Master thesis in Computing science, 20 credits

A Sequence Diagram Editor for BlueJ

Matilda Östling, <c99mog@cs.umu.se>

Department of Computing Science, Umeå University

10th May 2004

Abstract

Today most programming courses for beginners use an object-oriented language. This has led to many new learning tools, theories and methods for teaching object-orientation. One such new tool is BlueJ which is developed to teach Java and object-orientation for beginners. BlueJ does not have any support for drawing sequence diagrams and this thesis describes the development of an editor for sequence diagrams. The editor is developed as a plugin for BlueJ and designed to be used by beginners. This paper describes the design and implementation of the editor and it contains a user manual for the editor.

In this thesis it is also talked about UML diagrams in general and more specific about sequence diagrams. The use of sequence diagrams in education of object-oriented thinking is described and advantages are pointed out. Object-oriented education is discussed and different teaching methods are addressed.

Contents

List of Figures

1

Introduction

1.1 Background

Today object-orientation is widely used both in education and software development. Even though object-orientation has been taught for quite a long time now, there exist many theories how to best teach object-oriented programming and thinking to students. The widely usage of object-orientation in education have raised a lot of questions. Is it more natural for a student to think in an object-oriented way then in a procedural way [Neubauer and Strong, 2002]? Is it harder to teach an object-oriented programming language then a procedural programming language? What is the correct way to teach object-orientation and which method should be used? There exist many opinions in this matter and some "guidelines", when teaching object-orientation with Java, have been developed [Kölling and Rosenberg, 2001]. One often used technique is "active learning" which can shortly be described as having the students more active in the learning process [Smialek, 2000]. Some commonly used techniques are so called CRC sessions, Role play sessions and Use case sessions. They let the students act like different objects in the system by playing scenarios where the system should achieve different tasks. Documenting these plays will help the students in designing their systems.

When teaching object-oriented programming languages to beginners, such as for example Java, new tools are used. One example is BlueJ that is an IDE (Integrated Development Environment) for Java designed to be used by beginners, see `www.bluej.org`. It is designed to help students understand the foundations of object-oriented programming and is not meant to be used by experienced programmers. When teaching object-orientation today sequence diagrams are often used. They have shown to be very beneficial for beginners to object-orientation and are good since they can for example be helpful in the design phase, clarify the program flow of the system and be used in the test phase [Fowler and Scott, 1997, Kutar et al., 2002].

1.2 The Purpose of this Master's Thesis

BlueJ does not have any support for drawing sequence diagrams and this paper will describe the development of an editor for sequence diagrams designed as a plugin for BlueJ. The editor will be connected to projects [1] in BlueJ and help the students draw sequence diagrams describing their implemented classes and methods. The editor is designed just as BlueJ to be used by beginners and it does not provide all possible notations for a sequence diagram but only the most basic parts. The editor has been developed with the same philosophy as BlueJ, to "keep it simple" and make it easy to use and understand.

1.3 Thesis Outline

In the first chapter the UML and its different diagrams are shortly described. Later in this chapter use cases, scenarios and sequence diagrams are described in more detail and the advantages of using sequence diagrams in education are distinguished.

In chapter three teaching and learning software development is discussed in general and in the next chapter, chapter four, teaching and learning object-oriented programming is discussed. Different methods are illustrated and active learning is described in more detail. In chapter five the use of sequence diagrams in education is talked about and advantages and disadvantages are pointed out.

Next chapter, chapter six, describes the design and development of the editor for sequence diagrams. The editor is described and the design choices are explained. The system of the editor is also shortly explained and shown. After this a chapter follows containing a short summary and conclusions of this work.

At the end of this thesis an appendix is attached containing an user manual for the editor. The user manual helps the user to install and use the editor.

[1] In BlueJ implemented classes belonging to the same system are saved together as projects.

2

UML

The development of the Unified Modeling Language (UML) began in October of 1994 and is an evolution from Booch method by Grady Booch, OMT (Object Modeling Technique) by James Rumbaugh, OOSE (Object-Oriented Software Engineering) by Ivar Jacobson and other object-oriented methods [Mrozek et al., 2002]. In October of 1995 UML 0.8 (then called the Unified Method) came. UML 1.1 was released in September of 1997 and the development is still in progress. UML 2.0 was released in 2003 and newer versions are expected soon [Björkander, 2003].

UML is a visual modeling language consisting of nine diagrams. A model of a software system can be explained as an abstract representation of the system. These nine diagrams in UML are used to visualize, specify, construct, and document the different components of a software system before the implementation. UML is used to make "blueprints" of a software system, this is necessary within big projects to get an overview of the system. UML has a tight mapping to object-oriented languages and is best suited for designing systems of this kind but UML can also be used advantageous with other programming languages [Rumbaugh et al., 2001]. Though UML is intentionally process independent the authors of UML advocate that UML is used in use-case-driven, architecture centric, iterative and incremental development processes [Rumbaugh et al., 1997b]. Two examples of such processes that are used for development in software engineering are the Unified Process and RUP (Rational Unified Process) [Phillips, 1998].

Static view	**Dynamic view**
Class diagrams	State-chart diagrams
Object diagrams	Activity diagrams
Use case diagrams	Sequence diagrams
Component diagrams	Collaboration diagrams
Deployment diagrams	

Table 2.1. The nine diagrams in UML version 1.5 belonging to the static view and the dynamic view, respectively.

2.1 Diagrams in UML

The UML is composed of nine different diagrams. Which diagram to use depends upon each special situation and the types of problems being solved. Usually the different diagrams are divided into two groups, static and dynamic models, see table 2.1. The static models, also called the structural models, underline the structures of different objects in a system, including their classes, interfaces, attributes and relations between them. The diagrams belonging to the static view are class diagrams, object diagrams, use-case diagrams, and implementation diagrams (component and deployment diagrams). The dynamic models, also called the behavioural models, emphasize the behaviour of different objects in a system, including their methods, interactions, collaborations, and different states. The diagrams included in this view are behaviour diagrams (state-chart and activity diagrams) and interaction diagrams (sequence and collaboration diagrams).

The following sections will briefly describe the different diagrams in UML and in what situations to use each diagram.

2.1.1 Use Case Diagrams

Use case diagrams describe *what* a system does and are closely connected to scenarios. Scenarios will be described in section 2.3. They can be thought of as a summary of scenarios for a single task or goal. Use case diagrams are best when developers are communicating with clients since they are relatively easy to understand and relate to. Use case diagrams are also useful when designing the system, when the requirements are made, and for generating different test cases for use in the testing phase [Booch et al., 1999].

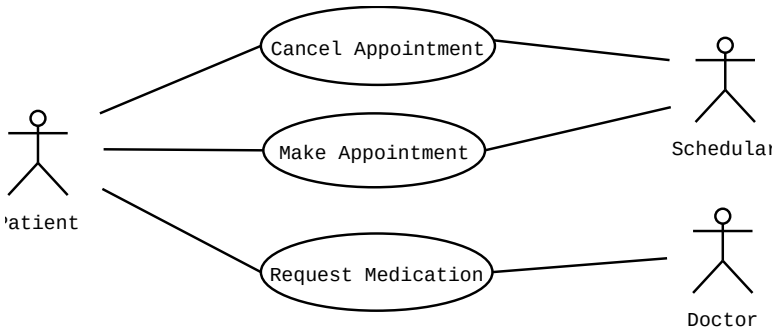

Figure 2.1. An example of an use case diagram.

2.1.2 Class Diagrams

To get an overview of a system, its classes and the relationships between them, a class diagram is used. The relationship among the classes can be several different kinds. The three most used relationships are association, generalization, and aggregation. An association between two classes is a labelled relationship. A generalization is a hierarchical relationship between two classes, meaning that one class is the parent of the other class, as in an inheritance hierarchy. An aggregation between two classes (also called a dependency) is a relationship meaning that one class depends on another in some way that if changing one class may lead to have to change the other class too. Class diagrams are static in the way that they describe what parts interact and not what happens when they interact with each other [Booch et al., 1999, LeBlanc and Stiller, 2000].

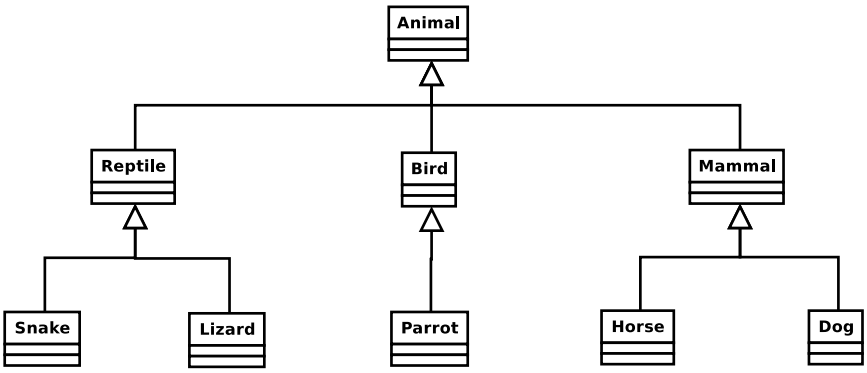

Figure 2.2. An example of a class diagram.

2.1.3 Object Diagrams

The object diagram is very similar to the class diagram except that they show instances (objects) instead of classes. The object diagram models a set of objects and their relationship during a system snapshot. An object diagram is very useful when, for example, explaining complicated relationships within a smaller part of a system [LeBlanc and Stiller, 2000].

2.1.4 Interaction Diagrams

Interaction diagrams describe dynamic aspects of a system and there are two kind of interaction diagrams, sequence diagrams and collaboration diagrams.

A sequence diagram explains how operations are performed, describing different scenarios. They show how different objects work together and what messages (method calls) are sent between them. A sequence diagram shows time-ordering between messages and the messages are organized chronological. A sequence diagram is shown in figure 2.3.

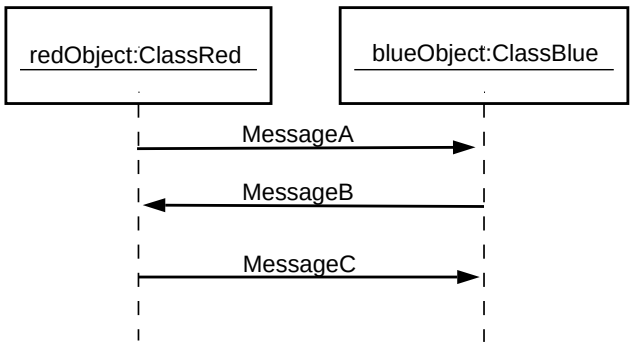

Figure 2.3. An example of a sequence diagram.

Collaboration diagrams contain the same information as sequence diagrams but they concentrate on the role of the objects, the relationship between the objects, and the communication between them. Instead of having an axis showing the ordering in time between the messages each message has a sequence number. A collaboration diagram is shown in figure 2.4.

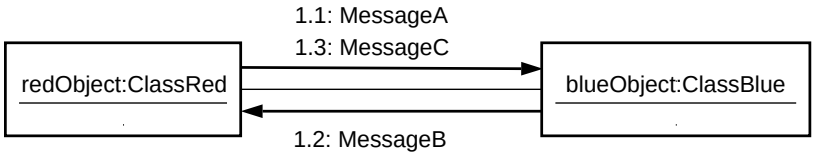

Figure 2.4. Example of a collaboration diagram.

Both collaboration and sequence diagrams are best to use when different scenarios are being described. The collaboration diagram emphasizes “who-is-talking-to-whom” and the time-ordering of the messages gets a little clouded. The sequence diagram on the contrary accentuates the time-ordering of the messages though here the-who-is-talking-to-whom situation gets obscured [Software, 2001]. Sequence diagrams and scenarios will be described in section 2.3 and in section 2.4.

2.1.5 Behaviour Diagrams

Objects can have different behaviour and states depending on their current actions or conditions. There are two diagrams to describe the behaviour and states of an object, statechart diagrams and activity diagrams.

A statechart diagram displays the possible states for an object and the transitions that will change the state of the object. The activity diagram instead focuses on the activities in a special process and the flow of activities. It shows how the different activities depend on each other. Even though the two different types of behavioural diagrams are closely related an activity diagram can be described as a “flowchart”, showing the flow of activities in a process, the statechart diagram concentrates on the object and its different states in a process and what causes the change in state for the object.

