University of Oslo
Department of Informatics

Teaching OOP using graphical programming environments

An experimental study

Richard Edvin Borge

13th January 2004

This thesis is written as a part of the COOL project. The COOL project, short for Comprehensive Object Oriented Learning, was started by Kristen Nygaard in 2002 and is a project with many participants from different academic environments. The main objectives of the COOL project are: Exploring the complex area of learning (and teaching) object-oriented concepts; Maintaining and further developing the Norwegian (Scandinavian) heritage from object-orientation; Designing blended learning environments. My focus is on learning environments and it is my hope that this thesis will be a helpful contribution to the COOL project.

I would like to thank my supervisor Jens Kaasbøll for much constructive input and for making this thesis a very enjoyable process. I would also like to thank Christin Borge, Vera Louise Hauge and Annita Fjuk for reading through my thesis and giving me helpful pointers. Finally I want to thank all the members of the COOL project group for helping me realize how interesting the field of learning and teaching computer programming really is.

Contents

Chapter 1

Introduction

When I learned to program in Java, I did not like it much. I liked the idea of programming and being able to tell the computer what to do, but I did not really understand Java. It was not until I had a course that looked more into Object Oriented design that a light bulb went on in my head and programming in Java became much easier. Object Oriented design was for me a new way to make Java programs as opposed to my introductory course.

Computer programming is a fairly young discipline seen in a historical perspective, and the discipline of OOP[1] is even younger. Object Orientation was first introduced with the creation of SIMULA 1 in 1965 [16] but was largely constrained to research. One can say that the term *object-oriented* became a fashion word in the 1980‘s. However, in the early 90‘s there was still a broad reluctance in bringing the OO paradigm into the undergraduate curriculum [27]. The main focus on programming was procedural programming, e.g. Pascal and C, and OOP was still reserved for the more advanced courses.

When OOP was introduced to the undergraduate curriculum, the ”procedural” approach to teaching programming remained and students started learning OOP the procedural way. Many educators claim that this is not a good way to teach OOP and that one has to change not just the way of thinking but also the way of teaching. Hence, the field of teaching OOP in a CS1[2] course is divided into two main fractions: Those who think the education should start off with objects right away, and those who think it is important to give the students some practical programming experience before they learn OO concepts. The first fraction mentioned is also referred to as the *Objects First* [13] fraction, while the second fraction is

[1]OOP means Object Oriented Programming and will be the standard abbreviation for the rest of this thesis

[2]CS1 is short for Computer Science 1 and is very often the abbreviation for a introductory course in computer science and computer programming.

referred to as *Procedures First*. Here follows quotes from one representative from each side. First, Jeremy Gibbons [18]:

> *We argue that for computing majors, it is better to use a 'why' approach to teaching programming than a 'how' approach; this involves (among other things) teaching structured programming before progressing to higher-level styles such as object-oriented programming. (...) Java is a reasonable language for doing so.*

The 'how' and the 'why' are names for approaches used. In a 'how' approach, the students learn methods for how to solve an assignment. In a 'why' approach, the students also learn why they use a particular method to solve an assignment.

Second, Joseph Bergin [10]:

> *My conclusion is that procedural programming is no more a good introduction to object-oriented programming than it is to functional or logic programming.*

The most common languages used in CS1 today are Java and C++, both object oriented languages. In C++ it is possible however, to program in a procedural C style if one wants too, and it is possible to combine procedural programming and OOP. In Java, on the other hand, one is introduced to the Object-Oriented philosophy right away: Even your first program must be wrapped in a class[3].

Many Object Oriented languages have been introduced for teaching OOP in undergraduate courses, e.g. Smalltalk, ADA and Eiffel, with varied degrees of success. Also, many educators were reluctant to switch from procedural to object oriented programming. Some of the reasons for this reluctance were mentioned and dispelled in [17]. According to this article, the top ten reasons why OO should not be taught in CS1 are:

1. OOP is just a fad!
2. OOP is too hard for my CS1 students!
3. The dreaded paradigm shift!
4. You still need algorithms!
5. The "OOP-ish" overhead!
6. OOP languages are ugly!
7. There's already too much material in CS1!

[3]An OO program is made up of classes.

8. It screws up the rest of our curriculum!
9. It fits perfectly with CS2 and data structures!
10. OOP is too hard for us!

This article was written in relation to C++ and its introduction into a CS1 course in a college. I will come back to this list in chapter 8 and discuss some of these reasons in relation to the work I do in this thesis.

Many papers have been written on the use of Objects First, but little empirical data exist on the success/failure of such an approach. An example of this is [10] quoted above. This article adds many arguments to the discussion about Objects First versus Procedures First programming, but as far as empirical data, there is none. An example of an article with empirical data on the use of Objects First is a case study Böstler, Johansson and Nordström [5]. In this case study, the approach was Objects First using BlueJ[4] and CRC (Class Responsibility Collaboration) cards.

> *The results of our case study indicate faster and better understanding for object-oriented concepts through delaying the introduction to complete applications and I/O. Being able to explore classes and methods without worrying about the context in which they appear, supports understanding without the confusing syntactical details.*

The article is fairly new (2002) and illustrates an approach used in an Objects First coursework that shows an improvement in results.

We note that BlueJ is a graphical environment and in the case study above, this was used in combination with CRC as supporting tools to an Objects First approach. In this thesis I have chosen to test an alternative Objects First approach: The use of graphical environments as the only supporting tool. My main research question is:

> **In an Objects First approach to learning OOP and specifically Java, how does the use of graphical environments influence learning?**

I am looking at Java because this is the language being taught at my faculty and it is a language where you can not avoid the use of classes. In C++ it is possible to program procedural C as well, but this is not possible in Java in the same way: One still need to use OO specific terminology such as **class** even if the program never generates any objects.

Many educators have for a long time suggested and tried the use of graphical environments as an aid to teach OOP. These environments give a visual display of programs and might be able to make the teaching of

[4]BlueJ is a tool that I will be presenting more thoroughly later in this thesis

OOP easier. In this thesis I look more closely and examine two graphical environments: **Karel J and Robocode**. In addition I will look at BlueJ, but not in the same degree as the two other. These environments will be presented in chapter 4.

A basis for answering my main research question will be two experiments with students using the Karel J and Robocode environments. The reason for not doing an experiment with BlueJ is that there is a limit to how much time I have. I therefore chosen not to focus on BlueJ because it is the environment that has been tested most of the three mentioned above. One result of a course where BlueJ was tested is quoted above. There exist little material on the effects of the use of Karel J and Robocode in the literature. BlueJ also differs a lot from the two other environments: In BlueJ, the code is represented in a UML like diagram. In Karel J and Robocode, during the program execution, each object in the code is represented with moving objects in a certain setting: A grid for Karel J and a battlefield for Robocode. In BlueJ it is possible to make any kind of program and system and this will be visualized in the diagram. In Robocode and Karel J, the programs are made specifically for the environments, the students are limited in that the program must work with the environment.

To help me answer my main research question, two supporting questions must be answered:

1. What do the students learn during teaching with Karel J and Robocode?

2. What do the students find difficult?

This thesis is divided into seven chapters:

In the first chapter I will look at some general theory about OOP and briefly mention concepts that are important for OO understanding in order to see if I am able to recognize these in my experiments. The next chapter outlines some basic learning theory, both general learning theory and more specific learning theory concerning teaching of computer science. These two chapters are the basis for my experiments.

In chapter 4, I take a closer look at the programming environments I have focused on. These environments represent three different approaches to OOP and cover the general thought behind many other environments not mentioned here.

In chapter 5 I describe various possible methods for gathering data and explain the methods I have chosen, namely experiments and observation. The experiments are described in depth in the next chapter, where I also give a long summary of them as a basis for my discussion.

In chapters 7 and 8 I link the data from the experiments to the theory mentioned in earlier chapters, tie up loose ends and make conclusions.

Chapter 2

OO theory and use

In this chapter, I will present some basic concepts from OO theory and motivate the use of OOP. This theory is general and applies to all OO languages. In later chapters, I will look back at what I have presented here and see if the theory is supported by the environments[1] I have chosen to use in my experiment.

2.1 What is OOP?

The thought behind the use of OO is that the world we wish to model, or ‘make a computer program of‘, can be separated into *classes*, representing different parts of this world. Each class contains the data and methods unique for the part of the world this class represents. From these classes we can generate *objects*, representing one unique entity of the class with its own set of data.

In brief, when making a program using OOP we split the program into parts sharing the same data and responsibility, so the program is easier to organize. For instance, if we are making a bank system, we can say: ‘OK, the customers need this data stored about them, and they will need these methods´. We therefore make a class called customer, and then the responsibility for the customers has been assigned to this class, and we do not have to think more about how to implement methods for the customers. Instead, we can use the methods implemented, knowing what sort of result they produce.

Procedural oriented programming languages also have the option to store data in separate parts, e.g. **structs** in C. However this can only store data and not methods. OOP offers a way to store everything related to certain part in a separate chunk and in that way represent a program in a way that will be easier to implement and maintain.

[1]These environments will be explained explained more in depth in the next chapter

A big problem with OO programming languages, and especially Java, is that because of the use of classes and objects, short programs come with just as much wrapping as long programs (see example below). Students are introduced to many new concepts from the start. In the example below, a much used starting example, there are many new terms introduced in addition to the method that prints the text: **class, public, static**[2]**, void, main, String, []**.

Listing 2.1: Intro example

```
class HelloWorld {
    public static void main( String [] args ) {
        System.out.println("Hello World!");
    }
}
```

The problems this can cause in teaching OOP will be looked at in the next chapter.

2.2 Comparing Procedural and Object Oriented

What is the difference between procedural and object oriented programming? Is one approach easier than the other for students to learn? In a study on the use of different methodologies for developing requirements specification in information systems, Vessey and Conger conclude the following [15]:

> *Results suggest the process methodology was the easiest for novice analysts to learn over time, followed by the data and then the object methodology.*

This suggest that people are not so familiar with the object oriented approach as with the procedural approach to a problem. Later in this thesis, experiments will show that the object oriented approach may not be so hard to understand after all. I will now continue with comparing the procedural and object oriented approach.

I am now going to build two houses, first by using a procedural approach, then using an object oriented approach:

[2]This is a very complicated term and one that many lecturers avoid, using different strategies. The word static is also a very bad name for what it does in the program, as it refers to the idea that a static variable has the same value for all objects in the class also called a class variable. Static can often be confused with final, referring to a variable that can not be changed.

2.2.1 A Procedural house: Brick by brick

40-50 years ago, nearly everyone knew or were expected to know how to build their own house, and so most did. Now I will do the same, I might skip some details as I am really not a good builder. I start out with digging out the area where I want my house, using some tools, like a bulldozer and maybe some dynamite and set up water and plumbing with some help from someone who has done it before (maybe an older brother or friend who has already built a house). Then I start building the foundation, with bricks and cement. After the foundation is done, I build the framework with a hammer nails and a saw. After the framework is done, I put in windows and a roof. Then I dress the outside walls, isolate and put up the electrics before dressing the inside walls. After this, the painting and wallpapering start before putting in furniture and such.

This was a very short and faulty house-project, and the key concept from this approach is: I do all the operations, I build everything and therefore need knowledge of the whole process. I might get some help from some people I know to do some of the hardest bits, but I mainly do things myself, brick by brick. I have to be aware of all the flow of information flowing through this project, in fact I am the one holding all the information.

2.2.2 An Object Oriented house: Section by section

Today, few people know how to build their own house, actually, it is not allowed due to rules and regulations. So what do I do now then? I hire people to do the jobs for me. All I am left with now is knowing the logical order of the different sections and who can do the different jobs. First I hire someone to dig out my property and then someone to do the plumbing. Then I hire someone to build the foundation, before I hire someone to make the framework. Then a roof worker, an electrician and a painter.

The key point from this process is this: I want the job done, but I don't know how the different parts are made, only how they are to be put together in the end. I don't need the knowledge of how to do a job, I just need to know who can do it and that they can actually do it. I no longer need to keep track of all the information flowing through this project, it is held by many different actors, I just need to know of results and the most important information: The logics of the house project.

2.2.3 Different views

The two ways of building a house can be translated into the world of computer programming. To answer the question "What is the difference between Procedural and OOP?": Procedural programming requires a lower-level knowledge of programming than in an object oriented setting. This means that as a programmer one must know the specifics of all the methods and in many situations one must write these methods. And argument here is: The methods must be written in OOP too. This is true, but often in OOP one reuse classes previously written. These classes contain more information than just a few specific methods and will therefore be more flexible than procedural methods that can be reused.

Object oriented programming is like a client-to-server relationship: The classes of the program are servers that provide some sort of service that the client needs, and the client uses these services without having any knowledge of how the job is done as long as it is done correctly.

In a small program where the amount of information is small, it is more tempting to use a procedural language because of the amount of code that need to be written. Object Oriented languages need a lot of wrapping of code even for very short programs and will in some cases result in more wrapping than code. No one would ever use Java if the problem was printing some text to a screen, but rather use languages like BASIC or C. But when the amount of information flowing through the program becomes vast, OOP offers a better structure and the risk of making errors decrease.

