaboration diagrams and their transformation to Java. In R. France and B. Rumpe, editors, *UML '99 - The Unified Modeling Language - Beyond the Standard.*, pages 473–488. Springer, Berlin, October 28-30 1999. Second Intern. Conference. Fort Collins, CO. LNCS 1723.
- [FT02] William Frank and Kevin P. Tyson. Be clear, clean, concise. *Communications of the ACM*, 45(11):79–81, 2002.
- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, New York, second edition, 1994.
- [GM96] James Gosling and Henry McGilton. The java language environment: A white paper. Technical report, Sun Microsystems, 1996.
- [HK99] Martin Hitz and Gerti Kappel. *UML@Work - Von der Analyse zur Realisierung*. dpunkt.Verlag, Heidelberg, first edition, 1999.
- [Kai99] Hermann Kaindl. Difficulties in the transition from OO analysis to design. *IEEE Software*, 16(5):94–102, 1999.
- [Kob99] Cris Kobryn. UML 2001: A standardization odyssey. *Communications of the ACM*, 42(10):29–37, 1999.
- [Lar02] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall, Upper Saddle River, second edition, 2002.
- [MB02] Stephen J. Mellor and Marc J. Balcar. *Executable UML: A Foundation For Model-Driven Architecture*. Addison-Wesley, first edition, 2002.
- [Mel02] Stephen J. Mellor. Make models be assets. *Communications of the ACM*, 45(11):76–78, 2002.
- [Mil02] Joaquin Miller. What UML should be. *Communications of the ACM*, 45(11):67–69, 2002.
- [MW99] Stephen J. Mellor and Ian Wilkie. A mapping from Shlaer-Mellor to UML. Technical report, Projtech Inc. and Kennedy Carter Limited, 1999.

- [NM01] Eric J. Naiburg and Robert A. Maksimchuk. *UML for database design*. Addison Wesley, Boston, first edition, 2001.
- [OMG02] OMG. UML profile for CORBA specification: April 2002 version 1.0. Technical report, Object Management Group, 2002.
- [Qua03] Terry Quatrani. *Visual Modeling with Rational Rose 2002 and UML*. Addison Wesley, New York, first edition, 2003.
- [Ros01] Rational Rose. The rational unified process. Technical report, Rational Rose, 2001.
- [SFL98] Neeraj Sangal, Edward J. Farrell, and Karl J. Lieberherr. Interaction graphs: A system for specifying and generating object interactions. Technical report, Tendril Software, Inc., 1998.
- [SRK02] Bran Selic, Guus Ramackers, and Cris Kobryn. Make models be assets. *Communications of the ACM*, 45(11):70–72, 2002.
- [Sta02] Leon Starr. *Executable UML: How To Build Class Models*. Prentice Hall, Upper Saddle River, first edition, 2002.
- [UML01] OMG Unified Modeling Language Specification: Version 1.4 September 2001. Technical report, Object Management Group, 2001.
- [UML03] OMG Unified Modeling Language Specification: March 2003 Version 1.5. Technical report, Object Management Group, 2003.
- [WKC⁺03] Ian Wilkie, Adrian King, Mike Clarke, Chas Weaver, Chris Raistrick, and Paul Francis. UML ASL reference guide: ASL language level 2.5, manual revision D. Technical report, Kennedy Carter Limited, 2003.
- [⇒Normalization] Database normalization basics. URL <http://databases.about.com/library/weekly/aa080501a.htm>.
- [⇒OOAD-Roadmap] A road map for OOA and OOD. URL <http://www.gvu.gatech.edu/edtech/B00ST/designmap.html>.
- [⇒Proposals] The proposals for UML 2.0. URL <http://www.community-ml.org/UML2.htm>.
- [⇒UML-Tools] A list of tools for the software development with UML, maintained by the OMG. URL <http://www.omg.org/technology/uml/index.htm\#Links-Tools>.

- [⇒xUML02] Supporting model driven architecture with executable uml, 2002. URL http://www.kc.com/cgi-bin/download.cgi?action=ctn/CTN_80v2_2.pdf.
- [CCG02] User guide: Together controlcenter, 2002. Updated September 10.