2.1.6 Implementation Diagrams

There are two kinds of implementation diagrams in UML, the component diagram and the deployment diagram. These diagrams display the view of implementation and the run-time implementation structure [Booch et al., 1999]. The component diagram describes the organization of physical software components, including source code, run-time code, and executables. The deployment diagram depicts the physical resources in a system, including nodes, components, and connections.

2.2 UML Diagrams in Education

The UML with its nine different diagrams is extremely large and offers many notational possibilities. When students new to object-orientation and programming are trying to learn and use UML they can easily be confused and find it difficult to use UML. It easily happens that they miss the big picture which is very important for understanding the goals and processes of software engineering. Though UML is very complex with a lot of rules for syntax and semantics the most important core of the language is quite easy to learn and understand which is pointed out by LeBlanc and Stiller [LeBlanc and Stiller, 2000].

When students start to use UML they only need to use a subset of the nine diagrams in UML or otherwise the students would be overwhelmed with all the possible notations. The subset has to offer the student enough of notational possibilities so he or she can model a software system without getting confused. Which subset of UML to use in education is under discussion and there are many proposals of suitable subsets of the UML. The class diagram, the sequence diagram and/or the collaboration diagram appear to be the most used diagrams in education.

These diagrams would be enough for beginners in programming and make it possible for the students to document the software systems without finding it confusing and too hard. Others wish to include more diagrams in the subset of UML for beginners. For example LeBlanc and Stiller [LeBlanc and Stiller, 2000] suggest a subset containing the use-case diagram, the class diagram, the object diagram, the collaboration diagram and the sequence diagram. The activity diagram and the statechart diagram have not been mentioned but they are also used when tutoring beginners. Since object diagrams are quite similar to class diagrams a common opinion is that they are not very important for beginners to UML. Sequence diagrams and collaboration diagrams contain and display almost the same information so it may be a good idea to use only one of these diagrams. According to different empirical studies sequence diagrams are often shown to be easier to learn and read than collaboration diagrams. Kutar, Britton and Barker [Kutar et al., 2002] have performed a study that showes that sequence diagrams are better than collaboration diagrams. The study performed both an empirical and a cognitive study of the two diagrams. The result of the cognitive study was that sequence diagrams are easier to read and understand then collaboration diagrams. The empirical study did not support this theory as much as expected, but it did not reject it neither. The conclusion of the study was that sequence diagrams are often better than collaboration diagrams though this must be better established with further studies. Even though this study shows that sequence diagrams are to prefer there are certainly a number of occasions when a collaboration diagram is better then a sequence diagram.

2.2.1 Educational Benefits for Using a Subset of UML

A class diagram is useful in the way that it models the composition of classes in a system. Showing the different relationships between the classes and the elements being part of each class. A class diagram gives a static view of the system, though a system can have many class diagrams showing different structural aspects of the system. For students having difficulties understanding inheritance (which is a basic concept in object-orientation) between classes a class diagram can help. It also shows how the classes depend upon each other and how the system is composed.

An object diagram reminds very much of a class diagram and has the same advantages. The big difference is that instead of showing how different classes depend upon each other an object diagram shows instances of classes, objects. An object diagram contains values of different attributes making it possible for students to understand and see the relationships between objects at a given time.

Use case diagrams are used to show how the system interacts with a user and are commonly used in the design phase of a system. A use case diagram is useful for students when designing the new system/program and modeling the behaviour

of the system towards a user. This kind of diagram is good since it hides much of the functionality of the system and only describes what the system should do. The students can then concentrate of the behaviour of the system instead of thinking about how the system should achieve certain tasks.

The interaction diagrams, collaboration diagrams and sequence diagrams are quite similar and display almost the same information. They show the interaction between objects in a system and describe the communication between them. These diagrams have shown to be very useful for students and quite easy and intuitive to learn and understand. Both collaboration and sequence diagrams can really help a student to see the program flow of the system and help them understand how the objects work together. Often it can be very hard in an object-oriented system to find the flow of program, especially for beginners in the area, since there is seldom an obvious program flow in an object-oriented system as it is for example in a system written in a procedural language [Fowler and Scott, 1997].

Sequence diagrams also have a very prominent role in teaching object-orientation today together with active learning, see section 4.3. Active learning can shortly be described as including the students more in the teaching/learning process and making them more active in their search for knowledge. Many investigations also show that sequence diagrams are found to be very useful both when designing a new system and also when trying to understand an already existing system. Sequence diagrams have shown to be appreciated both by experienced programmers and by novices. Though collaboration and sequence diagrams almost contain the same information, sequence diagrams more clearly show the program flow and helps the student understand the system. A sequence diagram is also often quite structured and it is easy to read and follow the program flow. A collaboration diagram can easily become a little indistinct if it gets too big and complex including many objects and messages. This makes the messages cross each other and makes it hard to arrange the objects in a structured way, leading to a diagram being difficult to interpret. Sequence diagrams and their use in education together with active learning will be further described in chapter 5.

2.3 Use Cases and their Scenarios

Use cases are used to describe sequence of events and to show how the system is supposed to interact with an actor (a user of the system). A use case is often just a plain text documentation that can be obtained from, for example, the requirements document. A use case is created for every major functionality of the system. Use cases are very useful in the design-phase since they describe *what* a system does without having to say *how* the system should achieve a certain task and how to

implement it [LeBlanc and Stiller, 2000].

Scenarios can be used in many situations, for example in the analysis of the requirements, in the software design, or in the implementation. The first two mentioned are probably the most important and mostly used areas. A scenario is used to describe a system's behaviour in a specific situation and can also be described as an instance of a use case. It is created from a use case by looking at every possible outcome and creating a scenario for each possibility. It is important to not only create scenarios for the normal system but also to show what will happen if an error occurs or if the system breaks down. Scenarios can be documented just in plain text or sometimes using different states or logic. From the different scenarios resulting from the use cases, sequence diagrams can be drawn. These sequence diagrams are often very useful in e.g. the design-phase of the software development and the documentation of it.

2.4 Sequence Diagrams

A sequence diagram shows the interaction between objects in time. The diagram displays the communication, i.e. the objects participating in the interaction and the actual messages sent between them. The sequence diagram focuses on the time-ordering between the messages, compared to the collaboration diagram that focuses on communication between the objects and the relationship between them. The relationship between objects is not shown at all in a sequence diagram. Sequence diagrams are used throughout the design phase in the development process to show the different scenarios in the system. They are suitable in real-time specifications and for complex scenarios. Sequence diagrams are used to model use case scenarios, protocols in a framework, subsystems, classes, and method logic [Miller, 2001b].

Or as Fowler [Fowler and Scott, 1997] explain the role of sequence diagrams:

> *"One of the hardest things to understand in an object-oriented program is the overall flow of control. A good design has lots of small methods in different classes, and at times it can be tricky to figure out the overall sequence of behaviour. You can end up looking at the code trying to find the program. This is particular true for those new to objects. Sequence diagrams help you to see that sequence."*

The sequence diagram is very attractive since it permits a lot of useful information to be shown at the same time. One thing is that the objects interacting with each other (in the described scenarios) can be ordered in such a way that the interaction is easier to understand and follow. Another good thing with sequence diagrams is that the lifeline of the object can point out the activity of the

object in isolation. Last and maybe most important is the way that sequence diagrams show the sequence of execution and the distribution of execution between objects [Fowler and Scott, 1997].

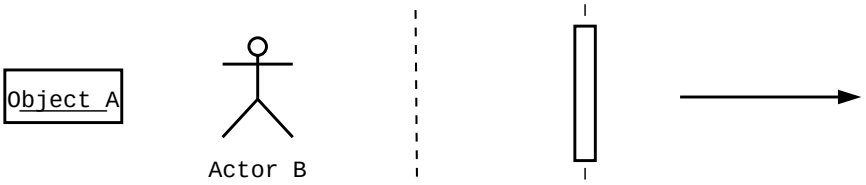

Figure 2.5. The different notations in a sequence diagram. First the notation for an object is shown, then an actor. In the middle is a lifeline for an object and then an activation for it. Last is the notation for a message sent between the objects.

A sequence diagram has two dimensions, time which is represented by the vertical dimension and different objects and actors that are represented by the horizontal dimension. In most cases time proceeds down the page though if desired the dimension may be reversed. The ordering among objects in the horizontal dimension is of no significance and arbitrary. But often the objects are ordered in such a way that the call arrows (messages) are arranged to point in the same direction (to the right) but in some cases this is not possible. Each object has a vertical dashed line which is called the lifeline of the object. The existence of the object is represented by the lifeline at a particular time [Rumbaugh et al., 1997a]. The lifeline can show whether the object is being created or destroyed during the shown scenario and when and for how long the object is active. The activity of the object is shown as a tall thin rectangle which starts at the initiation of the activation and ends when the object is not active anymore. If the object is being destroyed this is marked with a big “X” and the message destroying the object points at the “X” and the lifeline is ended at this point. If the object is created this is shown by letting the message creating the object point to the object symbol. This is shown in figure 2.6. An object that exists throughout the whole scenario has its lifeline from the top to the bottom of the diagram.

A sequence diagram can also display conditions, branches and loops, see figure 2.7. A condition is shown by putting the condition (e.g. an if-statement) enclosed in square brackets to the left of the message name. To symbolize a loop an asterisk (*) is used before the name of the message. If the number of iterations are known this number can be put in square brackets after the asterisk, though this is very seldom the case [Miller, 2001a]. To present a branch several arrows can start at a single point each with a guard condition. If the object sends a message to itself the call arrow just points back to this object, making an U-turn, see figure 2.6.

To express conditions, branches and loops in a sequence diagram (or another

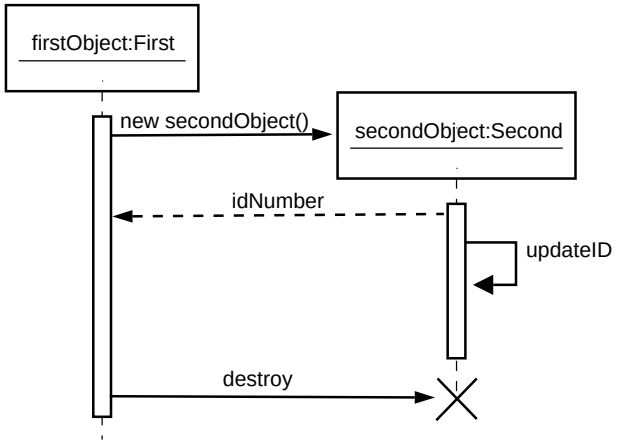

Figure 2.6. The notation for creation of an object and the notation for destroying an object. The secondObject is first created be the firstObject and then after sending a message back to the firstObject the secondObject is destroyed by the firstObject.

UML diagram) the Object Constraint Language (OCL) is used. OCL is part of the UML and it is a formal language used to express constraints. It is not a programming language but a specification language that uses simple logic for expressing different constraints [Rumbaugh et al., 2001].
OCL can be used for different intentions [Rumbaugh et al., 2001]:

- To specify invariants on classes and types in the class model.
- To specify type invariant for Stereotypes.
- To describe pre- and post conditions on operations and methods.
- As a navigation language.
- To describe guards.
- To specify constraints on operations.

OCL is often used to show constraints for a message in a sequence diagram. The given constraint says that it should only be executed if a special condition is fulfilled and OCL is used to express this condition. OCL is a typed language so each expression must have a type. A String can not be compared to an Integer in OCL and there is a set of predefined types in OCL that can be used when writing expressions using OCL. There are also predefined operators, keywords and the like in OCL [Rumbaugh et al., 2001].

In later versions of UML sequence diagrams have come to have a more prominent meaning. In the specification for UML 2.0 that was recently released (summer

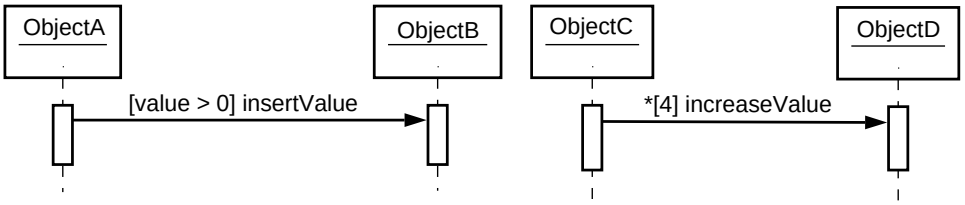

Figure 2.7. Two different sequence diagrams. The one to the left illustrates a message with a condition (value $>$ 0) and the other one illustrates a loop. The asterisk tells that there are a loop and the 4 tells that the number of iterations are four.

of 2003) [Group, 2004] one new thing was the increased potential for sequence diagrams. Sequence diagrams based on scenarios are powerful in many aspects but are maybe most associated with the specification and analysis phase. Sequence diagrams are also very powerful and useful when describing different test cases during the testing phase. One of the simplest but also the most powerful change is the introduction of references to other sequences. This gives the opportunity to break down a sequence into smaller sequences and then have references to it from other connections. All this leads to increased usability of sequence diagrams within UML [Björkander, 2003]. The increased interest for sequence diagrams in the context of UML is also reflected in education when UML is adopted.