Is the object oriented way of viewing the world so very far from how the real world is? Every day we use services without having any idea of how they work. All we know is that they do what we want them to do. This sort of high-level knowledge is something we all have but might not reflect to much on. I believe that this way of thinking can be taught just as easily as a procedural way of thinking. To think object oriented, being a technical wiz in imperative programming [3] is not necessary and the code specific aspects of programming can come at a later stage: For instance, Java offers many sorting, storing and search algorithms that are ready for use.

2.3 Some OO and Java theory

Moving past language specific terminology, OOP has four key concepts (van der Linden): **Abstraction**, **Encapsulation**, **Inheritance** and **Polymorphism**. These concepts are collected from "Just Java" by Peter van der Linden [28]. There are of course other aspects of OOP, some of which will also be presented here.

[3]A programming style that describes computation in terms of a program state and statements that change the program state (Wikipedia on-line encyclopedia).

2.3.1 Abstraction

When making a computer program to represent a part of the real world, we have to extract the characteristics that are relevant for our program. These characteristics are dependent on what we are trying to do. To illustrate, I will use a car as an example: A repair shop would represent the car in their computer system by license plate number, billing information, past work, work description and owner. An insurance company would represent the car in their computer system with owner, past damage, price of the car, model, and so on.

These are examples of abstractions of data. Abstraction of data can be seen as removing insignificant details from an object so that what is left is that which is important in our computer system. The objects then become a model or representation of the part of the reality that the computer system is made for.

Abstraction is not something that only applies to OOP of course, abstraction is a skill that is applied to any programming language. However, abstraction leads up to a very important concept in OOP: **Encapsulation**.

2.3.2 Encapsulation

Encapsulation goes one step further than abstraction. Just as data is important, so are the operations that can be performed on it. Putting the data and the operations together is the principle of encapsulation. Then one easily sees that there are some data representing an object and that there are some legal operations that can be performed on these objects.

Another aspect of encapsulation is the opportunity to hide certain data and prevent that these data are changed by accident (by using the **private** or **protected** keywords).

In C for example it is possible to collect all data about an object in a class like structure (called a **struct**), but it is not possible to put any methods in this struct. This is something that makes OO languages unique: Collecting data and methods associated to this data and putting everything in one part of the computer system for later use.

Going back to the car example from the previous sub chapter: Once we have abstracted the data we feel are relevant for this type of car system, say physical data like color etc., owner, license plate, price and frame number (unique to each car to prevent theft), we can put this into a class called **Car**. In this class we declare all these variables and methods the **Car** class can perform. These methods include **changeOwner(String name)** and **printData()** to mention a few. The **frame number** would be declared private to prevent it from being changed anywhere but in the object itself. In this way we can control that no outside system can easily access or change this frame number by putting a control method in the object for when access or change is requested.

The strength of encapsulation is that one has the opportunity to complete parts of the program with methods and variables and then put these parts away and only use the methods required later in the program. This reduces the amount of information the coder has to keep track of while designing and writing her program.

2.3.3 Inheritance

Inheritance is also an OO concept. As an example, consider the hierarchy of how different species have evolved: All species that live today has inherited some characteristics from earlier extinct species. Take humans for example; we have inherited some common characteristics from earlier species that make us into humans, along with many other characteristics we have inherited from later species that makes us unique. In OOP this relation between older and younger species is referred to as **Subclasses** and **Superclasses**.

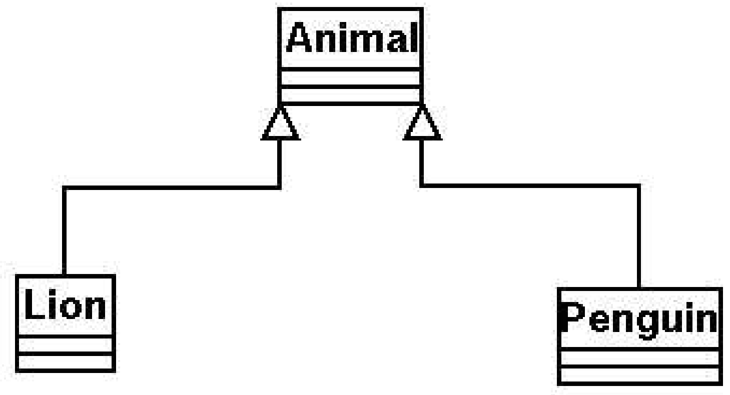
Let us say that I want to make a registry of animals in a zoo. All animals have some common data about them (name, id, cage number, age etc.), but there are also many differences: The penguin does not eat the same as the lion, and they do not share the same preference for water. It would be hard to make a common class of all animals without having to write many special methods and cases within the class. It will also be a bit redundant to make two classes where much of the information must be repeated. A possible solution and indeed much of the power of OOP can be seen in figure 2.1: We declare a superclass called **Animal**, where all common characteristics for the animals are stored. Then we declare a subclass for each species of animal where the characteristics unique for a certain animal is stored.

Now the **Lion** and **Penguin** classes have inherited all the methods and variables from the **Animal** class while keeping their unique characteristics as well. In a zoo there are many different species, so making many subclass levels is possible, for instance: **Animal** -¿ **Mamal** -¿ **Predator** -¿ **Lion**.

Inheritance is only possible because of encapsulation. A hierarchy over different entities in a computer system will make expansion and alteration of the system much easier, just imagine wanting to add a new data field to only the predators in the zoo, not any of the other animals.

The **Lion** and **Penguin** may share methods from the superclass, like for instance $feed()$ or $cleanCage()$, but these methods would not contain the same data: Cleaning the cage of a lion requires more preparation than just having a sardine to ward off annoying penguins. So the method

Figure 2.1 Inheritance



is shared, but the contents of the method may vary. This leads us to another key concept: **Polymorphism.**

2.3.4 Polymorphism

Polymorphism is the final key concept from OOP suggested by [28]. In Greek, this word means 'many shapes' and in OOP it refers to redefining or rewriting previously defined methods. I will illustrate with examples from the two types of polymorphism that exists in Java: **Overloading** and **Overriding**.

Overloading

Overloading means that you use two methods with the same name but with different signatures[4]. Here is an example of two methods with the same name but with different signatures (example from the Java String class):

> int **indexOf**(String str)
> int **indexOf**(String str, int fromIndex)

When this program is compiled, the compiler will know which of these methods to use based on whether the method call in the program has one or two parameters.

[4]A signature of a method will be the name of the method plus the number of parameters plus the returntype: **int indexOf(String str)**

Overriding

Overriding is best illustrated with an example. Let us say we want to rewrite the HashMap method *remove* and extend it with a return message:

Listing 2.2: Example HashMap

```java
class HashMap2 extends HashMap {

    public HashMap2() {
        super();
    }

    Object remove( Object key ) {
        super.remove( key );
        System.out.println( key.toString() + " removed!");
    }

}
```

If we now make a **HashMap2** class, we will be able to use all the methods that HashMap has because we are extending the class, but in addition, we will get a little extra message when calling the *remove* method. As we see in the example, the *remove* method also calls the superclass method *remove*. As we see again, overriding is possible because of inheritance and encapsulation.

2.3.5 Other aspects of OOP

OOP has, of course, many other aspects, and I will mention them briefly below. These help in the general knowledge of OOP.

Implementing one domain in another

As seen above, an advantage with encapsulation is that it is possible to complete parts of a program and store this in separate classes to be used later in other domains[5]. To illustrate with an example: In a programming environment called Karel J presented in depth in chapter 4, students program robots in order to learn to program in Java. These robots have some commands that are used to maneuver around and pick up or place items in a grid. Using these robots to place items around in a grid, it is suddenly possible to make drawings with these items. It could then be possible to make separate classes for making lines, circles, diagonals etc.

[5]By domains I mean another program that illustrate another part of the world

This is a good way of illustrating reuse of code and that classes in a program are stand-alone entities that can be used for different purposes.

Active objects

Active objects can be understood in different ways.

1. One way is to think of objects as independent entities that can communicate with each other and behave according to each other instead of taking commands from the programmer, i.e. handling events based on the other objects in the system.
2. The second way to think about active objects is as objects that work dependent on conditions set in its own code, i.e. if it is Sunday in the system, the objects handling customers add interest to the accounts.

The concept that objects in a program are entities that operate independent of each other and communicate, stems all the way back to the birth of OOP. It was always the thought of Nygaard and Dahl, the inventors of the first Object Oriented language SIMULA, that objects are active parts of the program. This is mentioned in the first book written about teaching OOP [11]:

> *We think of a system as a collection of objects, each operating more or less independently from the others according to specified action patterns.*

A programming environment called Robocode, presented in depth in chapter 4, illustrates active objects very well. In this environment, tanks battle it out on a square battlefield. Whenever one tank sees another tank or is hit by a bullet or crash into another tank, an event is created that the tank can work with. It is the student's task to decide how a tank shall handle these events.

Modeling the world

In the book *Objects first with Java - A practical introduction using BlueJ* (2003) [3] the concept of modeling the world outside the computer using objects is used as the approach for teaching students OOP. These object represent the part of the reality that the program is supposed to model. In this approach, modeling skills are developed before coding skills. Making models of the world require that the classes contain some data that is recognized by the students and can be related to the real world. Below is an example that is revisited in chapter 4, figure 4.2:

Listing 2.3: A car dealership

```java
// The Super class
public class Cars {
    // Common attributes
    private String regNumber;
    private Color color;
    // And more...

    // Common methods
    public void displayColor() {
        // Method definition...
    }

    public void changeReg(String newReg) {
        // Method definition...
    }
    // And more...

}

// Subclass of Cars -> SUV
class SUV extends Cars {
    // Attributes only for SUV
    private int vat;
    private boolean terrain;
    // And more...

    // Methods only for SUV
    public void changeTax(int newVAT) {
        // Method definition
    }

}

// Similar for the other classes
```

Using cars as an example is used by many educators as cars contain a lot of data and because cars are something most students can relate to.

Objects and pointers

A program written in an OO language consists of objects working together, exchanging information, producing results, storing and retrieving data and much more. Each object has a certain responsibility, such as reading a file, storing data about a customer in a bank, running a thread etc. Knowing how to make objects and access their information and methods is the essence of OOP and it comes with understanding many of the concepts above.

Constructors

Constructors are common for object oriented languages. In brief a, constructor is a method that is initiated when a new object of a class is created to assign certain values to the object and otherwise start the object (by initiating new methods). Constructors can be empty and there can be multiple constructors, called with different parameters, a reason why polymorphism is important to understand.

In the sub chapter ”Methods and Parameters” a little further down, I will briefly look at the difficulties of methods and parameters. If an Objects First approach is used, will the introduction of constructors generate serious problems of understanding?

2.3.6 General concepts in programming

There are concepts in OO languages that are common for most programming languages and students will meet these concepts sooner or later as they learn to program. I will only mention a few I see as important for my context. These concepts will be mentioned in my discussion when looking over my cases to find out what sort of knowledge the subjects in my experiments acquired.

Variables

The declaration, storage and use of variables are normally taught very early in a programming course. The difference between local and global values is something that can be a bit confusing for many students. The notion that variables declared inside a method can not be reached from outside is not so easily understood at first, and can lead to many confusing errors[6]. In my own experience as a tutor in introductory programming courses, many students solve this problem by making the variable global instead of passing it as a parameter to another method, something I will mention in the next sub chapter.

Methods and parameters

Jens Kaasbøll describes in [20] problems many educators perceive as difficult for students to comprehend, one of these being methods and parameters. Methods are of course well known to all who have done some programming, and can briefly be described as a chunk of code performing one or more tasks, given a name so that it can be called multiple times without having to rewrite any of the code. Some students have

[6]Here I speak from my own experience as an instructor for a group of students in a CS1 course.

problems understanding that the execution of a program can suddenly ”jump” to a method and then back and that methods can produce a result. However, when parameters are introduced I have experienced that more students face problems, especially if the name of the parameter is the same as a variable used earlier in the program. I have experienced that drawing a flowchart of the program execution can aid in explaining methods, just showing code that work will not help.

Chapter 3

Learning theory

In this chapter I will present some learning theory as a basis for observation in the experiments. There are many factors that are said to affect learning and I will here present those I see as most important for this thesis. When conducting the experiments, I will see if these factors play a part when the students learn programming.

When teaching a course, the goal of the course is important for what kind of focus the teacher choose. If the goal is scientific programming in for instance FORTRAN, the focus would be on the mathematical aspect and formulas, and if the goal is to learn Java the focus should be on topics important for Java and design of object oriented programs.

Shirley Booth mentions two main views at teaching computer programming: The traditional view, where students learn the syntax of the programming language before looking at how to put the syntax together in the semantics of the language. The second view is the spesificational, where ”the students learn to devise solutions to problems and formulate specifications from them in such a way that a program can thereby be developed” [12]. A parallel can be drawn to Procedures First (traditional) vs. Objects First (spesificational) in the teaching of OOP. As object oriented languages often have a much larger class library for use than procedural languages, e.g. the Java API, students do not need to know more advanced algorithms to program. In Java, storing, sorting and searching, for example, is implemented in the API and ready for use. An objection is ”we are not educating users, we are educating programmers”, referring to using the algorithms vs. knowing how they are built.