3

Teaching and Learning Software Development

When teaching software development there are some concepts that are essential and it is important for the teacher to explain and clarify these concepts for the student. Most people think that software development is just about sitting in front of the computer and writing code. They do not understand the concepts design, analysis, problem solving etc. It is very important that a student new to programming learns and understands these concepts and realizes the meaning of them. Many students find it less interesting with design and does not fully understand the importance of design. They just want to take the problem and write the code for it without making any design or thinking about the best way to solve the problem. In the beginning this can work but when the problems get more complicated and the systems to implement get more complex this method will fail. So it is very important that the teacher lets the students realize the importance of design and have them see the positive side of it. Many authors have agreed that the most important thing when teaching/learning programming are the issues of problem solving, design and expressing a solution or design as a program [Robins et al., 2003].

Learning to program is not easy and it involves getting complex new knowledge and practical skills. Du Boulay [du Boulay, 1989] describes five sources of difficulty that must be overcome. These five domains are overlapping and sometimes all these new things can come as a "shock" for the student.
These domains are:

General orientation - What programs are for and what can be done with them. Understanding that programs are usually written for a purpose, with respect to some task, problem or specification.

The notional machine - A model of the computer as it relates to executing programs. Getting an overview of the computer, how it works and "how" the programs implemented are executed.

Notation - The syntax and semantics of a particular programming language. Learning to understand and use the new programming language. Learn how to write different statements and how to combine them into a program solving a given task.

Structures - How we learn new things according to special schemas and/or plans. Learning about these things can simplify the learning and understanding of e.g. a new programming language.

Pragmatics - Meaning to develop and exercise the skills of planning, developing, testing, debugging and so on.

3.1 Programming and Problem Solving

One common misunderstanding with programming and software developing is the importance of good problem solving. A good programmer is not just an expert at writing code but is also (often) a good problem solver which results in well written and effective implementations for different problems. A given problem can of course be solved and implemented in a number of ways and here the problem solving phase is crucial. Solving different problems and finding solutions are something that can be exercised. The hard part is often to see the whole picture of the problem, understand it and find a good solution to the problem. One good start is to split the problem into smaller parts. So instead of having one *big* problem there are many *smaller* problems. Then these small problems can be solved and by combining their solutions the big problem can be solved. This division of a problem into smaller parts is something that must be practiced. By doing this repeatedly a person can learn good ways to divide a problem and be able to see common patterns and often old solutions can be reused. Today many courses in programming for beginners use problem solving based teaching, also called problem based learning. Though there are many people positive to this approach there are also people meaning that the biggest difficulty for a novice programmer is to express solved problems using a programming language [Robins et al., 2003]. These people mean that problem solving is of course important, but the student also has to learn the programming language, how to use it, and how to write programs for the solved problems.

3.2 Coding and Design/Analysis

When the problem has been analysed and solved the design phase of the program/system can start. Before starting with the implementation the system has to be designed. When students new to programming learn and hear about design they often find it unnecessary and do not realize the importance of design. Because the programs they write in the beginning are so small it can be hard for them to see how significant the design is. But it is really important that it is pointed out for them why design is so crucial. Trying to explain how difficult (impossible) it would be to create a bigger system without any design is very important and also to show them examples of bigger systems. Since most programming courses for beginners today use an object-oriented language the significance of design grows since design is quite fundamental in object-orientation. To express the design UML-diagrams are often used. Since the UML is composed of nine different diagrams it is recommended that only a subset of the diagrams is used so the student does not find it too overwhelming. Design is just as problem solving something that must be practiced and exercised. There are many different methods to teach design and many are connected to object-oriented design. This will be further discussed later in section 4.3.

3.3 Syntax and Semantics

When learning a new (programming) language the student first has to learn what everything means and how it is written. This is when syntax and semantics are the crucial concepts. Shortly the syntax of a language can be described as how the different expressions, statements and program units are written. The semantics of the language can be described as the meaning of those expressions, statements and program units. Syntax and semantics are closely related and it is often found easier to understand/explain syntax then semantics. As Robert W. Sebesta [Sebesta, 1999] says this is partly because a concise and universally accepted notation is available for syntax description but yet there is none developed for semantics.

To clarify the difference between semantics and syntax lets take an if-statement written in the syntax of the programming language C.

$$\textbf{if}\ (< expr >) < statement >$$

How we write this statement form is controlled by the syntax and this is the correct syntax for an if-statement written in C. The semantics of this statement form is that if the expression (the current value of it) is true then the statement will be executed.

Many different studies have shown that novice programmers who know the

syntax and semantics of individual statements in a programming language often find it very hard to combine these statements into a program. They have solved the problem (by hand) but they can not write an equivalent computer program for the solution [Robins et al., 2003]. So learning the syntax and sematics of a programming language does not mean that the student will be able to write programs using the given language. First he or she has to learn how to “translate” the solved problem into code by combining different statements.

4

Teaching and Learning Object-Oriented Programming

4.1 Object-Oriented Programming vs. Procedural Programming

Today most programming courses for beginners use an object-oriented language instead of a procedural language. One of the main reason for the change in programming language is that the proponents of object-orientation are claiming that object-oriented thinking is more natural than procedural thinking. Neubauer and Strong [Neubauer and Strong, 2002] write about the theory that object-orientation is more natural and base this theory on the idea that the world we live in and experience is filled with things, or objects, which have both attributes and behaviours. Though this theory would lead to beginners preferring object-oriented thinking some people point out that beginners prefer the procedural way. Neubauer and Strong write that the explanation for this could be that students from the first day in school learn mathematics in a procedural way looking at procedural processes applied to data. Object-orientation has a drawback as many students see it, it includes a lot of design and abstraction. Most students are not really interested in design and think of programming as equal to writing code. Another disadvantage with object-orientation is that many find it more difficult to debug and correct an object-oriented system than a system written in a procedural language [Neubauer and Strong, 2002].

Though there are many pros and cons about object-orientation almost all universities today use an object-oriented language in the first programming course and the ones not doing so are planning to. This change in programming language has led to discussions about how to best teach object-oriented programming and which pedagogy is the best. Some mean that since object-orientation is quite new in education there are (yet) no good software tools and teaching support materials. This has led to teachers finding it more difficult to teach object-oriented thinking

than the procedural approach [Kölling and Rosenberg, 2001, Lewis, 2000]. Considering the opinion that it is harder for students to think in an object-oriented way, rather than in a procedural way, the strategies for teaching object-orientation really has to help the students to think in an object-oriented way. The students have to learn to think and program with objects [Neubauer and Strong, 2002]. Since all this is rather new there are several different theories and strategies about how to best teach object-oriented thinking and it is hard to say which strategy is the best.

Today most programming courses for beginners use Java as the programming language. Next section 4.2 describes different teaching approaches for object-orientation with Java as the programming language.

4.2 Different Teaching Approaches Using Java

There are some guidelines that are often mentioned in many articles about teaching object-oriented programming and thinking to beginners with Java as the programming language. The most common approaches are *"Objects first"* and *"Objects early"* which point out the importance of that students first of all get the understanding of objects when learning object-oriented programming and thinking. Though there is no scientific evidence, or very little, to support this theory most teachers and textbooks today are following this approach and start with objects early [Kölling and Rosenberg, 2001]. Ralph Westfall [Westfall, 2001] on the other hand says that this is not the case and one of his explanations is that many teachers have a background from programming with procedural languages. Another argument in the discussion is the use of "Hello world"-programs for beginners. Many claim that this is a very bad example to start with and that it is not object-oriented at all [Kölling and Rosenberg, 2001] when others mean that if it is correctly used and explained it is a good example [Lewis, 2000]. Ralph Westfall [Westfall, 2001] says that the "Hello World"-program must be rewritten to be object-oriented and gives an example of this. He says that in most books for teaching Java the following code is presented:

```
class HelloWorld {
       public static void main(String[] args) {
              System.out.println(''hello, world'');
       }
}
```

He means that this code will only confuse the students and not teach them anything about object-orientation. Ralph Westfall says that the code has to be rewritten to include a user-created object. The result would then be:

```
class HelloWorld {
       public static void printHello() {
              System.out.println(''hello, world'');
       }
}

class UseHello {
       public static void main(String[] args) {
              HelloWorld myHello = new HelloWorld();
              myHello.printHello();
       }
}
```

The main method in Java is also considered to be a problem by many teachers since its only purpose is to connect the application to the operating system. The code does not naturally relate with any classes or objects and it does not implement an operation on an object [Kölling and Rosenberg, 2001]. Another thing with Java is its support to deal with input and output which is huge and complex and can lead to difficulties for beginners. A proposal to solve this is that the teachers provide students with classes for I/O that encapsulate the complexities in Java. Another approach is that the standard I/O is just avoided until the students are able to understand and use it [Lewis, 2000]. There are more guidelines for teaching object-orientation that often come up. One is that the students in the first stage do not start with a blank screen but instead make changes to already existing code and see what the result of the changes are. It is also recommended that students read code to get a sense of well written programs and learn about good style and idioms [Kölling and Rosenberg, 2001]. It is really not easy for a student to write good code if he or she never got the opportunity to read well written and structured code.

4.3 Active Learning

To fulfil all, or at least some, of the guidelines mentioned in section 4.2 when teaching object-oriented thinking some newer approaches have been adopted. One widely used approach is *active learning*. Active learning means that the student instead of just receiving new knowledge from the teacher instead gets a problem and solve it, often with some guidance from the teacher. This type of education forces the learner to find and to use new knowledge to solve the given problem. This type of education is also called learner-centered education since it involves the students more in the learning process and make them more active. It seems that

people learn best when absorbed in the subject and actively participating in the process towards their own learning and understanding [Harley et al., 1998]. Though this technique with active learning seems to give the learners good understanding quickly it cannot be backed with any statistical data [Smialek, 2000]. It has also been shown that if students work in pairs or in groups better results are achieved. This depends on two things. The first thing is that students working in a group can solve more complex and interesting problems than a student working alone leading to more active and engaged students. The second thing is that students in a group have to discuss for example different designs and argue with each other and this is the kind of reflection that leads to learning [Harley et al., 1998].

One often used technique is pair-programming. This technique is used both for beginners to programming and for experienced programmers. The proponents of pair-programming mean that by always being two persons common and "unnecessary" errors can easily be avoided, the solutions often get better and the two programmers can switch places to avoid having a person doing the same thing for a long time [Jensen, 2003]. When beginners use pair-programming more complex and interesting problems can be solved and the student not typing can check for errors in the code which are often easier to discover when "sitting on the side" [Williams and Kessler, 2003]. It has also shown that when students working together the process of learning a new programming language is significantly faster then for a student working alone [Williams and Kessler, 2003].

According to these new approaches for education of object-oriented thinking there are some basic elements that are widely used. Two of them are *use cases* and *CRC cards*. Using these methods together with the philosophy of active learning the best result is achieved when teaching object-oriented thinking [Smialek, 2000].

Use case session - In a use case session the different use cases of the system are modelled and for each use case when it is possible a number of scenarios are developed. The participants of the session, which the students are, write down the scenarios using a very elementary grammar. Sometimes these use case sessions can be formed like interviews where somebody plays the role of the user, somebody the role of the analysts and the rest of the group review the session by documenting the scenario. This documentation is for example done with a sequence diagram or a simplification of one [Smialek, 2000].

CRC sessions - CRC cards characterise objects by Class name, Responsibilities, and Collaborators and are widely used for teaching novice programmers the concept of object-oriented thinking and design [Beck, 1989]. Often the process starts with identifying the classes in the future system and making a CRC-card for each class. Then the responsibilities and collaborators of the object are added on the cards. Now the students can use the CRC-cards to

illustrate different scenarios in the future system and document these scenarios. This is often done using some kind of role play where the students play different classes in the system. The responsibilities and collaborators of the different objects are surely to change during the session and can easily be adjusted during the session. CRC-cards are suitable for groups of students and the groups must not be too big since it will then be hard to engage all students actively in the session. Four to six students have shown to be a good size of the groups [Nordström and Börstler, 2002]. Since the role-play sessions will create different scenarios describing the behaviour of the system it is a good idea to have one or more students documenting the role-plays. Sequence diagrams are often used when documenting the different scenarios resulting from the CRC sessions.

The main purpose with active learning is to include the student more in the learning process and making him or her more active in the seeking for new knowledge and understanding. Taking away the “old picture” of teaching and learning when the teacher talked and the students listened and hopefully learned what they where supposed to learn.

Both techniques described above often use sequence diagrams, or a simplified version of them, for the documentation. In chapter 5 the use of sequence diagrams in education will be further discussed and described.

5

Sequence Diagrams in Education

Since object-oriented languages have become the leading programming languages in the beginner's programming courses, instead of procedural programming languages, UML has begun to play a bigger part in the education.

Relating to previous sections about teaching object-orientation and the guidelines mentioned together with the "new" approach with active learning, sequence diagrams become very important and useful. Since object-orientation is much about design and the understanding about objects and classes and the connection between them, UML is widely used. When beginners start with object-orientation one complicated thing is how to understand and see the communication between different objects, to understand how objects cooperate to accomplish a given task and how messages are sent between the objects. As Fowler and Scott point out [Fowler and Scott, 1997] it can be very difficult to understand the program flow in an object-oriented system, especially for beginners new to object-orientation. Here sequence diagrams can be very helpful for the students trying to understand and learn about object-orientation, objects, and the communication between objects in the system. The three different teaching approaches, described in section 4.3, all include the use of sequence diagrams in some way.

Since the major part of sequence diagrams describes the communication between objects and the time-ordering among messages, they can help a student both to understand an already designed system, or help the student to design and analyse his or her own new system. If the students are to design a new system and for example use the model with role-plays, sequence diagrams will play a meaningful part. Since the role-play produces different scenarios when the students play different classes cooperating to achieve certain goals the scenarios have to be documented in some way. Since the purpose of sequence diagrams are to describe different scenarios the natural thing would be for the students in the role-play to use a sequence diagram to document the scenario the role-play resulted in. If a sequence diagram is used for the documentation this diagram will help the students later in the design and the implementation of their system. One other guideline when teaching object-orientation is that students get a bigger system and try to un-

derstand it and later make changes to the system. When the student “gets a system” and the documentation belonging to the system, it will surely make it easier for the student to understand the overall program flow if a sequence diagram is used. Then the student can see how the different objects cooperate and how messages are sent between them. This will certainly facilitate the understanding of the system for the student and help him or her see the overall program flow. If there were no sequence diagram to help explain and show the program flow of the system the student had to read and try to understand the code to get the whole picture. If the student were a beginner in programming and maybe also in object-orientation this would for certain not be an easy task.

When students in a group together are supposed to design a system with its classes and methods the process is simplified if sequence diagrams are used. When several people (students) sit together discussing for example the design of a system it is very easy to misunderstand or misinterpret each other if only words are used. If the group wants to be sure that no misinterpretations are made and that everybody is talking the “same” language it is very good to draw a “picture” [Hussman, 2002]. When designing an object-oriented system this “picture” very natural becomes a sequence diagram since the discussions in the design phase certainly are concentrated around classes and the communication between them. If it is the students’ first programming course it is even more expected that there will be misunderstandings in the discussion since everything is new for the members of the group. Drawing pictures (e.g. sequence diagrams) will then increase the understanding within the group and probably result in a better design of the system since everyone in the group can participate in the design.

6

A Sequence Diagram Editor for BlueJ

The main task with this work was to design and implement an editor for sequence diagrams. The editor should work as a plugin for BlueJ. BlueJ is an IDE (Integrated Development Environment) for Java and is created to be used by beginners. More information about BlueJ and available downloads can be found at `www.bluej.org`.

6.1 Requirements

There were no specific written requirements for this sequence diagram editor. The editor should be implemented in Java and designed as a plugin for BlueJ.

In BlueJ different classes are saved together as projects. It seemed quite natural that a sequence diagram should be connected to a project meaning that the sequence diagram describes the classes and methods belonging to the given project. Each project in BlueJ should also be able to contain an arbitrary number of sequence diagrams. When drawing a sequence diagram the user should be able to choose from the implemented classes and their methods. In this way the user can not invent own classes and methods that do not exist and are forced to only use existing, implemented classes and methods. Since BlueJ is mostly used by beginners this is a positive thing because the sequence diagrams are only going to reflect implemented classes and methods. This functionality also has its drawbacks which will be further discussed in section 6.4.

A sequence diagram can be very complex and contain a huge amount of information if all of the notational possibilities are used. Since BlueJ and the sequence diagram editor will be used by beginners the sequence diagrams drawn in the editor will be of the basic kind. The editor does not offer all notational possibilities but only provide the most basic concepts. Belonging to the basics of a sequence diagram are objects and actors, destroy symbols, messages, return messages, and

lifelines. Of course this is a big limitation, but when beginners draw sequence diagrams these notations are mostly (only) used. Since the editor must not be too complex and difficult to use this subset of all notational possibilities will probably be enough.

The sequence diagrams that are drawn should also be able to be saved and opened later on. There were no special requirements how the sequence diagrams should be saved and in what form. Though it seemed natural that sequence diagrams belonging to a certain project should be saved in the same directory as the project it belongs to.

6.2 Design

6.2.1 The Graphical User Interface

The first thing that I started with was the Graphical User Interface (GUI). The main thing was that it should be easy and intuitive to use and not differ from the GUI of BlueJ too much, see figures 6.1 and 6.2. Since they are beginners who will use the editor it should not include too much functionality and the program should not take a long time to understand and learn. The window of the editor contains a drawing area, where the sequence diagram is drawn, some menus in the top of the window and some icons that are used to draw the sequence diagram. The icons represent an actor, an object, a message and notes for the diagram. The GUI is implemented using Java's swing-library, see `http://java.sun.com`.

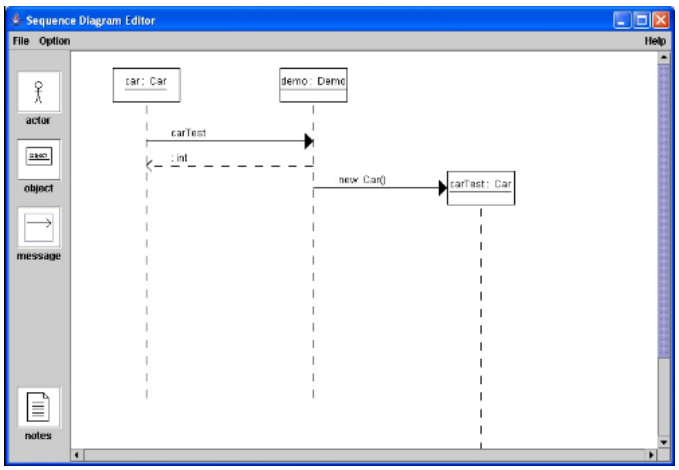

Figure 6.1. The window of the sequence diagram editor.

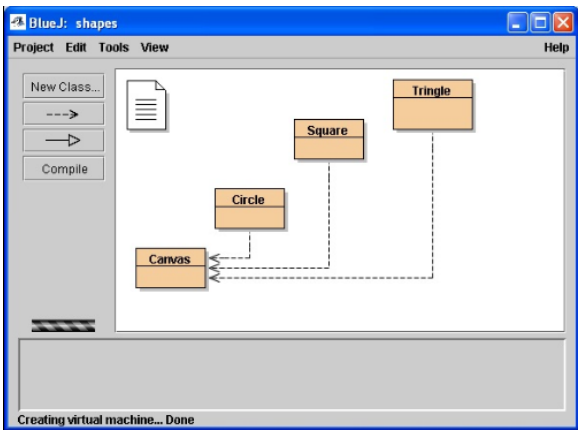

Figure 6.2. The window of BlueJ.

The menu The menu contains three different items, *file*, *option* and *help*. The menu item *help* has been placed to the right in the window. The reason for this is because BlueJ has the menu item *help* placed here. Then the user will quickly find the help if he or she is used to BlueJ. To the left in the window *file* and *option* are placed. In the menu item *option* the user can choose BlueJ-mode on or off and whether to show return messages or not. Under the menu item *file* is *open*, *save*, *save as*, *close* and *quit*. These menu items are also connected to the keyboard if the user for example wants to use `Ctrl-S` instead of choosing save from the menu. The same functionality is also included in BlueJ.

The icons At the left side in the window four icons are placed. These icons represents from top down an actor, an object, a message and notes for the sequence diagram. The three first mentioned icons are the most common parts of a sequence diagram and because of this it seemed natural to give them each an icon. In this way the user easily can create for example an object without having to use the menu. The icon for notes is placed in the left bottom a little bit away from the other icons because it has a different functionality. In BlueJ each project has a note belonging to it where the user can write information about the project. To be consistent to BlueJ the editor also has a note where the user can save information belonging to a certain sequence diagram.

The drawing area The drawing area where the sequence diagram is drawn is placed to the right in the window. The background is white and the dif-

ferent parts of the sequence diagram are drawn in black. Simple and easy to read. The drawing area has a default size from the start, but if the user draws a sequence diagram that is bigger than the drawing area scrollbars are added automatically. Then the user can scroll the drawing area if the diagram is too big.

When an actor or an object is added to the drawing area by the user it always gets a lifeline with a default length. If the user wants to make any changes to an already added actor or object he or she can click with the right mouse button on the object. Then a menu appears and the user can choose between changing the name, deleting it, and adding a destroy symbol. In BlueJ the name for an object can be changed if the user double clicks on the object. Because the editor should be similar to BlueJ this function also exists in the program. The name of an object, an actor, or a message can be changed by double clicking on it.

When a message is added to the sequence diagram the user first marks the starting object with the mouse by clicking on its lifeline. Then the message gets visible and "follows" the mouse (like a rubber band-effect) until the user has chosen the end object of the message. This is also done by clicking on the end object´s lifeline.

Most parts in the drawn sequence diagram can be moved after they have been added. When a part is marked (clicked on) it is drawn in red and there appears "handles" that the user can grab and move the selected part of the diagram. The user moves the selected part by pressing the left mouse button and draw the part with the mouse. I have chosen this method because many drawing programs use this method ("press and drag") when the user should move, for example, an object. So many users will find this method natural and easy to use.

6.2.2 Return Messages

When a user adds a message to a sequence diagram the message will automatically get a return message. An exception to this is if the message goes to and from the same object. Then the message will not get a return message. If the type of the return message is known it will be added to the return message, for example if the type of the return value is int, the return message will get the label ":int". The return message is placed under the message it belongs to with a distance around one cm. If the user wants to hide the return messages he or she can do this in the menu under option. Return messages can be switched on and off during the development of a sequence diagram.

6.2.3 Sequences

Beginners to programming often write sequential programs and because of this one functionality that should be implemented was the opportunity to have messages in a sequence in a sequence diagram. If messages are added to a sequence they are automatically arranged and the distance between them and their return messages is updated. This leads to a sequence diagram that is symmetric and easy to read. If the user on the other hand wants to draw a sequence diagram not having the messages in a sequence this is also possible.

To illustrate how this functionality has been implemented an example is used. If message A is to be added to an already existing sequence after message B, message A's start point is placed between message B and message B's return message. Then message A will be added to the sequence after message B and all positions for messages and return messages belonging to the sequence will be automatically arranged.

6.2.4 Different Modes of the Editor

Since one requirement was that the editor should be a plugin for BlueJ and that a sequence diagram should be connected to a certain project the editor is developed so it can be run in two different modes, BlueJ-mode on or BlueJ-mode off. If the editor is in BlueJ-mode the sequence diagram that is being drawn will be connected to the current open project in BlueJ. Then the user must choose classes and methods belonging to the project. If the editor is not in BlueJ-mode the user can draw a sequence diagram that contains objects, messages etc that are not yet implemented. If the program is started from BlueJ, as a plugin, the default is that the editor is in BlueJ-mode. The user can change the mode under option in the menu. The editor can also be started as an independent program and then it is used to draw arbitrary sequence diagrams.