3.1 Learning psychology

There are two main branches when it comes to the views on how people learn[1], referred to as *behaviorism* and *constructivism*.

3.1.1 Behaviorism

From nature, people have certain reflexes based on stimulus, like for instance the Babinski reflex[2]. The Babinski reflex is an unconditioned reflex (response) and the tickling of the foot is an unconditioned stimulus.

One path in behaviorism says that people can learn a conditioned response from a conditioned stimulus through manipulation of the unconditioned response and stimulus:

> Returning to the Babinski reflex: If a doctor induced the Babinski reflex many times in a little child by tickling his foot and at the same time introducing another stimulus, for instance shaking a rattle at the same time as the foot curled, something would happen: The rattle, a conditioned stimulus, would be associated to the unconditioned response of the Babinski reflex. After many repetitions the child's foot might curl even without the tickling of the foot. The Babinski reflex would then be a conditioned response to the conditioned stimulus of the rattle.

One can say that the doctor "short circuit" the natural responses in the child's brain to react to a different type of stimulus. The Bambinski reflex is the desired response that the doctor wants, but he wants to control how this response is induced.

A different path in behaviorism says that people can learn a certain behavior through how a persons actions are rewarded, either through positive or negative reinforcement:

> A classic example of reinforcement is that of the hungry rat pressing a lever to get food: If a hungry rat presses a lever by accident and receives food, the likelihood of the rat pressing that lever again will increase. This is positive reinforcement: The subject receives positive feedback on its actions and therefore it is likely the action will be performed again. If, however the rat receives some sort of punishment when it

[1]Learning can also be explained as the gaining of knowledge

[2]The Babinski reflex is a test done on infants and it stems from our ancestor apes: When a child is tickled on the sole of his foot, the foot curls in-wards. This is from when apes instinctively grabs a tree trunk with their feet when they climb.

> pushes the lever, the likelihood of the action being performed again will decrease.

This is a different way of getting a subject to perform a desired action, by awarding the correct action and punishing the undesired action.

Yet another aspect of behaviorism is how people transfer concepts from one part of their understanding to another part.

In 1982, Pea [25] did a study on different mistakes that novice programmers make, one of them being the confusion of programming language with natural language. This is an example of *negative transfer*, where the novice tries to relate the program to natural language when attempting to understand how the language works. This can be helpful in the beginning but it will soon lead the programmer into trouble. *Positive transfer* is something that occurs when faced with a task similar to one that has been solved before. The programmer can then use this knowledge to solve this new task more easily than if she had no prior knowledge of this type of program. This [12] positive transfer will then reinforce the knowledge. Reinforcement is an important aspect of learning as this helps commit knowledge to the long term memory.

3.1.2 Constructivism

Behaviorism bases itself on that knowledge is somewhere ready made, either out there in the world or inside people, and that we get this knowledge in different ways (a couple mentioned above). In constructivism, the knowledge is not ready made, but is constructed in a person's psyche through the interaction with the environment around them. The mechanisms used to adapt this knowledge are accommodation and assimilation. [1] explains these concepts the following way:

> **Assimilation:** fit practice to theory. Complex but familiar external object are simplified to fit pre-existent categories in your head.
> **Accommodation:** fit theory to practice. You have to change the ideas in your head to fit the realities of external objects.

In constructivism there are two tools for analyzing how students learn: *Individual* and *Social*. With respect to individual constructivism, the researcher looks at how the individual constructs her knowledge through experience and interaction with the environment around her. With respect to social constructivism, the researcher looks at how individuals in a group learn.

3.1.3 Memory

There is no learning if one does not remember what one is taught. Much research exist on the field of human memory and a crucial point is that of rehearsal as an aid to memory. Students who review subjects they are taught remember what was taught to them much better than students who do not review[3]. Allowing students to have time to review old material before starting on something new is important for a good learning curve.

3.2 Methods of teaching how to program

There are many different methods that are said to have a positive effect on learning and I will consider a few here.

3.2.1 Pair programming

Pair programming is exactly what it sounds like: Two people sit together and work with a program on one computer. The usual dynamics[4] of a pair programming team works thus: One person sits and program, and one person sits and keeps an overview of the progress, giving instructions and correcting errors as they pop up. Both members of a pair programming team participate in the discussion and development, but they have different responsibilities that circulate among them (one writes for an hour while the other inspects and then vice versa).

In [23], the effects of Pair Programming are well spoken of: Better quality of code, fewer bugs, faster production and better use of the time. Many interesting points are discussed in this article, but there is especially one thing in the article that is of particular relevance to this thesis. That point is that pair programming is a good chance for the members to learn by discussing the problems at hand and drawing on each other's experience, as an example of social constructivism mentioned above.

A problem with pair programming can be if one person is more skilled than the other and does not take well to corrections. It is important that both members of the pair programming team to be motivated for the working style.

An argument against pair programming is that it is very easy for one person to become a passenger on the project while the other person does everything. A way to avoid this problem is that both parts of the team should write parts of the time to ensure that both are involved.

[3]http://brain.web-us.com/memory/memory_and_related_learning_prin.htm

[4]I say usual dynamics because this is how pair programming was meant to work, but that is not always the case.

3.2.2 Reading, testing and running code

Zeller [29] reports in his article on the use of a system called the Praktomat to make students read and review other students' code. 61.5% of the students reported that it helped reading and reviewing other students' programs and another 19.2% were partially pleased. 63.5% of the students said it helped that others reviewed their code and another 13.5% were partially pleased.

At the same time, grades went up for students who read many other students' code. The grade also went up for students who received reviews on their own code[5].

The above mentioned article gives good evidence that the students learned from reading other student's code. However, the good grades might be just because students who review more code are more fascinated with programming and have a better motivation for learning and working more. Of this, the article says nothing. In this article the students were able to validate code also by having tests run on it. It is important to understand that a program is first correct when it behaves the way it is supposed to.

Zeller shows us that looking at code examples will be of good help, but not just as reading; the students must be able to test the code they read as well as to see it performing on the computer screen.

When a student runs code, she will get a good chance at seeing exactly what her code has done. Then, by doing a small change in the code and re-run it, she will see how changes affect the execution of the program. This is a way of learning how programs works and gives hands-on experience that is invaluable in learning to program.

It is important that the code produce comprehensive output so that changes to the code will be visible outside the computer. Complicated algorithms with much calculation and little result will be hard to understand just from its output. Look at this simple example, illustrating computation that takes place over more than one line of code:

Listing 3.1: Program with a simple output.

```java
/**
 * Simple calculation program that prints out a
 * result of a computation.
 */
class Calculate {
  public static void main(String [] args) {
    final int MAX = 15;
    int result = 3;
    String [] prints = new String[MAX];
```

[5]This was on average, there were some who still got a lower grade even though they reviewed many programs and got many reviews

```
10      for(int i = 0; i < MAX; i++) {
11        result = result++;
12        prints[i] = "Result " + i + ": " + result++;
13      }
14      // Print the string at place 5 in the array.
15      System.out.println(prints[5]);
16    }
17  }
```

Try changing line 11 to

result = ++result

and you will get a completely different answer. Since *result* is calculated in two places (line 11 and 12) it is not apparent what has happened with the calculation of *result*[6].

In behaviorism, positive reinforcement is seen as an important aspect when learning. The example above (3.1) does not give a very helpful feedback, just a different answer and does not help to reinforce the understanding of pre-increment.

3.2.3 Playing a game

One view on learning how to program by playing a game is that it can be very inspiring and motivating to play around with graphics and many moving objects. In many games there are also the aspect of competition to encourage students to ”play” harder. A question that arises is: *Can the competition introduced by some of these games work as an added stress factor for the students and in this way have a negative effect on learning?* Through some communication with a student in the USA, I learned that the course instructor there used Robocode in his course and that the grade of the students depended on how many battles they won in contest with other students. Will this solely work as a motivator or is it over the top? It certainly encourages students to do well but will many feel too much pressure and leave the course? Another objection to the use of games can be that game-playing might overshadow the reasons for using the game in the first place. Winning the game can be more important than actually learning. Still, students will learn from this experience as well, even if it is only how to win a game.

(Amy Bruckman: MOOSE Crossing) The MOOSE Crossing offers a simple scripting language a bit more advanced than LOGO to create different objects in a text based world. In her doctoral thesis (ref) Bruckman describes the following experience in the MOOSE Crossing project:

[6]By changing the order of the ++ notation, java computes in a different order. When ++ is written before the result (called a pre-increment), result is incremented twice at line 11.

One Friday afternoon in April of 1996, I accepted the MOOSE Crossing application of a new member, a twelve-year-old girl who chose the character name Storm, and then I left town for the weekend. (...) When I returned on Sunday, I was surprised to learn that Storm now knew the basics of how to program. She had had limited previous experience she had once tried Logo in school. Over the weekend, she made Jasper (a frog you can hug),(...) collaborated with Rachael (girl, age 13) (...) Storm and Rachael spent most of the weekend together, talking and helping one another with their projects.

This is a clear statement on the positive effects games can have on learning how to program.

3.3 Structure of coursework

When structuring a beginner's course in computer programming there are many factors to take into consideration. If the students are faced with problems they have no qualification for knowing or understanding, learning will be hampered.

In this section, I would like to write a little about my personal experience with observations of a course where I study (the University of Oslo). What I say about students behavior and understandings and misunderstanding below, I draw from my own experience as an instructor for a group of students in an introductory Java course and an article submitted to NIK (Norwegian Conference on Informatics) this year [8]:

Starting out with the **HelloWorld** example mentioned earlier (2.1), students in a CS1 course are shown their first Java program. This is not a program illustrating a good way to use Java or showing Java's strengths, it is the minimal amount of code needed to successfully compile and run a Java program that produces output. No experienced programmer would ever use Java to write out a text string. The students are shown the wrong tool for the problem. But students have no problem understanding and replicating this type of code as long as they are told that the words class, public, static, void, String are "magical" words needed to make the whole thing work.

The next steps from this example is more programming in *main* using variables and most likely some custom made input/output class[7] and then imperative programming (the use of if, for and while) and control structures. These are all concepts that students learn easily as soon

[7]Input and output in Java is not very easy to understand right away and there are many things to learn at the same time to use this way of input and output. The solution is therefore that instructors make easy-to-use input and output classes.

as they are comfortable maneuvering inside the *main* method. The way up to this point has been a procedural approach and the next logical steps will be arrays and methods. Methods of course being ”magically” static. Methods are a bit harder to explain and the students spend some time learning that and the difference between local and global variables. Thereafter the first OO problem arises. The problem may not look exactly like this, but it is most likely closely related:

The *String* variable in Java is, in a Procedural First approach, treated as a primitive variable such as *int* and *char* in that it is assigned a value as a primitive (ex: String temp = ”Hello”). When using a Procedural First approach, arrays and if-tests come before classes and objects. The following code or similar might pop up in an assignment:

Listing 3.2: A student register with names.

```
1  class Register {
2    public static void main(String [] args) {
3      // Array that can contain 100 strings
4      String [] students = new String[100];
5      // Code to register a couple of students
6      // omitted...
7
8      // Serch for a student:
9      String find = "John Doe";
10     for(int i = 0; i < students.length; i++) {
11       if(students[i] == find) {
12         System.out.println("Found " + find);
13         break;
14       }
15     }
16   }
17 }
```

The first error in this program is found on line 11. Strings can not be compared in this way. The correct way to write this comparison is:

if(students[i].equals(find))

Students may be confused and not understand why the comparison has to made in this way. The students might see this as a more cumbersome way of comparing two strings, when it was so easy with *int* and *char*. Can they do this with the *int* and *char* too? Of course they can’t, so they wonder why, but it is possible to convince the students that ”this is the way it is done”.

Now a second problem arises: If the array is not completely full, line 11 will after the previous correction result in a *NullPointerException*[8].

[8]An error in Java, that occurs when the program tries to use a method or variable in an object that is not yet instantiated and is therefore null.

What is that? Why do I get that? What is a pointer? How do I avoid it? It compiled OK, why does this happen? Explaining the concept of $null$ and that it is illegal to point at something that is null will be met by nods and smiles, but the students are confused. This is an example of problems one is faced with when doing a Procedural First approach, and there are many more examples. About this time, classes and objects show their ugly faces. I say ugly faces because this is how many students perceive it. ”This is not proper Java, we have already learned how to program, why are you showing us this?” It is not very satisfying to be told that ”this is the proper way, what you have learned up to now is not the correct way.” At the same time as the appearance of classes and objects, assignments grow fast and the students, not used to structuring[9] will be stuck.

When teaching OOP, the structure is important, and the structure is based on the approach chosen and the approach chosen should be based on the goal of the course.

3.4 Approaches to teaching OOP

As we have seen, there are two different approaches to teach OOP to students: Objects First and Procedures First. Procedures First has long been the leading approach although there is a trend to wards more OO theory earlier and earlier in OO courses.

3.4.1 Procedures First

Procedures First is described in the example above.