6.2.5 Creating a Plugin for BlueJ

When creating a plugin for BlueJ I got a lot of help from the webpage for BlueJ, see `www.bluej.org`. They had a number of tutorials and examples. When the editor is in BlueJ-mode it has to get information of the current open project. For example what classes exist and what methods each class contains so the user can choose what class or method to add to the sequence diagram. For this BlueJ provide an interface that makes it quite easy to integrate the editor with the BlueJ program.

I started with adding the editor to the menu in BlueJ and when the editor starts it receives an object of the BlueJ program. Through this object the editor gets access to the current open project and its classes, methods etc.

6.2.6 File Management

There is a number of ways and methods to use when implementing the saving of a sequence diagram, for example as a plain text file, using XML etc. It seemed very complicated to save them as plain text or some kind of combination of numbers since there is quite a lot of information that must be saved and it is easy that something goes wrong. If you want to change something later on, like remove or add an attribute, that can be quite complicated. Since I have never used XML it seemed like a good idea to use and learn about XML. One positive thing about XML is that it is very easy to add or remove things during the development. Changes are easy to make which would have been very hard if the diagrams for example had been saved as plain text.

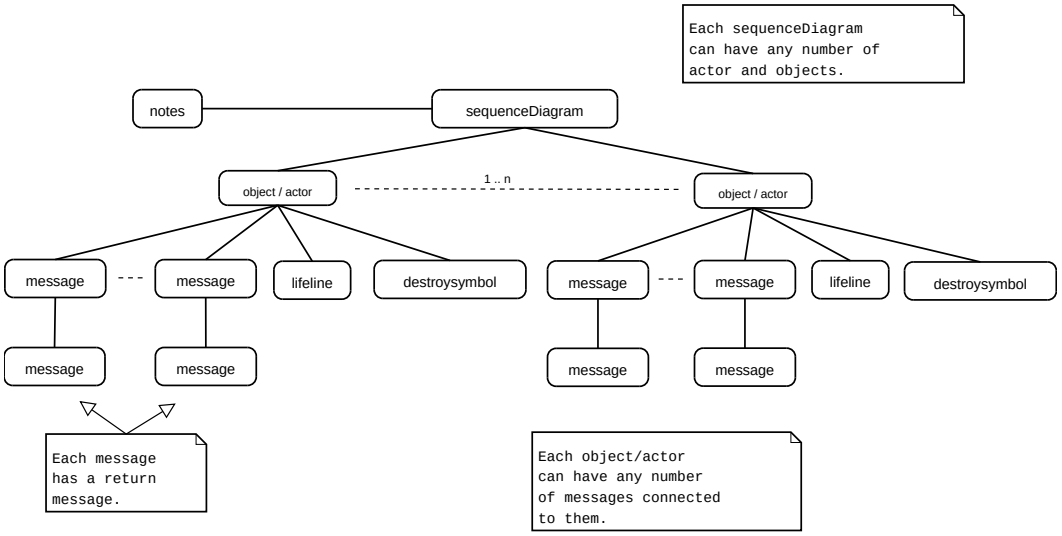

Figure 6.3. How the XML-file is arranged. All parts of a saved sequence diagram are arranged in a tree structure with the sequenceDiagram node as the root.

The XML-file is arranged with a tree structure, see figure 6.3. The node “sequenceDiagram” acts like the root. All actors and objects in a sequence diagram are saved as child nodes to the root. Each actor/object has child nodes containing information about the lifeline, destroy symbol and all messages belonging to the actor or object. Each message also has a child node containing the information about its return message. The sequence diagram contains a “text” representing the notes about this sequence diagram. This note is saved as a child node to the root. The number of actors and objects in a sequence diagram are arbitrary and also the number of messages belonging to an actor or an object.

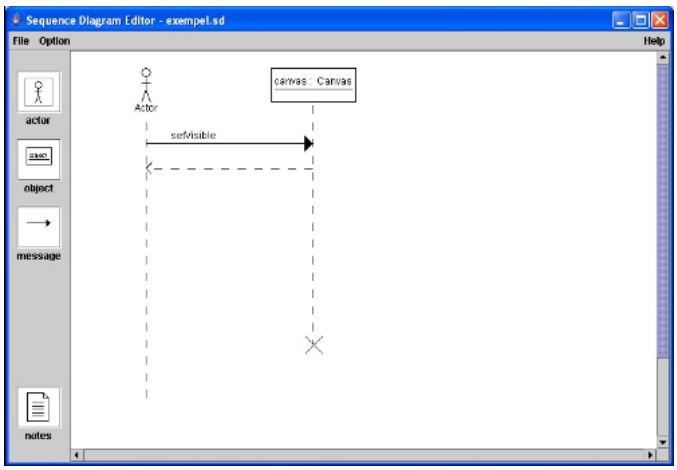

Figure 6.4. An example of a simple sequence diagram.

In figure 6.4 a simple sequence diagram is displayed. The XML textual syntax for this sequence diagram will be:

```
<?xml version="1.0" encoding="UTF-8"?>
<sequenceDiagram bluej="shapes">
<notes>Notes about this Sequence Diagram
This diagram is only an example...</notes>
<actor startPoint="85,20" endPoint="95,60" name="Actor" BClass="null"
       hasDestroySymbol="false" hasCreationMessage="false">
<message mess="setVisible" startPoint="90,105" endPoint="290,115"
       isCreateMessage="false" isReturnMessage="false" startEntity="Actor"
       endEntity="null" sequenceIndex="null" returnMessage="notNull">
<message mess="" startPoint="290,135" endPoint="90,145" isCreateMessage="false"
       isReturnMessage="true" startEntity="null" endEntity="Actor"/>
</message>
<lifeline startPoint="86,65" endPoint="94,420"/>
</actor>
<object startPoint="240,20" endPoint="340,60" name="canvas" length="101"
       BClass="BClass: Canvas" hasDestroySymbol="true" hasCreationMessage="false">
<destroysymbol startPoint="281,341" endPoint="301,361"/>
<message mess="setVisible" startPoint="90,105" endPoint="290,115"
       isCreateMessage="false" isReturnMessage="false" startEntity="null"
       endEntity="canvas" sequenceIndex="null" returnMessage="notNull">
<message mess="" startPoint="290,135" endPoint="90,145" isCreateMessage="false"
       isReturnMessage="true" startEntity="canvas" endEntity="null"/>
</message>
<lifeline startPoint="286,65" endPoint="294,351"/>
</object>
</sequenceDiagram>
```

Since the program can be in two different modes, see section 6.2.4, BlueJ-mode on or off, the saving procedure also differs a little. When the program is in BlueJ-mode the sequence diagram is always saved in the same directory as the current project. The user then only gets to choose the name of the file. If the program is not in BlueJ-mode the user gets to choose both the name of the file and which directory the file should be saved in. All sequence diagrams are saved with the extension *.sd*. If the user does not add this to the filename the extension is added by the program. When the user chooses to open a file the program only shows files with the extension *.sd*. If the editor is in BlueJ-mode the *.sd*-files belonging to the current project is shown and otherwise the *.sd*-files is shown in the current working directory.

Like most programs the editor always asks the user if he or she wants to save the current open sequence diagram if the sequence diagram is being closed for some reason. If the user switches BlueJ-mode with an open sequence diagram the editor closes the open sequence diagram before changing mode. Doing so the editor makes sure that a sequence diagram belonging to a given project is not changed without confirming the correctness with the project.

6.2.7 Automatic Consistency Check against BlueJ

When the editor is in BlueJ-mode a sequence diagram can only include classes and methods already implemented in the current open project. If the user draws a sequence diagram belonging to a certain project this sequence diagram has to be checked against this project when it is later opened again. If the editor discovers that a class included in the sequence diagram does not exist in the project anymore this object (of the missing class) is drawn in red in the diagram and the user gets a warning-message from the editor.

6.2.8 Creation Messages

Sometimes an object is created by another object during a displayed scenario in a sequence diagram and this functionality is provided by the editor. To create a new object, or make a creation message, the user first chooses to add a new message. He or she selects the start object, meaning the object going to create the new object. But instead of selecting the end object for the message the user clicks with the right mouse button and then gets to choose which object to create. The editor then creates the new object and places it at the end of the creation message.

6.3 System Description

If all classes and interfaces were to be described it would take up too much space so only the most important and interesting classes will be described. All documentation for the program can be found at:
`http://www.cs.umu.se/~c99mog/javadocs/`.

A class diagram including the most important classes can be seen in figure 6.5.

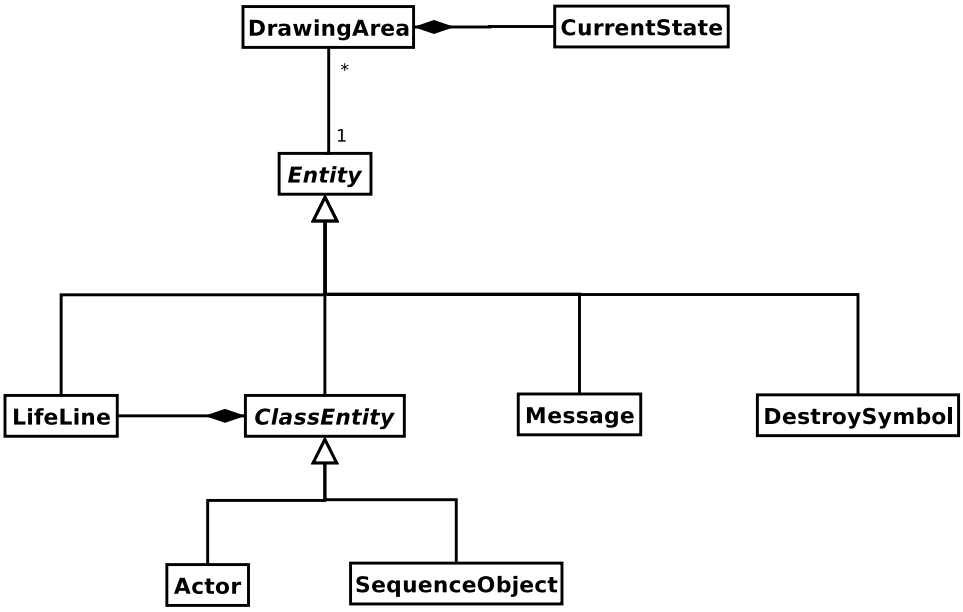

Figure 6.5. A class diagram describing the most important parts of the system.

EditorExtension This is the main class. If the program is started from BlueJ the editor is not started using main but another method called “startup”. To start the extension, BlueJ calls this method sending an object of itself as an parameter. This class later starts the editor by creating an object of the class SeqDiagramEditor.

SeqDiagramEditor This class starts the editor and creates menus, icons, the drawing area etc. It also handles all events that comes from the menus or if the user clicks on one of the icons.

DrawingArea This class contains most of the code and could be said to be the “brain” of the editor. DrawingArea.java implements the interfaces ActionListener, EventListener, MouseListener, and MouseMotionListener. DrawingArea

keeps track of a lot of information, e.g. what file is currently open, what parts to draw in a sequence diagram, and it also saves and opens a sequence diagram to and from a file.

CurrentState This class is used by the Drawingarea to keep track of what to do next. If a user selects e.g. a lifeline the current state is updated for the Drawingarea so it knows how to handle next event created by the user. For example if a new message is to be created and the start point of the new message has been selected the class Message sets the CurrentState for DrawingArea to `RUBBERBAND`. Then the DrawingArea knows how to behave and what action to take when the user makes his or her next choice. Every class has a kind of state machine that tells the DrawingArea what the next step will be, see figure 6.6. The class CurrentState defines a number of different states e.g. `NORMAL`, `INSERT`, `FIND_END_POINT`, and `CHANGE`. These states are used by the different classes to set the state correct for the DrawingArea.

Entity This class is an abstract class that all parts belonging to a sequence diagram inherit from.

Message This class inherits from Entity and represents a message in a sequence diagram. A message can go between two objects or actors but can also have the same object or actor as start and end point. This class has an attribute saying whether it is a regular message or a return message. Depending on the value of this attribute the message is drawn in different styles on the drawing area.

LifeLine This class inherits from the class Entity and represents a lifeline in a sequence diagram. A lifeline is always associated with either an actor or an object.

DestroySymbol This class represents a destroy symbol for an object and it also inherits from the class Entity. A destroy symbol is always connected to an actor or an object.

ClassEntity This is an abstract class that represents either an object or an actor in a sequence diagram. It inherits from Entity and implements a few methods common for an actor and an object.

Actor This class inherits from ClassEntity and represents an actor in a sequence diagram.

SequenceObject As the class Actor this class also inherits from ClassEntity and an object in a sequence diagram is represented by this class.

SDFilter This class is used when saving and opening files in the editor. The filter only make files with the extension *.ds* visible for the user.

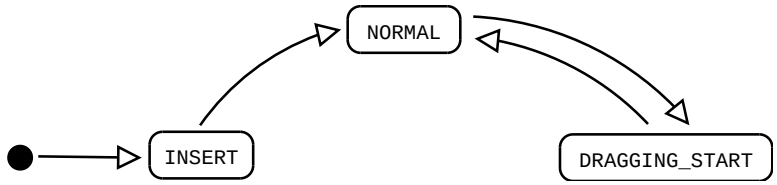

Figure 6.6. The state machine for an object of the class DestroySymbol. When the object has been created and inserted to the sequence diagram it can toggle between the two states `NORMAL` and `DRAGGING_START`. The state `INSERT` only appears when the destroy symbol is added to a sequence diagram. When the destroy symbol is added the state is updated to `NORMAL`. If the user marks the destroy symbol by clicking on it the state is updated to `DRAGGING_START` and the destroy symbol can be moved by the user. If the destroy symbol is "unmarked" by the user the state is updated back to `NORMAL`.

6.4 Future Work

There are a number of features that had not (yet) been implemented. The main thing missing is an undo-button. The current solution for the user is to use the delete function if something goes wrong or gets into the wrong position. Probably this can be quite frustrating for the user and an undo function must be seen to have high priority.

Another function that would be useful to have implemented is the possibility to print a sequence diagram. Of course the user can take a screen shot and print that, but it would be practical with a print functionality in the program.

There is also the possibility of code generation, meaning that if the user draws a sequence diagram the program should create code based on the drawn sequence diagram. For example, when the user is in BlueJ-mode and suddenly realizes that he or she would need a new class or a new method in a class. Instead of having to go back to BlueJ and create the class or method the user could just add the class or method to the sequence diagram and the editor would then generate the code for the method or class.

Since the editor is mainly supposed to be used by beginners there are a lot of notational possibilities of the UML that have not been implemented. If the editor is to be used by more experienced programmers the editor has to offer more notational possibilities. Activation boxes are not implemented in the editor since the conclusion was that this functionality was not very important for beginners. If the editor is to be further developed activation boxes should be implemented and also the possibility for each actor or object to have a number of activation boxes.

If the drawn sequence diagram gets too big scrollbars are added to the drawing area. A function that could be beneficial to have would be some kind of zoom so the user can zoom in and out. Then the user can zoom out if he or she would like to look at the whole diagram at once or zoom in to look at a smaller part of the diagram in more detail.

7

Related Work

There exist a number of editors for UML, e.g. Rational Rose, see `www.rational.com`, Together, see `www.borland.com/together/` etc. The big drawback with these editors is that they are too complicated for beginners to use since they are designed to be used by experienced programmers. If a beginner new to programming and UML should use any of these programs he or she would only get confused and intimidated probably leading to less interest in the area.

Today there are less complicated editors for UML too, for example MS Visio (developed by Microsoft) and Violet, see `www.horstmann.com/violet/`. MS Visio has a library for drawing UML diagrams but the program does not understand the diagrams. This leads to that the components can be arranged in any way and there is no check for correctness of the diagrams. Violet is an editor designed to be used by students and it is quite easy to learn and understand. Violet has the same drawback as MS Visio, it does not interpret the diagrams and the components can be combined in any way. If a student draws a sequence diagram it would be helpful if the editor could make a consistency check against a class diagram of the system. Another drawback, with the editors mentioned, is that they are not platform independent. MS Visio for example does only work together with Windows.

Today many courses in programming for beginners use Java as the object-oriented programming language and BlueJ as an editor for writing programs in. Currently BlueJ does not have any support for drawing sequence diagrams and there exists no editor that works integrated with BlueJ. If the student using BlueJ would like to draw a sequence diagram he or she has to use some editor not connected to BlueJ at all. Doing so there will be no consistency check against the implemented classes and methods in BlueJ and the correctness of the diagram will not be controlled. The editor developed in this thesis is (hopefully) easy to understand and use. It is very uncomplicated to install and since it is implemented in Java it is just as BlueJ platform independent. The editor is connected to BlueJ and the students using it can draw sequence diagrams describing their implemented classes and methods. The editor always checks against existing classes and methods resulting in a correct and consistent sequence diagram.

8

Summary and Conclusions

In this paper the development of an editor for sequence diagrams has been described. The editor is designed to be used by beginners to programming and implemented as a plugin for BlueJ. Since the editor is mainly to be used by beginners the design has focused on making the editor easy to use and learn instead of having it contain a lot of functionality which would make it more complicated to use, especially for beginners. The plugin is easy to install into BlueJ, just copy the jar-file into the correct directory of BlueJ, see section A.2 in the appendix. Then the editor is started from the menu of the BlueJ program.

The editor only provides notations for the most basic parts of a sequence diagram. What the basic parts of a sequence diagram are is only based on my own opinion and can of course be discussed. The editor has support for drawing actors and objects with lifelines. It provides two different types of messages, regular synchronous messages and return messages. The reason for only supporting synchronous messages is because the editor is designed for beginners and they often only use this kind of messages. If the editor would support all possible messages of the UML the user would probably find it complicated to know what message to use if there were too many possibilities. Since beginners new to object-oriented programming often write sequential programs the editor also provides the possibility to have the messages creating a sequence. Through this sequence of messages the program flow is easy to follow and understand. Return messages are also supported and the user can choose whether he or she would like to show or hide the return messages. Other notations that the editor has support for is destroy symbols and create messages. The possibilities of having activation boxes are missing in the editor and this could be seen as a drawback. The explanation for this is that it showed to be quite difficult to get them right and it did not seem too important to offer this functionality for beginners since they do not bring anything very important to a sequence diagram. Of course it could be implemented in later versions of the editor.

Since the purpose of the editor is to be used as a plugin to BlueJ it is designed to be run in two different modes, BlueJ-mode on or off. When it is run in BlueJ-mode the editor is connected to a project in BlueJ. Then the user can only draw sequence diagrams describing a given project, including its classes and methods. When drawing the sequence diagram the user gets to choose from existing classes and their methods. This will hopefully help the student draw correct sequence diagrams describing their systems. If the editor is run with BlueJ-mode off the user can draw sequence diagrams containing any classes and methods.

One thing missing in the editor is the possibility of code generation. If the user draws a diagram and suddenly realizes that he or she is missing a class or a method he or she has to go back to BlueJ and implement the class or method. It would be very convenient for the user if he or she just could add the class or method in the editor and then the code would automatically be generated. Code generation could also be helpful if the user draws a sequence diagram and code was generated based from the diagram. All classes used in the diagram were automatically created and code-stubs for the methods where generated. This would save lots of time for the user letting him or her focus on more important parts of the implementation then writing new classes and methods. On the other hand, writing new classes and methods is not as intuitive and easy for beginners to programming as it seems to experienced programmers who often find this process quite boring and time-consuming. But if the editor later should be extended and include more functionality for more experienced programmers, code generation would be a good functionality to add.

When implementing this kind of program it feels like it is never completed. There is always some new functionality that could be added, changed or removed. The editor today hopefully provides enough functionality for a beginner in programming to draw sequence diagrams in and is not too complicated or hard to understand and use. Since the editor has not been tested with any appropriate users, meaning beginners in programming, this theory is only theoretical and should of course be further investigated. Studies should be made with beginners to see how they find the editor and what improvements should be made to the editor.

Acknowledgements

I would like to thank Jurgen Börstler at the Department of Computing Science at Umeå University for all his help and ideas during the development of this thesis and for the help with the oral presentation. I would also like to thank Claes Gahlin for his help and support during the development of this paper. Finally I would like to thank my mother Monica Östling for reading this thesis and correcting many of the grammar mistakes.

References

[Beck, 1989] Beck, K. (1989). A laboratory for teaching object-oriented thinking. In *Object-Oriented Programming, Systems, Languages, and Applications*. http://c2.com/doc/oopsla89/paper.html.

[Björkander, 2003] Björkander, M. (2003). Det våras för uml 2.0. *Elektronik i Norden*, 10.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobsson, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley Longman, Inc.

[du Boulay, 1989] du Boulay, B. (1989). Some difficulties of learning to program. In *E. Soloway & J.C. Spohrer(Eds)*, pages 283–299.

[Fowler and Scott, 1997] Fowler, M. and Scott, K. (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley.

[Group, 2004] Group, O. M. (2004). Interim ftf report of the uml 2.0 superstructure finalization task force. Technical report, Object Management Group. http://www.omg.org.

[Harley et al., 1998] Harley, H. D., Seals, C. D., and Rosson, M. B. (1998). A formative evaluation of scenariobased tools for learning object-oriented design. Webpage. www.acm.org/crossroads/xrds5-1/eval.html , last checked 040302.

[Hussman, 2002] Hussman, D. (2002). Test first design with uml / “a picture is worth a thousand programmers”. In *Workshop on Teaching in XP (WTiXP2002)*. http://www.cwi.nl/wtixp2002/cfp.

[Jensen, 2003] Jensen, R. W. (2003). A pair programming experience. *Crosstalk, The Journal of Defense Software Engineering*. http://www.stsc.hill.af.mil/crosstalk/2003/03/jensen.html.

[Kölling and Rosenberg, 2001] Kölling, M. and Rosenberg, J. (2001). Guidelines for teaching object orientation with java. In *Innovation and Technology in Computer Science Education*.

[Kutar et al., 2002] Kutar, M., Britton, C., and Barker, T. (2002). A comparison of empirical study and cognitive dimensions analysis in the evaluation of uml diagrams. In *PPIG, Psychology of Programming Interest Group*.

[LeBlanc and Stiller, 2000] LeBlanc, C. and Stiller, E. (2000). Uml for undergraduate software engineering. In *Consortium for Computing Sciences in Colleges*.

[Lewis, 2000] Lewis, J. (2000). Myths about object-orientation and its pedagogy. In *SIGCSE 2000*. http://duke.csc.villanova.edu/lewis/presentations/mythsPaper.html.

[Miller, 2001a] Miller, G. (2001a). Conditional logic in sequence diagrams. http://www.-106.ibm.com/developerworks/library/j-jmod0605/index.html , last checked 040302.

[Miller, 2001b] Miller, G. (2001b). Introduction to sequence diagram. http://www.-106.ibm.com/developerworks/library/j-jmod0508/index.html , last checked 040302.

[Mrozek et al., 2002] Mrozek, Z., Mrozek, B., and Adjei, O. (2002). Teaching object oriented software engineering with uml. In *13th EAEEIE Annual Conference on Innovation in Education for Electrical and Information Engineering(EIE)*.

[Neubauer and Strong, 2002] Neubauer, B. and Strong, D. (2002). The object-oriented paradigm: More natural or less familiar. In *Consortium for Computing Sciences in Colleges*.

[Nordström and Börstler, 2002] Nordström, M. and Börstler, J. (2002). Objektorienterad analys och design med crc-kort, version 3.0. Technical Report UMINF 02.19, Umeå University, Department of Computing Science.

[Phillips, 1998] Phillips (1998). *Rational Unified Process*. Addison Wesley Publishing Company.

[Robins et al., 2003] Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172.

[Rumbaugh et al., 1997a] Rumbaugh, J., Booch, G., and Jacobson, I. (1997a). *UML Notation Guide, version 1.1*. Object Modeling Group.

[Rumbaugh et al., 1997b] Rumbaugh, J., Booch, G., and Jacobson, I. (1997b). *UML Summary, version 1.1*. Object Modeling Group.

[Rumbaugh et al., 2001] Rumbaugh, J., Booch, G., and Jacobson, I. (2001). *OMG Unified Modeling Language Specification*. Object Modeling Group.

[Sebesta, 1999] Sebesta, R. W. (1999). *Concepts of programming languages*. Addison Wesley Longman, Inc.

[Smialek, 2000] Smialek, M. (2000). Teaching ooad with active lectures and brainstorms. In *Object-Oriented Programming, Systems, Languages, and Applications 2000 workshop*.