An advantage with this approach is that the programs can be very small and be expanded one line at the time. A problem with this approach, is that the shift from procedural to object oriented can be hard. Bergin writes that experienced programmers can spend as much as 18 months making the shift from procedural to object oriented thinking (see below).

I am not saying that Procedures First is a bad approach all in all, it is the only approach if the goal of the course is to teach a procedural language. However, there are examples that a procedural approach doesn't work [10]:

> *The experience of the industry is that an experienced procedural programmer will take a year to 18 months to make the*

[9]I will go as far as saying they can not structure at all, an example of this is that the students when asked to hand in UML diagrams with their assignments, draw the diagrams after the code is written, not before. It may sound harsh, but they have totally missed the point.

> *switch (from procedural to OO). (...) If you do a really excellent job of teaching them (students) to think like a procedural programmer, they will face this 12 to 18 month paradigm shift. I don't know where to put this year of confusion in a four-year educational program.*

3.4.2 Objects first

Starting off by introducing objects right away is an approach that offers a solution to the problem mentioned above. There are different ways of introducing objects to begin with, [13] lists some of these:

- Modeling the world: Illustrate objects as models of the real world.
- Implementing one domain in another: Illustrate that objects in a program can be used in other settings as well.
- Active objects: Show that objects can communicate with each other and affect each other.
- Introducing object oriented concepts: Introduce object oriented concepts as early as possible in a course.

An advantage of using the Objects First approach is that the students get used to the OO way of thinking from the start and therefore will not have problems making the shift from procedural to OOP. In OOP the students learn to put methods into classes and therefore learn procedural programming in the objects.

A challenge is finding a first example advanced enough to illustrate OOP, yet simple enough for students to understand.

3.5 Putting it all together

"I know how to code all the small bits, but where do I put them?" A problem when lecturing using slides or similar is that there is limited space on the foil and large programs are hard to show. One of the main problems for most students is to see "the big picture", putting all their little code-bits together. When a program becomes too large, it is hard to keep track of all the information that flows and wherein the code things happen. Pea ([25]) reports on this as one of the high-frequency "bugs" that students make.

Here OOP has an advantage: One can split up the program in many short files and responsibilities and this might help keeping track of all the code.

During the period from March to May in 2003, I was part of observing an introductory course in OOP [8]. During this time we observed many common problems for the students, for instance

- translating their mental model[10] of the computer system into code. The students knew what to do but not where in the code or how;
- being able to make a class diagram[11] of the system prior to coding, these diagrams were made after the coding was done;
- seeing the data flow. The students had very good knowledge of imperative topics (variables, loops) but lacked knowledge about the overall flow.

Some of these problems can be contributed to the fact that students have to relate to a very complex world (figure 3.1). We see in this figure that there are five different domains that the students must relate to. These domains are all a part of the computer program and relate to eachother. Without training, students will have problems knowing where to put their focus and in what order to concentrate on these. For instance, the ”permanent data file” is important when the functions for reading and writing data are written but not while making the data structure. In addition to having 5 domains to relate to, we see from the figure that there are 10 relations between these domains to relate to.

Being able to simplify this world for the students and limiting the areas of attention the students must focus on (figure 3.2). In this figure, the ”program execution” domain, the ”permanent data file” domain and the ”real world domain” have been removed. What we are left with are the ”code” and the ”screen display and typing”. Both these domains are visible to the students.

This simplification can be achieved by the use of a graphical environment. My experiments will hopefully support this claim.

[10]A mental model here means that the students knew for instance that they needed to store customers in a register, type in their information and store them to file.

[11]With class diagram, I mean a UML diagram showing the different classes in the computer system

Figure 3.1 The complex world the students must relate to

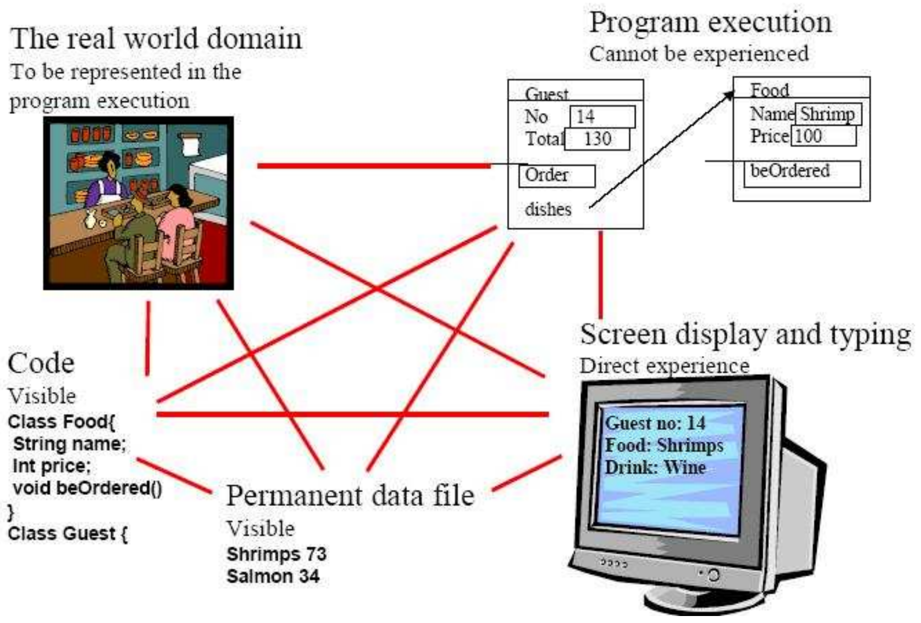
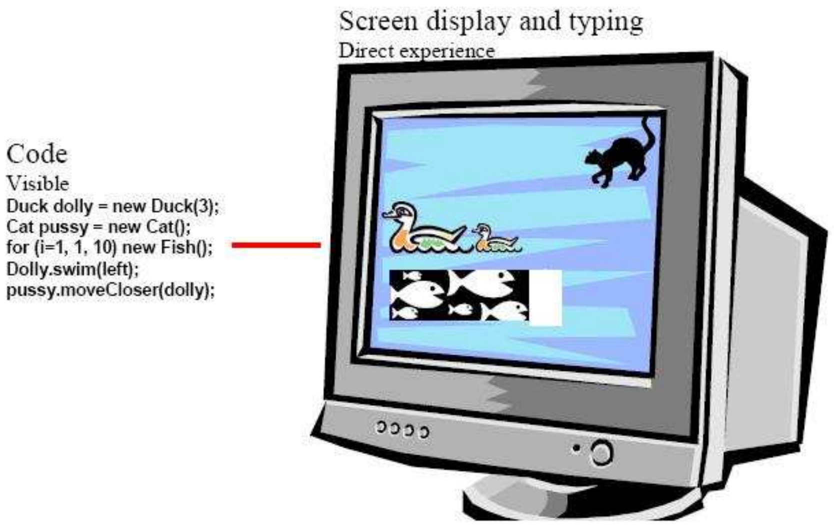


Figure 3.2 A simpler world



Chapter 4

Graphical environments

As announced in the introduction, my main research question relates to the use of different graphical environments to help teaching Object Oriented programming, and more specifically Java, using an Objects First approach.

By graphical environments I mean the use of a program or environment of some sort that represents 'a different reality' than the one the students are used to.

The term graphical environment should not be confused with the use of common pedagogical examples such as making a student register system or airport system. This different reality will be something other than just a text editor and a terminal where output is seen. There are different ways of representing this graphical environment. This can be done by both graphical output on screen done by predefined packages imported into the programs (examples discussed later) or something completely removed from the computer screen, e.g. LEGO Mindstorms.

4.1 Logo, where it started

The use of graphical programming environments is not a new idea. In the 1960's, Seymour Papert, Daniel Bobrow and Wallace Feurzieg developed a language called LOGO. This program was based on giving simple commands to a little mechanical robot called the Turtle. The image below (figure 4.1) shows two children playing with this turtle. The turtle was equipped with a pencil and with this the turtle could draw simple drawings. As seen in the picture, the turtle was hooked up to a computer via a cable where the children could input different commands. An example of a command would be FORWARD 50, resulting in the turtle moving 50 units (where units are predefined) forward. Children used the computer to talk to the turtle, and they would crawl around on the floor acting like turtles in order to envision how the turtle would respond to certain commands. Papert viewed the turtle as an object to think with thus giving the children a very visual and real link to the computer language [24].

Figure 4.1 Early stages of LOGO

The newer versions of the LOGO programming language that were moved to the computer screen instead of the mechanical turtle is a simple place to start if you are being introduced to a structured and procedural language. However, as an introduction to Object Oriented Programming the LOGO language falls short. The concept of an object or series of objects will not be apparent in LOGO as it focuses on giving commands to a single turtle that is never in any way instantiated as an object. A program in LOGO's spirit focused on OOP has been tried in a CS1 course learning Java by Stephen Schaub. The author reports that it has lead to more students completing the course [26]. What was done was that Schaub made a graphical package for the students to import. Drawing the letter **T** with this package would then look like this in a Java program:

Listing 4.1: Turtle eksample

```
import cps110.turtle.*; // Import the package

class DrawT {
  public static void main(String [] args) {
    Turtle t = new Turtle("Herb"); // Make a new turtle
    // Draw the T
    t.pendown();
    t.forward(30);
    t.right(90);
    t.forward(10);
    t.backward(20);
  }

}
```

Schaub also made other packages with graphics, and as mentioned, he writes that more students completed the course when using these packages. From this article [26] it is apparant that a graphical programming environment was a motivating factor and helped the students complete the course. The article says nothing about whether or not the avarage grade improved. Looking at the example above, one sees that this is a step in the right direction of introducing Object Oriented concepts, as the student makes a new turtle object which is represented in a window and it is clear that the object has certain methods connected with it. But I miss a more thorough exploration of all the possibilities that this package might introduce, such as making more turtles, drawing complex shapes and maybe illustrating algorithms such as trees with a drawing.

4.2 OO environments

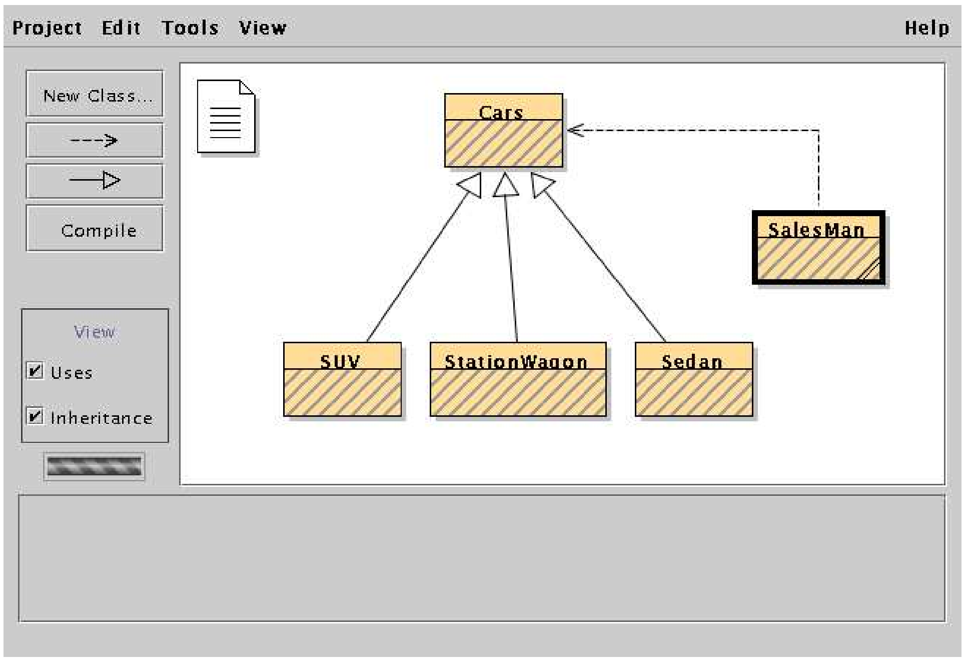
There are three environments that have stood out in my search for Object Oriented programming environments. One is Karel J, another is Robocode, and a third is BlueJ. I will briefly touch upon BlueJ before going more deeply into the use of Robocode and Karel J. These three environments have been frequently mentioned in the literature and are also being actively updated and used. Another reason for looking at these three environments is that Karel J, BlueJ and Robocode all represent different styles of presentation, different methods of teaching and different focuses on what is important. Although different in these ways, they are all built on the same philosophy that a visual representation is important.

4.2.1 BlueJ

BlueJ stems from an older language called Blue, developed in 1996 by Michael Kölling and John Rosenberg to teach OOP to CS1 students [21]. This language was was built on C++. BlueJ was developed in 2001 by the same people and was purely based on Java [22].

BlueJ separates itself from the other two environments first of all because it is not a game, and second of all because one does not have a framework of code where one can change a few lines and see the result in form of movement. In BlueJ, the classes modelled in a program are shown in a UML-style[1] window (figure 4.2).

Figure 4.2 BlueJ: Example of a car dealership



In the figure we see the classes marked in yellow boxes and the arrows marks the relationships between them. Solid arrows denotes an ”inheritance” relationship and the final arrow denotes a ”uses” relationship. Clicking on these boxes will open an editor that displays the code for the specific class where it is possible to edit the code.

When running the program, the student can see the objects generated during runtime and also the execution of the code itself. The student has the aopportunity to inspect the objects, check their state, add new data and so on. The classes can also be opened in an editor where it is possible to edit the code, recompile and run the program again.

BlueJ does not put the emphasis on the program as something running in a computer, but as a model of a real world domain. Students are able to see the classes that are needed to model the problem domain and the different objects that are generated from these classes.

A positive thing with BlueJ is that it offers a complete tool where it is possible to design, edit code, comment using Javadoc, execute and

[1]UML - Unified Modelling Language. A language used to describe a system by, among other things, modelling its classes.

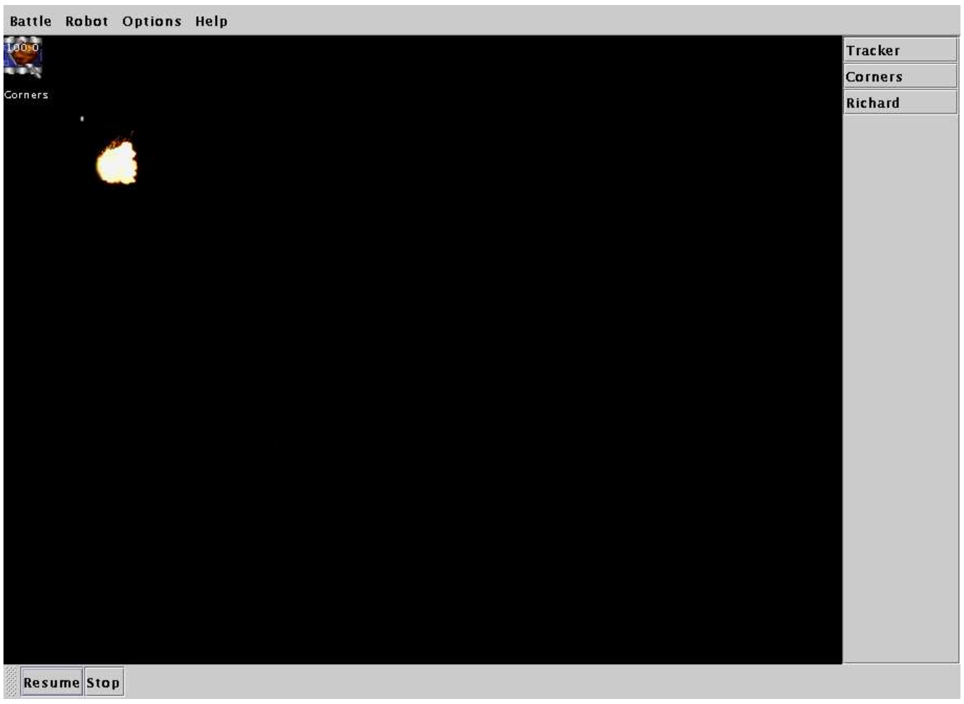
debug the program. BlueJ should not, however, be confused with a UML development tool.

BlueJ has been tested in many courses world wide[2]. In addition there exist a textbook on BlueJ [3], a website with discussion forums[3] and many articles on the use of BlueJ, however there exist few articles on experiences with using the tool, one article reffered to in chapter 1 [5]. BlueJ is therefore a well documented tool and its use in many courses is an indication that BlueJ works.

4.2.2 Robocode

Robocode is an old-style fighting game developed by Mat Nelson at IBM alphaWorks. The rules of the game are very simple and illustrated in figure 4.3:

Figure 4.3 One tank destroyed in the battle



1: A number of tanks battle it out in a square arena.
2: In order for one tank or team of tanks to win, all opponents must be destroyed.

[2]Browsing courses on different Universities all over the world gives clear evidence of this.

[3]http://www.bluej.org

So far it all seems straight forward, but when one looks more closely at how the tanks are controlled, Robocode gets interesting. Every tank in the battlefield is a separate Java class written to dictate the behaviour of the tank. When it is time to start programming a tank, Robocode has a predefined API where it is possible to find many simple methods for controlling the tank. In addition, when programming a new tank, Robocode will set up the basic framework so it is possible to start right away with entering commands and method calls without having to learn many standard Java phrases[4]. These phrases can instead be learned when the need or curiosity arises. In this way, the student can learn the phrase needed and at the same time see how and where it is used.

Robocode comes with its own editor where it is possible to write and compile the code for the tank (figure 4.4).

Looking at the code in figure 4.4, one sees a distinct resemblance to Seymour Papert's LOGO: For instance, the command $ahead(100)$ will give the same result as the FORWARD command in LOGO. In a way, the tanks in Robocode can in the same way as the turtle in LOGO capture the imagination of the student programming but on a more imaginary level. In the same way as a child would crawl around on the floor acting like the turtle drawing [24], the student using Robocode can imagine sitting in a tank controlling the gun. It might seem like a far fetched idea when thinking that imitating a turtle is quite easy as everyone has seen or played with a turtle up close, but not many people have seen a tank in real life and even fewer have sat in one. But when considering the amount of computer games that are out on the market it is not as hard to imagine driving a tank and fighting other tanks as it might seem.

At a first glance, Robocode does not give a very object oriented feel. One can see the $class$ modifier and $extends$ and $import$, but other than that there are mostly a series of methods and a few $while$ loops and if tests. What Robocode does have however, is a set of methods that directly and visually affect the tank itself, giving students a very interactive first glance at method calls with parameters.

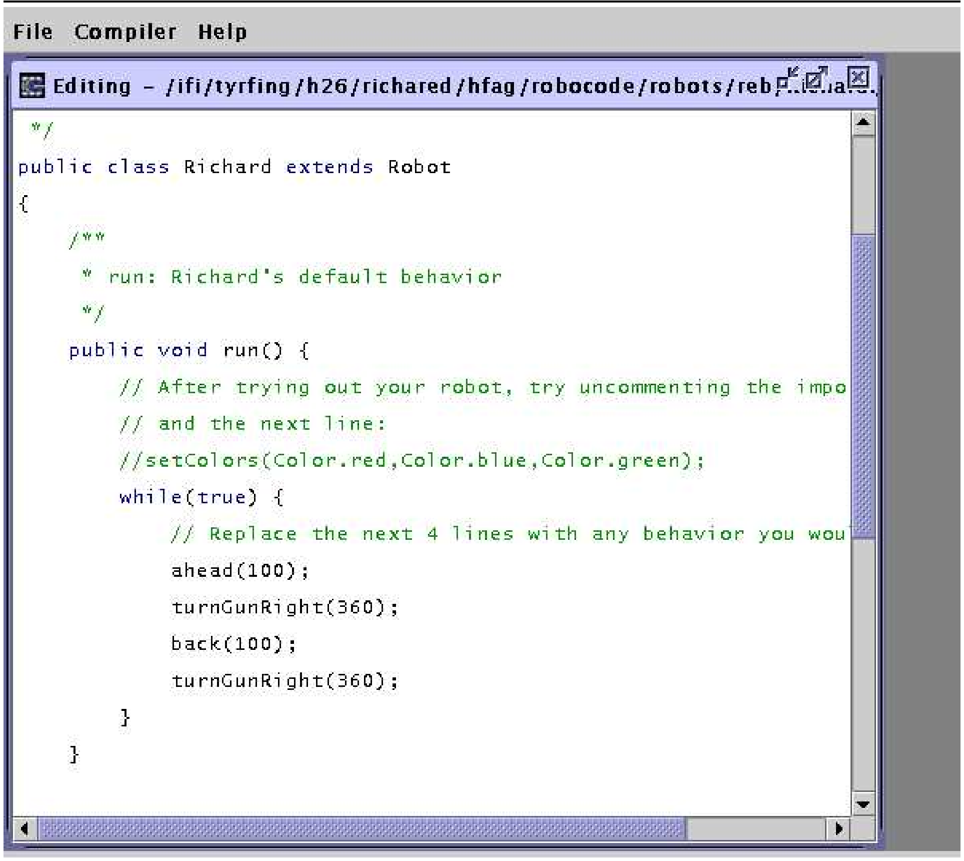
Robocode is a very visual and engaging game that quickly can spark interest in students as it introduces both a nice interface and an element of competition. The argument against the use of Robocode can be that the high game factor easily can overshadow the fact that there is supposed to be an element of learning involved.

Robocode has developed quite a big society on the Internet where people upload their own robots and compete with others[5]. It is interesting to see how the discussion board is a good example of an online learning community. As far as documented experience on the use of

[4]Standard Java phrases can be *class, import, public static*

[5]http://www.alphaworks.ibm.com/forum/robocode.nsf/current?OpenView&Start=1&Count=100

Figure 4.4 Robocode editor


File Compiler Help
Editing - /ifi/tyrfing/h26/richared/hfag/robocode/robots/reb

```
 */
public class Richard extends Robot
{
    /**
     * run: Richard's default behavior
     */
    public void run() {
        // After trying out your robot, try uncommenting the impo
        // and the next line:
        //setColors(Color.red,Color.blue,Color.green);
        while(true) {
            // Replace the next 4 lines with any behavior you wou
            ahead(100);
            turnGunRight(360);
            back(100);
            turnGunRight(360);
        }
    }
```

Robocode there exist little. In a didactics project at the University of Oslo, three students tested the Robocode environment on 14-16 year old boys with an interest in computing. After about 4 hours of training the students were able to make methods with parameters [6].

I was in contact with a student whose class was given Robocode as a project (Rochester University). The student's grade depended on how many points they accumulated in a 10 round battle. The students were free to copy and modify code, but they had to explain their code. This is an example of how Robocode can be used as a project for teaching. Although I think making course marks depend on a contest is a bit extreme in the sense that the contest to poorly reflect the quality of the students' understanding of the code, it is a way to motivate students to work with code.

4.2.3 Karel J

Karel J is, like Robocode a game-like environment. However, Karel J separates itself from Robocode in a few ways. First of all, Karel J is not a very action filled game and second of all, Karel J is more designed with the thought of being used in an educational setting.

Karel was developed by Richard Pattis and was first used to teach Pascal to programming students (1981), but was later rewritten by Joseph Bergin, Mark Stehlik, Jim Roberts and Richard Pattis to teach C++ [6]. In 2001 Karel++ (the name of the C++ version of Karel) was rewritten completely in Java by Joseph Bergin, and Karel J was born.

Karel is a little two dimensional robot that lives in a two dimensional world. The world is built as a grid (figure 4.5) with streets and avenues. In addition the streets and avenues can be blocked by walls. In this world Karel works and does small tasks. These tasks involve either picking up or placing out beepers.

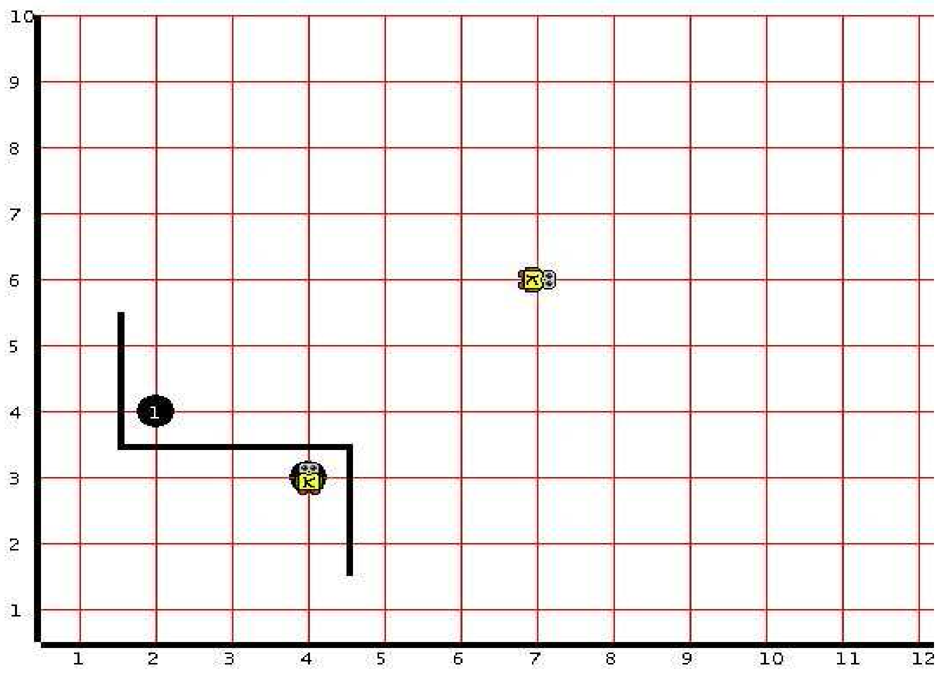
Being a robot, Karel must be told everything he has to do. This is the job of the programmer. The tasks given to Karel are solved when the student writes a program that compiles and get executed correctly[7]. One might say that having to build an entire program right away might be difficult for a beginner student. However, there is no problem for the lecturer to hand out a program skeleton in the lectures and ask the students only to fill in the blanks.