[Software, 2001] Software, C. (2001). Object modeling with uml. Webpage, PDF-file. http://www.omg.org/news/meeting/workshops/ presentations/eai-2001/tuturial_monday/tockey_tuturial/1-Intro.pdf, last checked 040302.

[Westfall, 2001] Westfall, R. (2001). Hello, world considered harmful. *Communications of the ACM*, 44, No. 10:129–130.

[Williams and Kessler, 2003] Williams, L. A. and Kessler, R. R. (2003). Experimenting with industry's "pair-programming" model in the computer science classroom. Technical report, North Carolina State University and University of Utah. http://www.pairprogramming.com/csed.pdf.

A

User Manual

In this user manual it is described how the editor works and how to use it. The manual contains both a tutorial and sections about each functionality. After a short introduction the tutorial follows. After this each functionality of the editor is described and explained in more detail.

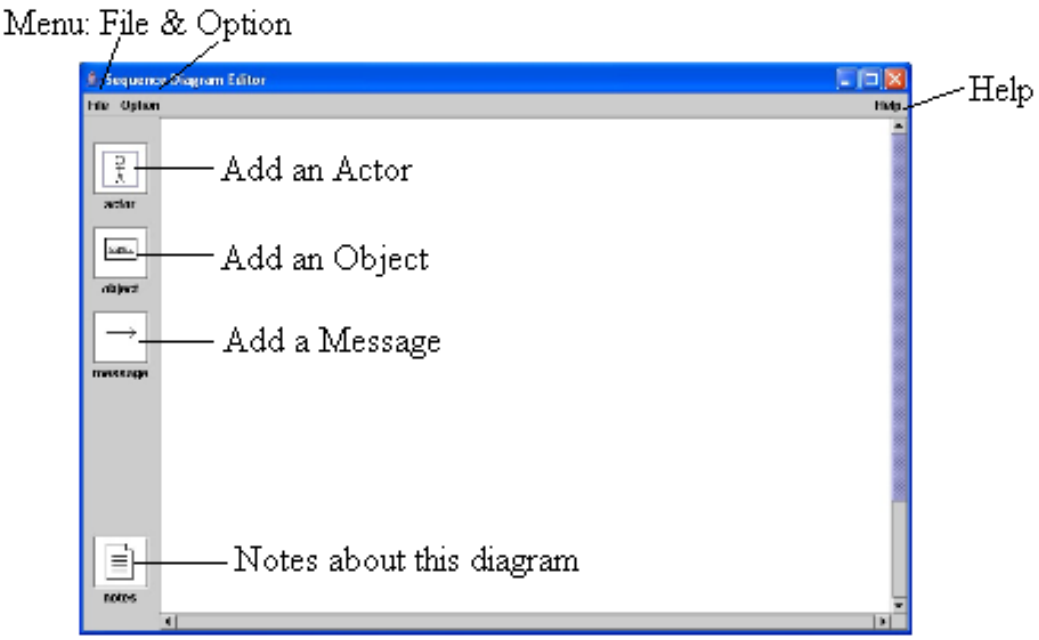

Figure A.1. A view of the editor with descriptions for the most important functions.

A.1 Introduction

This appendix will help the user to install and use the sequence diagram editor. Since the BlueJ program is a project still under development the plugin will only work together with the version 1.3.0 or later versions. To be able to run BlueJ and the plugin you will also need JDK version 1.4.2 or later installed on your computer.

A.2 Installation of the Plugin

To install the editor as a plugin for BlueJ copy the Jar file `Meditor.jar` into the `lib/extension` directory of your BlueJ installation. Next time BlueJ is started the plugin will appear in the menu of the BlueJ application.

A.3 Starting the Plugin

If the editor is started as a plugin to BlueJ it is started from the menu in BlueJ. Choose *tools* → *Sequence Diagram Editor* to start the editor.

If the editor is started without BlueJ the jar-file is executed to start the editor.

A.4 A Tutorial, Creation of a Sequence Diagram

When the editor has been installed and started a sequence diagram can be created. This tutorial will illustrate when the editor is in BlueJ-mode using a project called “people2” which comes as an example with the BlueJ program. The scenario described in the sequence diagram will display a secretary registrating a new student adding him or her to the database.

First an actor is added to the sequence diagram. This is done by clicking on the icon for an actor, see figure A.1. The name of the actor is then changed to “Secretary” by double clicking on the actor or by choosing from the menu. The menu of the actor is shown by clicking on the actor with the right mouse button.

Next step in the sequence diagram is the creation of a new student. The “Secretary” will create a new student to add to the database. This is done by first choosing to add a new message to the diagram, see A.1. Mark the start for the message on the lifeline of the actor (“Secretary”). Then click with the right mouse button and the editor will ask if *“Do you want to create a new object with this message?”*. Answering yes will create a new object. First choose which class the object should be created from and then write the name of the object, see figure A.6. The sequence diagram will now look as in figure A.2.

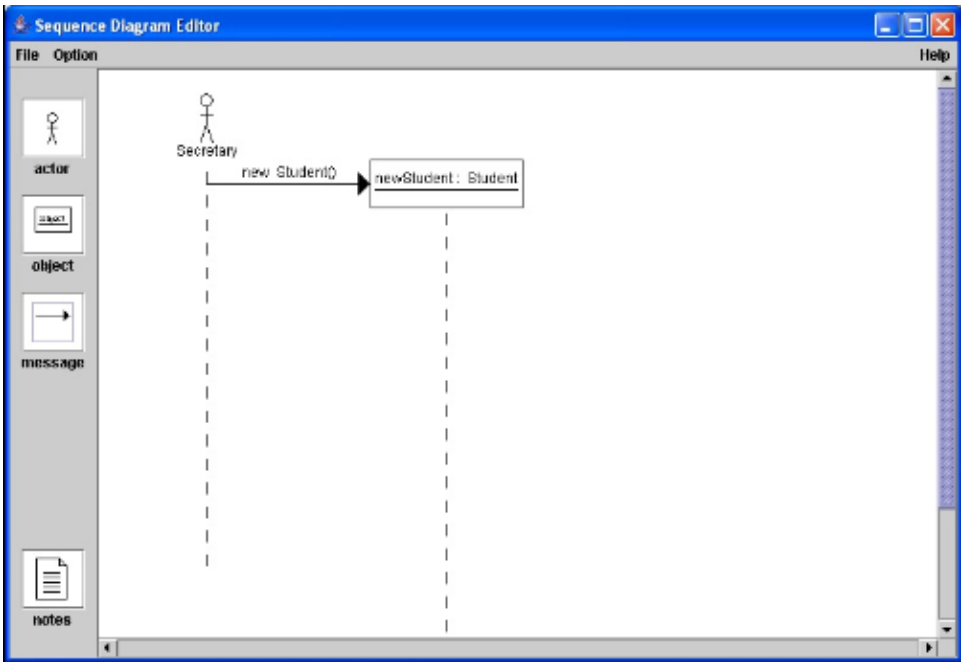

Figure A.2. The actor and the new student have been added to the sequence diagram.

Next step will be that the Secretary sets the address of the new student. This is done by making a call to the method “setAddress” which the student has inherited from its super class Person. The message call is made by first choosing to add a new message. First click on the lifeline of the actor, who will make the method call, then click on the lifeline of the student to which the call is made. Then choose which method to use in the drop down list and click “OK”. The diagram will now look as in figure A.3.

When a message is added to a sequence diagram the default in the editor is that the message gets a return message. If the type of the return value is known it will be added to the return message. Return messages can be switched on and off in the menu under *option → show return messages*. This could be done any time during the development of the diagram.

If a message has been added to a sequence diagram and the position of it is not satisfactory the message can be moved. This is done by selecting the message, by clicking on it, and then click on the handle[1] of the message and hold the left mouse button down and drag the message to the right position.

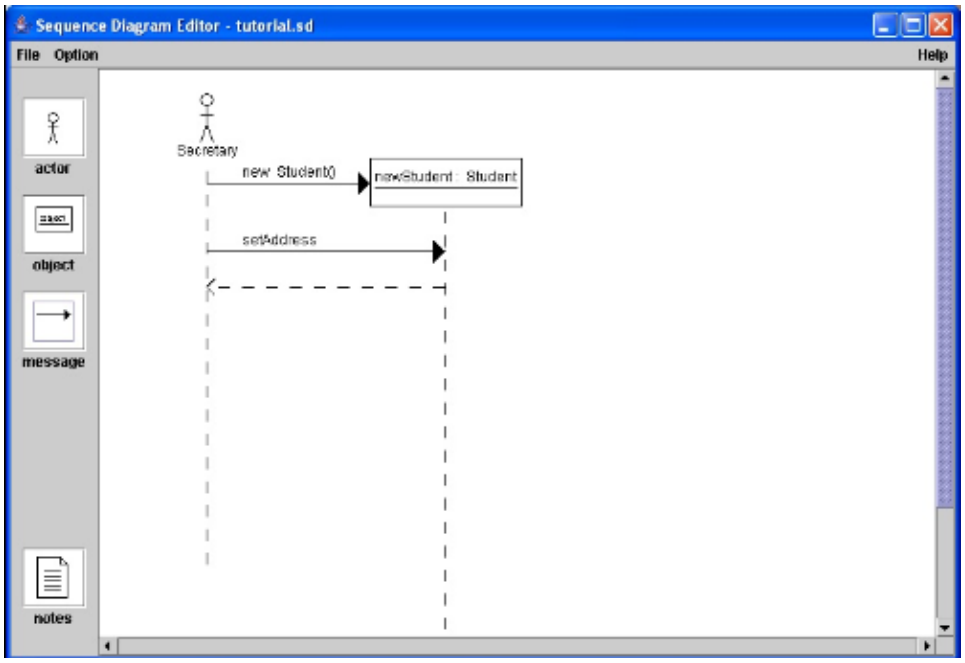

Figure A.3. The actor and the new student have been added to the sequence diagram. The actor calls the method “setAddress” to let the student set its address.

[1] A little square that will appear when the message has been selected.

When the method “setAddress” is called on the student it creates a new object of the class Address and sets its own address. This has to be displayed in the sequence diagram by letting the student create a new object of the class Address before returning the method call to setAddress, done by the actor. This is done by choosing to add a new message. Mark the start point of the method call on the lifeline of the student, since it is the student creating the new Address. To clarify that the new Address is created before the method call to “setAddress” returns the start point of the message should be placed between the endpoint of the “setAddress”-message and the return message belonging to it. Then create a new object from the class Address as described before. The look of the diagram will now be as in figure A.4.

If a message, an actor, or an object has been added to the diagram but should be deleted this is done by clicking on the component with the right mouse button and a menu will appear. Just choose “delete” from the menu and the component will be deleted from the sequence diagram.

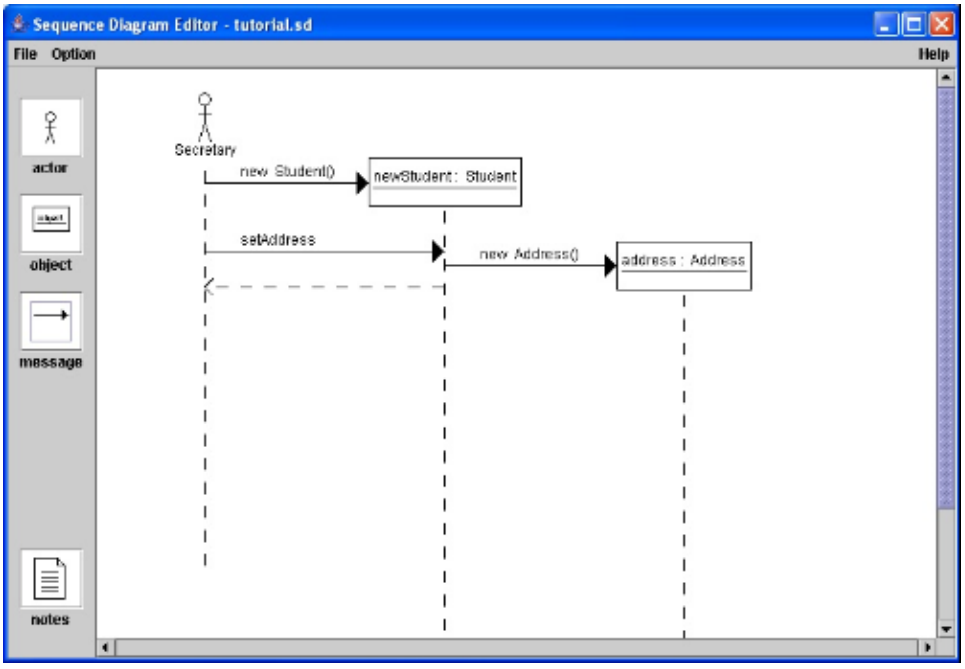

Figure A.4. The student creates a new object of the class Address to be able to set its own address.