Karel J shows one of its strengths over Robocode in that it is more flexible. By this I mean that Karel J does not set many boundries on the programs. In Robocode the students find themselves in a particular setting and have to follow the rules set up by the game. Karel J has

[6]INF-DID course, 2002. Bording, Kyrvestad and Pedersen

[7]A correct program means that Karel solves his task

Figure 4.5 Karel's world with walls and beepers



no rules, the students are free to solve the tasks any way they choose, unless told otherwise by their instructors.

Karel J can be made to be very easy or very complicated. It is possible to illustrate everything from making objects to recursion and threads. The reason this works so well is that the world Karel lives in is very free when it comes to what can be done. Karel has only a few basic methods, the rest must be written by the student. The Karel robot class is like any other Java class and it is therefore possible to make multiple objects of it, make subclasses, override methods etc.

Karel J was designed for teaching OOP and work as a pedagogical environment whereas Robocode was designed as a game. I feel that both robots and tanks are suitable for illustrating objects as they are both mechanical things with a fixed set of instructions or methods.

I have focused much of my attention on Karel J because I feel that it has a strength over the other two environments presented. As mentioned above I feel that Karel J is much more free and more focused on the educational setting than Robocode. At the same time it is more illustrative than BlueJ. In the next section I will make a small comparison of the three based on what concepts they introduce and how they would then typically present these concepts.

4.2.4 BlueJ vs. Robocode vs. Karel J

Specifics	**BlueJ**	**Robocode**	**Karel J**
Editor	In application	In application	Uses texteditor
Compiling	In application	In application	Uses terminal

Concepts	**BlueJ**	**Robocode**	**Karel J**
Classes	Shown in UML style	Coded by the students in Java programs	Coded by the students in Java programs
Objects of classes	Shown when the program is run	Only has subclasses of abstract classes and objects are graphically created	Coded in normal style and created with **new**, shown as robots.
Methods	Shown when the program is run, depending on code	Shown as behaviour in the tanks	Shown as behaviour in robots when the program runs
Inheritance	Shown in UML style	Subclasses of abstract classes	Possible to make subclasses, it is encouraged

As mentioned in the beginning of this chapter, BlueJ, Robocode and Karel J all have different approaches to OOP.

BlueJ focuses on modelling the world and showing objects and classes as models of the reality. The students learn to relate to programs as objects and classes that models the part of the reality that the program is used in.

Robocode focuses on displaying objects as separate entities that can interact and communicate as *active objects*. In these objects the students program behaviour using imperative programming thus learning the more technical side of programming, but locked in classes and objects. As seen in the example below:

Listing 4.2: Example Robocode

```java
/*
 * Eksample of a tank programmed in Robocode.
 */
class Example extends Robot {

    // Main run method
    public void run() {
        // Controll structure
        while(true) {
            turnGunRight(180);
            forward(100);
            backward(100);
        }
    }

    // Method called when a tank is scanned.
    // ScannedRobotEvent points at the scanned tank object
    void onScannedRobot(ScannedRobotEvent e) {
        // Determine what to do based on the object's data
        if (e.getDistance() > 200) { fire(1); }
        else if (e.getDistance() > 100) { fire(2); }
        else { fire(3); }

    }

}
```

Karel J focuses on introducing OO concepts by using a visual aid so that students have something to relate the concepts to. Karel J is very small and simple so that students start off programming simple Java programs but with the visual aid of a small robot.

Listing 4.3: Example of Karel J

```
package kareltherobot;

class Karel implements Directions {
  public static void main( String [] args ) {
    // Make a new robot called karel
    ur_Robot karel = new ur_Robot(1, 4, North, 0);
    // Instruct the robot
    karel.move(); // Move one step
    karel.move();
    karel.turnLeft(); // Turn 90 degrees
    karel.pickBeeper(); // Pick up a beeper
    karel.turnOff();
  }
  // Load the correct map
  static {
    World.readWorld( "eks6-1.kwld" );
    World.setVisible( true );
  }
}
```

What these three environments have in common is that they all use an Objects First approach to learning Java.

Summary

Before I start writing about the experiments I have conducted, I will give a short summary of the theory I have introduced so far and highlight the important aspects with respect to this thesis.

What sort of OO concepts that the students learn and how easily they learn them are the main issues for studying in the experiments. I will be looking for the concepts introduced in chapter 3, namely van der Linden's four main concepts **abstraction**, **encapsulation**, **inheritance** and **polymorphism**, and furthermore the concepts **active objects**, **modeling the world**, **constructors** and **methods**. For each concept, I will also try to find out what the students find as hard.

When it comes to the learning part, there are many aspects I will be looking for. My hope is to see that the environments used in the experiments will help stimulate the methods noted in chapter 3.

In the more theoretical field I will be looking for signs of constructivistic and behavioristic learning. When analyzing my experiments with respect to how students learn I will be looking through both the ”individual lens” and the ”social lens”.

With respect to the more specific computer aids, I will be looking for the use of **pair programming**, **reading of code**, **testing**, **debugging**, **visualization**, **tools** and the use of the **Objects First** approach.

I hope to observe as many aids as possible in the experiments I will conduct. I will not claim that the experiments were a bigger success if the students used all the aids rather than if only a few aids were used. However, I will use these aids and how they were used as some sort of guideline in my discussion in 7 and 8.

Chapter 5

Methods

I will now look at various methods for collecting empirical data and justify the methods I have chosen.

My main research question is: ”In an Objects First approach to learning OOP and specifically Java, how does the use of graphical environments influence learning?” To try to answer this, I will be conducting two experiments on students using two different environments. In one experiment I will only observe how the students use the environment without any guidelines from me, while in the other experiment the students will be given tasks to solve and also taught some theory. This will of course influence my choice of methods.

Before I start looking into the different methods, I will briefly present my experiments so it will be easier to relate to my choice of methods. The experiments will be discussed in depth in the next chapter.

In the first experiment, I will observe how students use an environment called Robocode. Through conversations with the students, I will try to get an overview of how they use the environment and if the environment stimulates understanding of any OO concepts.

The second experiment will be focused on the use of an environment called Karel J. In this experiment a course will be tested on a group of students and this will all be put on videotape. We will be filming pairs of students to get their conversations and behavior recorded. Conversations with the students is also important to gather information.

When choosing methods for collecting empirical data, one can either choose to use *Qualitative methods* or *Quantitative methods* or both.

5.1 Qualitative methods

One formal definition of qualitative research methods is given in [2, page 5]:

> *Qualitative research methods were originally developed in the social sciences to enable researchers to study social and cultural phenomena. (...) Qualitative data sources include observation and participant observation (fieldwork), interviews and questionnaires, documents and texts, and the researcher's impressions and reactions.*

The data given in qualitative research is often coming from a rather small population of research subjects. On the other hand, the data is more thorough and in-depth.

In an Information Systems (IS) course, and in my case: Informatics Didactics, qualitative methods seem like the way to go. Both qualitative and quantitative methods can be applied to the study of social and human aspects to a problem, but in a qualitative approach, each subject in the study is observed more closely. This is why I have chosen to use qualitative methods in my research as I am interested in getting a deeper understanding of how different individuals react to the environments I introduce to them. For me, I feel it is better to get a solid picture of maybe 10 different people and their experiences rather than a rating from 1-10 of the environment from 400 people where little is told about the environment itself or how the individual react to and interact with the environment.

I will now look at some different qualitative research methods.

5.1.1 Case study research

The definitions of case study research are many, and there is no specific definition that is better than the others. In [7, page 81], we find the following definition.

> *A case study examines a phenomenon in its natural setting, employing multiple methods of data collection to gather information from one or a few entities (people, groups or organizations). The boundaries of the phenomenon are not clearly evident at the outset of the research and no experimental control or manipulation is used.*

Case studies are very often used in IS as this is a way to investigate a phenomenon within its real-life context. Information can be collected by many different means, discussed in more detail below. The data collected are focused on the phenomenon (case) you want to investigate closer. Case studies are useful for answering the ”why-s” and ”how-s” of a phenomenon. The questions are decided beforehand and the setting is also decided upon.

5.1.2 Experiment research

Experiment research differs from case study research in that in an experiment, the terms are set by the researcher. The researcher creates a phenomenon that she wishes to study instead of looking at a natural phenomenon.

I will be conducting two experiments, having made an educational setting where coursework partly designed by me will be tested.

In both case study research and experiment research, the researcher must decide on several issues: Unit selection, site selection, data collection methods and data analysis.

Unit selection

The researcher should decide on who the study should focus on. Should the researcher focus on individuals, groups or parts of an organization? The selection will of course depend on the goal of the research and the case: If the goal is to test an application used by individuals independent of each other, one selects individuals for the experiment to get non-influenced feedback from the user[1].

If the goal is to test an application used for teamwork or cooperation between people in a group, then one chooses to focus on groups for research. This is because as a group they must all evaluate the application. Similarly, if the goal is to evaluate an application used in an organization as a whole, then one looks at parts of the organization and then puts these parts together for a bigger view.

In my first experiment I will focus on individuals and in the second experiment I will use pairs of students working together. This is because the environments I am considering are used to enhance learning experience for the individual students, but I am also interested in seeing how well one of the environments encourage discussion and cooperation. I have chosen to look at a two experiments and those will be the use of graphical environments used by students that are connected to an introductory course in programming, whether they are just starting or are nearly completed.

[1]Research has shown that an individual’s answer to a specific question will be influenced by other individuals in for example group interviews or teamwork.

Site selection

The choice of site where the case study or experiment is to be conducted must be considered well. It should be a place that fits well with the goal of the research. For instance, if the goal is to examine some sort of application used by a particular branch of business, e.g. law firm, software company, insurance etc., the site should be a place of that particular branch and ideally some place where a new system is needed.

My choice of site comes very naturally. In order to look at students using a programming environment I must choose a place where the students usually work. I therefore choose to observe students while they are working in front of a computer at the University. This is because this is the place students usually combine with programming, assignments and learning. I want to see what sort of stimulation the different graphical environments offer in this type of setting.

Data collection

There are many ways of collecting data: *Active and passive observation, interviews, questionnaires and documentation.*

In active and passive observation, the researcher gets his subjects to work with either the application or method that is the focus of the research. The researcher then observes how the subjects are using the application/method, either by watching quietly and letting the subjects work it out for themselves (passive) or by working with the subjects (active).

Demonstration as a part of observation is a good way to get hands-on experience of the usage of an application/method, and can usually reveal many strengths or weaknesses that the researcher never found while testing the application/method. Later interviews with the subjects will also be easier as the researcher has had the opportunity to learn more about the interview subjects and the situation from their point of view. A problem with demonstration can often be setting it up. The equipment used (video cameras, computers etc.) must work properly, the experiment must fit into everyone's time-schedule and there must be rooms available. This means a lot of work on the researcher's part and there is always a chance the demonstration might fail due to unforeseen events.

In one of my experiments, video recording will be used. This is because I will play an active part in this experiment, helping the students with their assignments, and it is impossible for me to collect data at the same time. Video will therefore be a helpful aid when going over the experiment later.

Interviewing gives the researcher the opportunity to have a closer talk with the subjects in the research. The researcher is free to structure the interview the way she likes: Either structured with many predetermined questions where the interviewer sticks to the questions, or an unstructured interview. In the latter case, the interviewer has put down some general questions and let the conversation circulate around these, with the opportunity to stray from the questions. In some interviews it is better to use a structured interview, and sometimes an unstructured interview gives the best result.

An example of when to use a structured interview is interviewing someone where the interviewer is on a time schedule. There is often not much time for unstructured talk. An example of when to use an unstructured interview is when the interviewer wishes to get some information that might be hard to get if he/she comes off as too professional.

In my experiments, I will be walking around having unstructured talks with the students about the taske they are undertaking, answering questions and guiding. This will help me to get an understanding of the situation of the students. Through these unstructured talks, I will also try to get some information about the students' programming background.

Questionnaires are a good way to collect data that one wishes to represent graphically (i.e. pie chart or bars). With a questionnaire it is possible to gather data from many research subjects at the same time. There are however negative sides to questionnaires: The questions must be well prepared so that the people answering the questionnaire understand exactly what the researcher wants to find out. In questionnaires with multiple choice answers the answers must fit well with the question. Questions on the form *On a scale from one to ten, how well do you like the environment?* is less interesting for me than to see the environment in action. It is also quite easy for a subject not to answer the questionnaire unless he/she is encouraged. It is easy to feel one can get away from the extra work by just staying invisible in the crowd.

One good thing about questionnaires however, is that they can be easier for a subject to answer truthfully on the questions as they (the subjects) are anonymous to the researcher analyzing the material. In my research I will not use questionnaires. The main reason for this is that I will not get a large enough population of subjects to justify the quantified data that will come out of the questionnaire. In addition, questionaires takes along time to make and evaluate.

Documentation is different kinds of written material like formal reports or newspaper clippings. Documentation can later be used in the research either as direct quotes in interviews (for example: *But sir, the report from last meeting clearly states that...*) or they can be used in the research paper as an introduction or for quotes.

One obvious kind of documentation that I will use is the user manuals and specifications of the graphical environments. Another sort of documentation I use is the collected computer programs that the students have written in the graphical environment they have tested. These programs can be very interesting to see for several reasons: First of all it can be interesting to see what sort of chances the students dare to take when programming. Another thing that can be interesting to see is if the students have tried to make their own methods instead of just calling the already given methods in the program.

5.1.3 Action research

Action research is a method that is very applicable in IS. It is not easy to find a good definition of action research that is easily understandable. One definition is given in [4, page 136]:

> *The type of learning created by action research represents enhanced understanding of a complex problem. The researcher obtains information about a particular situation and a particular environment. (...) The aim is the understanding of the complex human process rather than a universal perspective truth.*

In action research, the researcher is an active participant in the development of a system, but at the same time he/she is interested in evaluating a certain intervention technique [7, page 82]. In this type of research, it might be hard to stay neutral in the middle of the workers, administration and programmers. I believe it can take a while for the researcher to earn everybody's trust and get all the information he/she wants. It can be discussed whether it would be an advantage or a disadvantage for the researcher to be employed in the company the research is taking place. On one hand one gets access to more information and people as an employee. On the other hand, other employees or management might think the researcher has a hidden agenda and therefore are reluctant to give out information.

Action research can be said to be research into how the changing of a part of a person's reality affects this person. Usually the course of this research goes as: First the researcher analyse and find facts about what he wants to change[2]. Then the researcher plans how this change shall

[2]Change here can also mean adding something to the reality of the research subject, not just change what is currently there

happen, by making some part of the system which shall be introduced to the subjects. Then the researcher looks at how the subjects react to this new change and then evaluate what the change meant. This is actually something I will do in my research: I will introduce a new environment into the learning experience and see how the students react. However, I will only try this on some students, I will not make a change in an entire organisation (in my case a course). So on some level I will conduct an action research, but I will not look at the relationship between the administration and the workers (in my case the lecturers and the students).

This means that I will use some of the principles and strategies of action research. For instance, I will be actively involved with the students, working with them, showing them the environments, maybe even program a little with them to see how they work with the environment. I will also try to set up a discussion with a few of the students like we were both on the same level and I am not some authority figure telling them what to do. However looking at the system from an intervention point or looking at conflicts between administration and worker is not something I will do. As stated earlier, I be a part of designing a coursework for one experiment and be an active observer in another. I will therefore not be changing an existing course, I will be making a new one. Both experiments are new courses except that in the experiment where I am an active observer I will only be using an existing tool, not adding any course material too it, save from a quick introduction. Action research will therefore not be possible to use in my experiments.

5.1.4 Ethnographic research

Ethnographic research means that the researcher gets very close to the people he/she is observing. Ethnographic research is explained in [19, page 177]:

> *Ethnography offers a rigorous approach to the analysis of the institutional context of information systems practices, with the notion of context being one of the social construction of meaning frameworks. Ethnography, as a research method, is well suited to providing information systems researchers with rich development and application.*

Ethnographic research has the weakness that it can become too narrow in its' study. To be able to develop a more general knowledge of a situation, the researcher has to research many more situations, and this can be a very time consuming process. Ethnographic research could actually be a very interesting approach in my research, as it can enable me to look at a computer course as a complex social and cultural system and see if there are any social factors involved in learning computing[3]. An interesting part of ethnographic research that I will also use in my research is the study of how the student's different social backgrounds affect their learning. I will try to make some mental notes of things I find interesting as the experiments develop.

5.1.5 Grounded theory

[2] has this to say on grounded theory:

> *Grounded theory is a research method that seeks to develop theory that is grounded in data systematically gathered and analyzed. (...) Grounded theory approaches are becoming increasingly common in IS research literature because the method is extremely useful in developing context-based, process-oriented descriptions and explanations of the phenomenon.*

In my research, grounded theory can be a good way to go, but at the same time, I need to be careful not to spend too much time on the project, as mentioned above in the previous section. I think a big drawback in grounded theory is the amount of time spent on the gathering and analysis of data, especially if you have done something wrong in your research, this can mean a lot of time wasted. A positive thing with grounded theory is that you have a very solid base for your theory and in that way can persuade your readers that you are correct.

5.2 Quantitative methods

A definition of quantitative research is this [2, page 5]:

> *Quantitative research methods were originally developed in the natural sciences to study a phenomena. Examples of quantitative methods now well accepted in the social sciences include survey methods, laboratory experiments, formal methods (for example, econometrics) and numerical methods such as mathematical modelling.*

Something that is typical about quantitative methods is large amount of data. Often data from an experiment that has been run many times on a computer[4] or from laboratory experiments. The data are often collected with some sort of instrument (computer, scale, laser etc.) and is easy to

[3]Articles have been written around the positive factors of for instance pair-programming and a 50-50 male-female class or more female than male students

[4]It's common to run simulations of experiments thousands of times on a computer and store the data.

quantify and then use charts as basis for a conclusion of the problem researched.

I find the social and human perspective more interesting and want to be very close to my subjects. Quantitative research does not offer this opportunity in the same way as qualitative research. Quantitative research is better suited for a researcher doing research theoretical field like physics and mathematics. I see that it will be hard for me to use quantitative methods in my research as I feel I will not get enough subjects for my research to have a good basis to quantify well enough. Another thing is that I will conduct my empirical studies in only one semester of my work on this thesis[5], and therefore I will probably just have one set of data as I will focus on one class for my study.

5.3 Summary of methods

To summarize, I focus on experiment research with active and passive observations and unstructured talks. This is because it fits well into the area of research I am concentrating on. In addition I will bring in some elements of other research methods as well as they prove usefull to me. My main concern no matter what methods I choose, is the time frame I have to work within. It is very important that I manage to limit my area of research just as much as to make it exciting. A problem with doing research is that there will probably show up many interesting points during the period of research that would be fun to look into but which I am unable too because of the time constraint.

On a final note, the establishment of a formal research agreement is important, especially in, an organization where there is sensitive information and people's jobs might be at stake. In the classroom environment I will work in, everything will happen on a voluntary basis and the only thing at stake is a person spending some of his studying time on me. It is important however, to assure the students that the things they program and tell me will not in any way be evaluated or shown to anyone besides me.

It will also be important to get a good tone with the students I use in my study. This is both to make them open up to me more and to ensure that the experience of testing the environments is a positive one. I want to learn as much as I can from the students, but at the same time, I believe the students will learn a lot too, both about programming and also to get a look into what sort of work is being done at a Master level here at the Department of Informatics.

[5]My thesis work looks roughly like this: 1.sem: Read theory and conduct a few experiments to learn; 2.sem: Empirical studies and collect data; 3.sem: Writing of thesis

Chapter 6

The Experiments

As my thesis is based on testing different programming environments and their use in teaching OOP, it was natural that these environments were tested in a classroom setting with students that had little or no experience with programming. On two occasions did I carry out experiments using different environments.

6.1 Pilot experiment

As noted, I wanted students with no experience from OOP. However, in my first experiment I used students with some months of a semester of programming experience and therefore they did not fit my criteria of proper subjects. On the other hand, one can hardly say that students with one semester of programming training are experienced programmers. The pilot experiment was an opportunity for me to practice conducting experiments as this was something that I had never done before. It was a three hour test of the environment Robocode. I announced the experiment at a lecture and asked people who were interested to show up at an appointed place. Only three people showed up at this experiment as it was a bit close to the course exam and late announcement. After a short introduction to the environment, I let the students play around with Robocode on their own, using the info they got from a handout I gave them (found in Appendix A) and the API of Robocode on the Internet[1]. What I was looking for in this experiment was if the environment encouraged the students to play around with coding. The students sat alone in front of a computer and programmed without much help from me. I walked around, talking with the students and answering whatever questions they might have for me. These were seldom programming related, more technical (”How do I start this? How do I set up

[1]http://robocode.alphaworks.ibm.com/docs/robocode/index.html

the game?"). There is a code example in chapter 7, code listing 7.4.

6.1.1 Summary of the experiment

The students had some experience as they were first year programming students. They were about two or three weeks away from their final exam in the programming course they were attending, meaning they had studied programming for about 13 weeks. So the introduction went fast as they were already familiar with programming expressions and the basic dynamics of a program. The students managed to work independently and did not ask many questions from me. What i noticed was that two of the students, obviously friends, had a sort of friendly competition between them. During those three hours, the subjects seemed to understand the basic functions of Robocode and be able to program some new behavior. How much new Java programming they learned during two hours might not be much but I am sure they got a new illustration of how objects interact and communicate. The subjects were given a repetition of the use of while and for loops and if statements. One fairly new concept they were introduced to was Event handling: Seeing how an object reacts to and handles a certain event. This event can be getting hit by a bullet, hitting a wall or seeing another tank.

6.2 Experiment 2

The second experiment was a part of the COOL project and was aimed at testing Karel J on students that had no programming experience. In this experiment, I was not alone as there were others from the COOL project that were also interested in playing a part.

Prior to the experiment, we had to structure the material we were going to use and hand out to the students. As I wrote in chapter 3, the structure of the coursework needs much consideration. The designer of Karel J, M. J. Bergin has designed a course based on Karel J. We chose to base our course structure on this[2]. We did not have time to cover everything in our 2.5 days, so this was what we covered:

1. Making one object and manipulating it to move around by using the methods associated with the object. We expanded the methods that were able to be used one by one so they got familiar with one method at the time.

2. Making two objects of one class, emphasizing that they had to have different names so that it was possible to separate them. Then we

[2]http://csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html

manipulated the two objects using the different names to illustrate our point. No new methods were introduced.

3. Making a subclass of the robot class and adding a turnRight method that we then called from main and used the new method implemented.

4. Making a more advanced method in the subclass to illustrate a good use for subclasses.

5. Starting with if and $while$.

6. Making an example where the constructor is somewhat more clearly illustrated.

7. Advanced examples using Robocode.

Weeks prior to the experiment, emails were sent to 150 random people who were going to start at the university that semester. We asked for people with no OOP experience. 12 people answered this call within a week. We set a max limit at 20 so this was a good result. On the day of the experiment, only 6 people showed up. We can only speculate about the reasons for this, but the important thing is that we had some subjects that filled the criteria we had set. The subjects were all male and from 19-23 of age. The six subjects had some experience with using computers before but programming wise they were all fresh apart from one who had tried a little BASIC.

The experiment was conducted over 3 days with a total of approximately 15 hours. We set the students up two and two on computers to encourage discussions that we could observe. The tools we used in this experiment was ordinary blackboard and a video projector so we had a way of demonstrating programs on a PC. In addition we had some written material that we handed out as the different activities unfolded, see appendix for the complete hand out. The basic outline of the hand-outs were examples of code and drawing of Karels world and how the code affected this world. We tried to keep the amount of written material to a minimum, relying more on hands-on experience while using what we had written as an aid for the students to remember syntax. Four people worked on the experiment: Two were teachers, me being one of them and the other two were observers. The students were not bothered at all by being observed. During the experiment we tried our best to be aware of how we used our language, trying not to use advanced words and be coherent in what we said to the students. The reason for this was that we felt that the students could easily be confused by two teachers explaining theory using different language.

6.2.1 Day 1: Karel J

As we have no video from this day, taking notes was very important. It is however, harder to get a good analysis from written notes. This is a personal preference. I usually have trouble remembering something right after it has happened and so many of my impressions are lost. Recording also offers the possibility of running through the observations many times and capture small subtleties that are not easily spotted.

We started out by talking a bit about computers and reminded the students how primitive computers really are and that they just do as they are told. We then continued by introducing Karel J, describing the world and trying to get the students to imagine the robot world where the robots are ordered from a factory based on descriptions or work drawings. We referred to all robots as objects from the start, but not in a programming sense, more just to make the students used to using the word object of physical things in the world. We did not use the word class yet as that would be useless without some code to illustrate with. During this introduction we handed out ”karel intro” but not giving the students time to read it.

After the 25 minute introduction we started the students on Activity one, making an object. We handed out ”karel 1” and ”Activity 1”. Activity 1 was as follows:

> Look at the map (here, figure 6.1) and make a robot that can move from the intersection of 1. street and 4. avenue to pick up the beeper.

We demonstrated, using a video projector, how to make a new program, we introduced the first line of the program and showed how to save, compile and run.