Next and last step in the sequence diagram will be the Secretary adding the new student to the database. First we have to add an object of the class Database to the sequence diagram. This is done by choosing to add a new object to the diagram by clicking on the icon for an object, see figure A.1. Then choose which class to create the object from and write the name of the new object, see figure A.6. Now the Secretary adds the new student to the Database by calling the method “addPerson” in the database. To add this method call into the diagram first choose to add a new message, click on the lifeline of the actor and then click on the lifeline of the database. Then choose which method in the list (“addPerson”) and click “OK”. The diagram will now look as in figure A.5.

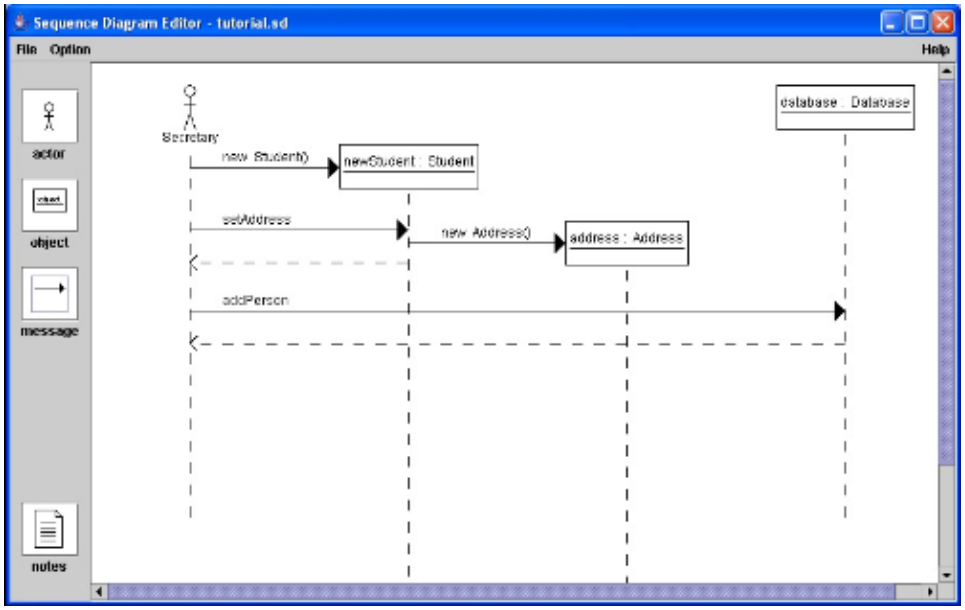

Figure A.5. The finished sequence diagram.

Now the sequence diagram is complete and all objects and messages have been added. If the diagram has not been saved yet this is done by either pressing `Ctrl-S` or choosing `save / save as` from the file menu. The name of the sequence diagram is typed and then click “OK”, see figure A.9. The file of the diagram will be saved in the same directory as the project it belongs to, that is, the currently open project in BlueJ.

A.5 Adding Components to a Sequence Diagram

If the editor is started the user can choose to create a new sequence diagram or open a saved one. To create a new sequence diagram the diagram is drawn in the drawing area and later saved under the menu *file* → *save as*. Otherwise a saved sequence diagram is opened using the menu *file* → *open*.

A.5.1 Adding an Actor

To add an actor to the sequence diagram just click on the icon for an actor, see figure A.1, and an actor will be added to the drawing area.

A.5.2 Adding an Object

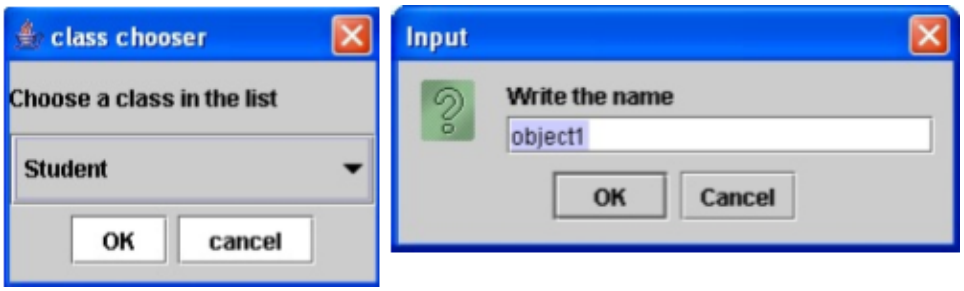

Figure A.6. If the editor is in BlueJ-mode the user first has to choose the class of the new object (the window to the left) and then the name of the object (the window to the right).

To add an object to the sequence diagram click on the icon for an object, see figure A.1. If the editor is in BlueJ-mode the class of the object first has to be chosen, see figure A.6. This is done by selecting one of the classes in the list and then click on the “OK”-button. After this the user has to fill in the name of the new object. If a new name is not given the object will be given a default name like “object1”.

A.5.3 Adding a Message

To add a message to the sequence diagram click on the icon for a message, see figure A.1. To select the start object of the message, meaning the object making the method call *to* the end object, click on the lifeline on this object (or actor). To select the end object of the message click on the lifeline of this object. If the editor is in BlueJ-mode the user has to choose a method. This is done by selecting one method in the list and click on the “OK”-button, see figure A.7. If the option “Show return messages” is selected in the menu under *option* each added message

Figure A.7. If the editor is in BlueJ-mode and a message is added the user has to choose which method to add by selecting one from the list.

will get a return message belonging to it. If the return message has a type this will be added as a label to the return message.

A.5.4 Adding a Destroy Symbol

A destroy symbol can be added to an object or to an actor. This is done by clicking on the object or actor with the right mouse button and a popup menu will appear. The destroy symbol is added by selecting *create destroysymbol* in the menu and then click on the lifeline of the object or actor where the destroy symbol should be placed.

A.5.5 Notes about the Sequence Diagram

Figure A.8. The window where the user can write notes belonging to a sequence diagram.

Each sequence diagram has a note (a text field) belonging to it. The note can contain information about the diagram for example if the user wants to point out something special about the diagram, a general description of the scenario etc, he or she can write it here. To write notes for a sequence diagram just click on the

icon for notes, see figure A.1. Then a window will appear, see figure A.8, where the notes can be written. Before closing the window click on the *save* button and the notes will be saved together with the sequence diagram.

A.5.6 Return Messages

Every message that is added to a sequence diagram will automatically get a return message. If the editor is in BlueJ-mode the type of the return message will be added as the label of the return message. The user can choose whether to display the return messages or not. This is done under *option* → *Show return messages*. Return messages can be switched on and off during the development of a sequence diagram.

A message having the same object as its start and end object will not get a return message.

A.5.7 Creation of a sequence

Messages added to a sequence diagram can be arranged in a sequence. All messages, including return messages, in a sequence will automatically be arranged with the same distance between them. If a message in a sequence is removed the messages after the removed message will be removed from the sequence.

To add message-M2 to a sequence after message-M1 message-M2's start point has to be placed between message-M1's endpoint and its return message. If this is done message-M2 will be added to the sequence and all messages and return messages in the sequence will automatically be arranged.

A.6 Save and Open a Sequence Diagram

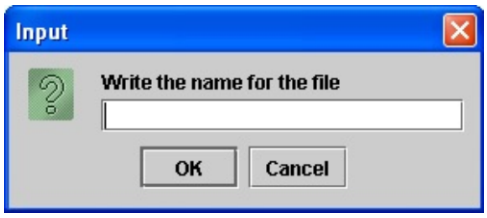

Figure A.9. If the editor is in BlueJ-mode and the sequence diagram should be saved the user just has to fill in the name of the file

To save a sequence diagram chose *file* → *save as / save*. If the editor is in BlueJ-mode only the filename has to be filled in, see figure A.9. The file of the sequence diagram will be saved in the same directory as the project it belongs to.

If the editor is not in BlueJ-mode the user chooses the name of the file and the directory where the file should be saved. To save a sequence diagram the user can also press `Ctrl-S`.

To open a saved sequence diagram choose $file \rightarrow$ *open*. Then choose which file to open and click OK. If the editor is in BlueJ-mode only the files belonging to the currently open project are shown. To open a sequence diagram the keyboard can be used by pressing `Ctrl-O`.

A.7 Moving, Deleting and Editing

A.7.1 Objects

Delete To delete an object click on the object with the right mouse button and choose *delete* from the popup menu. When deleting an object all attributes belonging to it will also be deleted, such as messages, destroy symbol, lifeline etc.

Rename An object can be renamed in two different ways. The name can be changed via the menu that appears when the object is clicked on with the right mouse button. The name can also be changed by double clicking on the object.

Move To move an object first select the object by clicking on it. The object will then be drawn in red and a handle (a small square) will appear. To move the object click on the handle and "drag" the object (with the left mouse button pressed down) to the new position. When the mouse is released all objects and actors in the sequence diagram will be rearranged automatically and the distance between them will be the same.

A.7.2 Actors

Delete To delete an actor click on the actor with the right mouse button and choose *delete* from the popup menu. If an actor is deleted, all attributes belonging to it, like messages, destroy symbol, lifeline etc. will also be deleted.

Rename An actor can be renamed in two different ways. The name can be changed via the menu that appears when the actor is clicked on with the right mouse button. The name can also be changed by double clicking on the actor.

Move To move an actor, first mark it by clicking on the actor. The actor will then be drawn in red and a handle (a small square) will appear. To move the actor

click on the handle and “drag” the actor (with the left mouse button pressed down) to the new position.

A.7.3 Messages

Delete To delete a message click on the message with the right mouse button and choose *delete* from the popup menu. The message will then be deleted. If the message has a return message this return message will also be deleted.

Rename There are two ways to rename a message. The message can be renamed by clicking on the message with the right mouse button and choose *change message* from the popup menu. The message can also be changed by double clicking on it. Return messages can also be renamed in the same way.

Move If a message is selected by clicking on it a handle appears. By clicking on this handle and dragging it, the message (or return message) can be moved vertically.

Creation message If a message is to create a new object this is done by first selecting the start object for the message, the object that is going to create the new object. Then click with the right mouse button and the editor will let the user choose which new object to create. The new object will then be added to the sequence diagram.

A.7.4 Destroy Symbol

Delete To delete a destroy symbol click on it with the right mouse button and choose *delete* from the popup menu.

Move To move a destroy symbol select it by clicking on it. The destroy symbol will then be drawn in red and a handle will appear in the middle. To move the destroy symbol click on the handle and “drag” the destroy symbol (with the left mouse button down) to the new position. The destroy symbol can only be moved up or down along the lifeline it belongs to.

A.7.5 Lifeline

The length of a lifeline can be changed. To change the lifeline first mark it by clicking on it. The lifeline will then be drawn in red and a handle will appear at the end of the lifeline. To change the length of the lifeline click on the handle and “drag” the handle to the new position for the end of the lifeline. The handle can only be moved vertical along the lifeline.

A.8 Known Bugs

There are a number of known bugs in the editor and surely a number of unknown too. Since the program still needs some adjustments and more functionality there are some features left to implement. One of the biggest bug is that the program is not all “waterproofed”, it is not that hard to have it crash. If the editor is correctly used it works OK, but if the user starts to “test” the program there will probably happen strange things.

The known bugs in the editor are:

Sequences If a sequence contains two messages which are messages back to the same object in a row this will cause problems. New messages can only be added at the end of the sequence and not in the middle of an already existing sequence.

Consistency check When a sequence diagram is opened it is checked against a BlueJ project. But only classes are checked and not methods.

Moving components Sometimes when different objects or actors are rearranged in the drawing area all attributes belonging to them do not get the correct new position.

Creation messages A creation message can not be in a sequence of messages.

Return messages When a regular message is added to the sequence diagram it gets a return message placed in the right positon. But later on the user can move this return message to any position without any controll from the editor.

Inheritance If a classA inherits from classB, all classB’s public methods are also shown for classA. But if classB inherits from classC, classC’s public methods should be shown for classA which is not the case. The inheritance is just shown in one step in the editor.