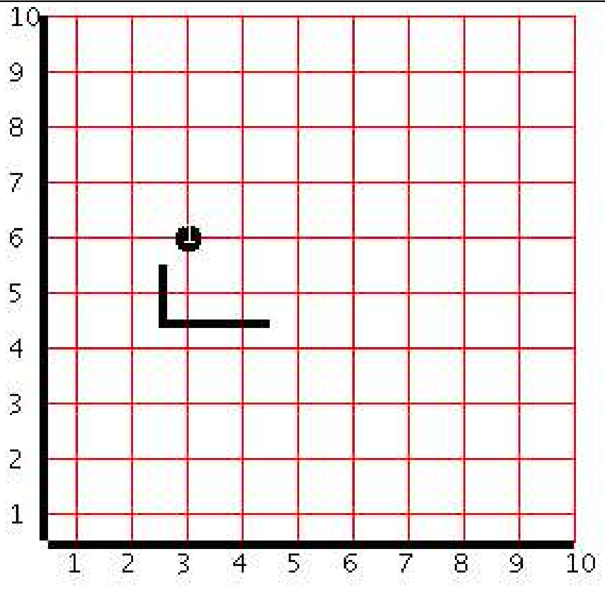
We wrote the following line:

```
ur_Robot karel = new ur_Robot(1, 4, North, 0);
```

At this point, we got the first question asking if ”karel” was the object. The answer of this question was yes. This is of course a modification of the truth: ”karel” is only the reference to an object of ur_Robot, but it seems that the students were creating a sort of mental model of the program that they found easy to understand and we did not want to say too much right away that might be hard for the students to handle. I was quite amazed at how easy the students started to use object oriented terminology. After having written this line, compiled and run the program, we introduced the next line and compiled again:

```
karel.move();
```

Figure 6.1 Map in Activity 1



After this the students started talking among each other, pointing at the screen, looking through the handouts, drawing on the map and counting. Soon, all the students had completed the assignment and reached their objective. The students compiled and ran the program many times as they wrote new lines. Upon completion, they started experimenting: *What happens if we crash into a wall? Can the robot put out beepers as well?* It was apparent that the students were encouraged to experiment because of the graphical representation of Karel J and the fact that they saw results fast. After this, we handed out ”Summary 1” and talked a little about what we had done, showing that we had to use the name of the object to access its properties (methods[3]) At the same time we explained that the robot could not turn to the right, having to use $turnLeft()$ three times instead.

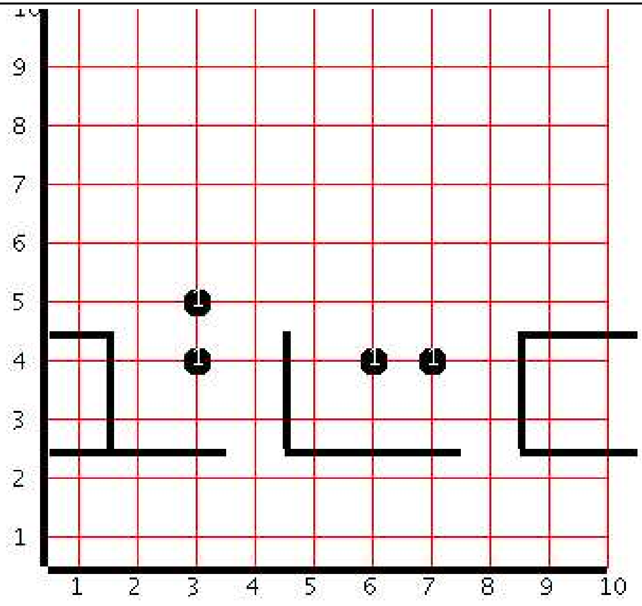
We then went on to the next activity, multiple objects. We explained that it was possible to order more robots of the same type from the factory. Then we handed out ”karel 2” and ”Activity 2” and let the students work. Activity 2 was as follows:

Look at the map (here figure 6.2). Make two robots that can

[3]We did not use the word ”methods” right away. We kept the terminology at a minimum for a reason: Get students comfortable with words they understood fully and then advance.

pick up one beeper each. One robot starts in the crossing (1, 4) and the other robot starts in the crossing (1, 8):

Figure 6.2 Map in Activity 2

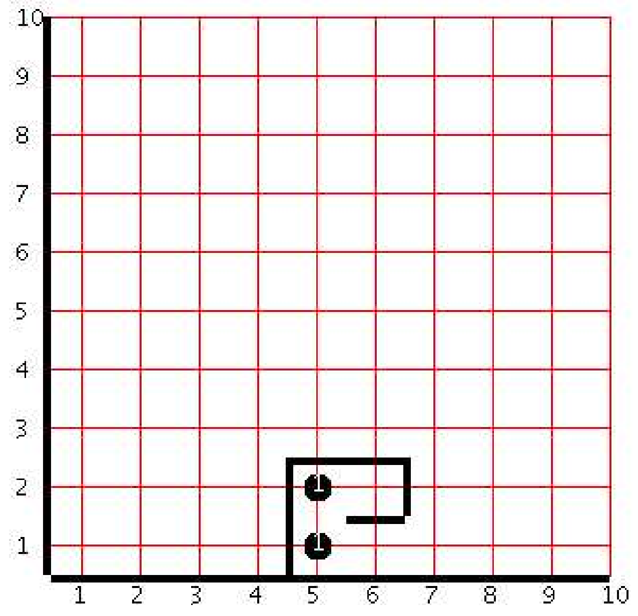


One of the students pointed out to the other that they had to give the objects different names while another student asked if they could work in parallel. I answered that it was a bit hard but possible. He said the closest suggestion he had was letting each robot execute one line each all through the program. It seemed that all the students understood and executed the task well. We rounded off by handing out ”Summary 2” and talking about the importance of different names on objects and being able to separate them with at least one property that was not the same for each object. In addition to using the robot analogy, we used a car analogy to aid us just to be able to draw parallels to other examples. The students did not know the difference between a reference and an object, but we felt they had a good grasp of what was going on any way.

These activities lasted until lunch. We may have proceeded a bit fast and should have been repeating some more, but we were also interested in seeing how far we could reach. After lunch we started on activity 3, sub classing. We handed out ”karel 3” and ”Activity 3-1, 3-2”. Activity 3-1 is as follows (Activity 3-2 explained further down):

Look at the map (here figure 6.3). Make a robot that understands the command ”turnRight”. Starting in (1, 4), pick up the beeper.

Figure 6.3 Map in Activity 3-1



We said we wanted to make an improved version of the factory's robot that understood the *turnRight*() command. We said it was possible to hand in our own description or work plan to the factory. Making the *turnRight*() property would mean making three turnLeft. A question was then posed why we couldn't just tell the robot to turn 90 degrees to the right instead of 270 degrees to the left. Our answer was formulated thus:

> The way to communicate with the computer is built up of layers. In the bottom we have byte code, but that is too hard to use, so we have built a more understandable layer on top of that with some means of interpreting this layer to the lower level. And so we have built many layers, trying to make the means of communication easier for each layer. Karel is such a layer, and changing or adding some functionality of a layer requires knowledge of that layer, and since we don't have that, we use the tools we have.

We then started to program on the projector, going through each line:

```
class ur_Robot2 extends ur_Robot
```

We now used the word **class** for the first time. We said that a work description of a new type of robot is called a ”class” in the programming language. The extends keyword falls very natural when we have already said that we want to make new extensions of the old robot. We then went on by making the constructor, not saying too much about it at this time, just saying it was something that gave the object our desired preset characteristics, like an order form containing some variable data. Understanding the constructor was a bit hard, but we said we would get back to it later. While the students worked on the activity 3-1, I talked to a couple of students about the constructor. Here is a code example written by me:

Listing 6.1: Code example, constructor

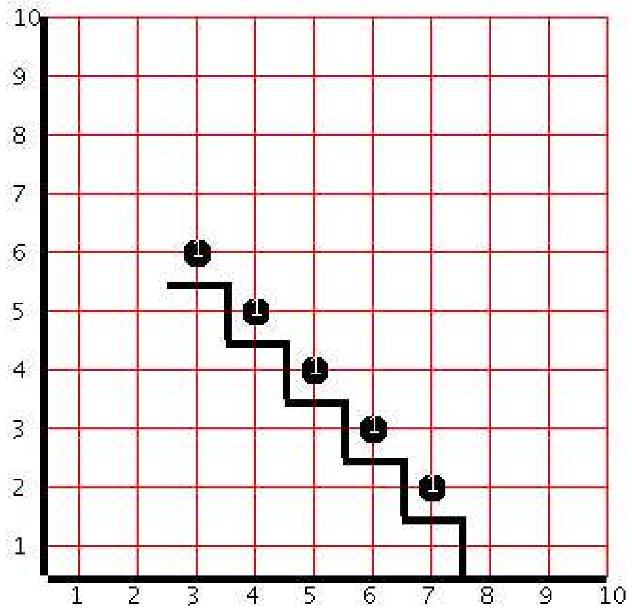
```
 1  /*
 2   * Constructor example, using karel notation
 3   */
 4  class Example implements Directions {
 5    public static void main(String [] args) {
 6      // Making a new object of a subclass
 7      SubRobot karel = new SubRobot(1, 2, North, 0);
 8    }
 9  }
10
11  /*
12   * Subclass that extends a Robot
13   */
14  class SubRobot extends Robot {
15    // The constructor
16    public SubRobot(int str, int ave, Direction dir, int beeper) {
17      super(street, avenue, dir, beeper);
18    }
19    // Methods follows
20  }
```

The big problem was the sending of parameters into the constructor (line 7 and line 17), to the students it seemed like a lot of extra work. This problem was a bit frustrating because we had to tell them that this was how Java worked. The students seemed to understand a bit better when I said that the constructor had to be written in this way so it could be possible to call the same constructor many times.

Other than that, the students had no problems, meaning they were able to solve the rest of the activity, but without understanding constructors, just accepting that this was the way it was. Activity 3-2 was as follows:

> Look at the map (here figure 6.4). Starting in (1, 8), get Karel to pick up the beepers. (Hint: Try to find a pattern in karels movement)

Figure 6.4 Map in Activity 3-2



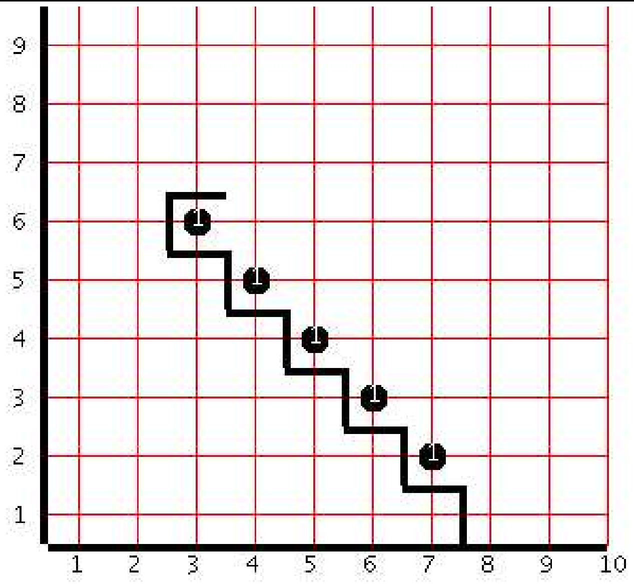
After they started on activity 3-2 I talked to another pair of students. They asked me if this wasn't the same as we had done in earlier assignments, just counting the way up and calling properties of the robot. I asked them if that wasn't a bit much work. The answer was: "Well, yeah. (pause) Oh, we can make a method for each step as they are the same." They agreed that would save them many lines of code. The three groups all solved the task using different approaches: One group reused the code from the last assignment; one group wrote the code off the handout (karel 3); the last group wrote a new class extending ur_Robot2 so they can use turnRight. The last group were a bit unsure whether they could call methods from within a class or if they had to make a class for each method. Having explained that this was possible, they solved the assignment.

We rounded off by talking about subclasses and super classes, using those terms. We handed out "Summary 3". Constructors were still a bit hard, so we planed on making an example for the repetition the next day. Activity 3-2 was a very important activity and the one we used most time on.

We proceeded to activity 4: Imperative programming. We handed out "Karel 4" and "Activity 4-1" and said that it was time to save even more lines and make the robots a little bit smarter. Activity 4-1 looked like the following:

Look at the map (here figure 6.5). Having learned about how karel can relate to the world around him, start in (1, 8) and let karel pick up all the beepers before he crashes in the wall at the top of the stair. This time, assume you don't know how many steps there are.

Figure 6.5 Map in Activity 4-1



We started by talking about if-tests and while-loops, avoiding for-loops as they require the use of variables as well. We discussed the assignment and wrote pseudo code on the projector. People came with their views and possible solutions. We soon came up with a solution using a while-loop so we get people coding. This took the rest of the day.

Concluding remarks, day 1

Here follows a short summary of thoughts I had about the project's first day. They will be discussed in-depth later.

- The students progressed very rapidly for many possible reasons:
    1. The students work together two and two.
    2. The people who signed up have a particular interest in computers.

3. It was a very small group so it is easy to follow up and make sure everyone understand.
4. Karel J helped the understanding.

- Karel J made it a lot easier for us as teachers to have something tangible to relate our theory to. The students seemed to take things easy when they can discuss problems using the robot world as a setting.

- Karel J made the programs the students produced very small so it was easy to have full control of the code. The graphical interface showed results fast and encouraged compiling often, something that is a good programming practice.

- Subclasses were the hardest thing we taught this day. It is also a very important aspect of OOP. The logic of using subclasses was easy enough to understand, but the code specifics of the constructors and the reasons for their existence was a bit harder.

- The students were very active in their discussions. It is very positive that they can correct each other and discuss different ways to solve something. One drew on paper and pointed while one typed.

- The students quickly became used to the concepts that are being used and easily slipped into a mode of talking OO.

6.2.2 Day 2: Repetition, more imperative and Robocode

Before this day we had made another example of sub classing and called it ”Activity 5”. The day started off with some repetition of the previous day. Two of the pairs were also filmed on video this day. After a 20 minute repetition we handed out ”Activity 5” and let the students work with this.

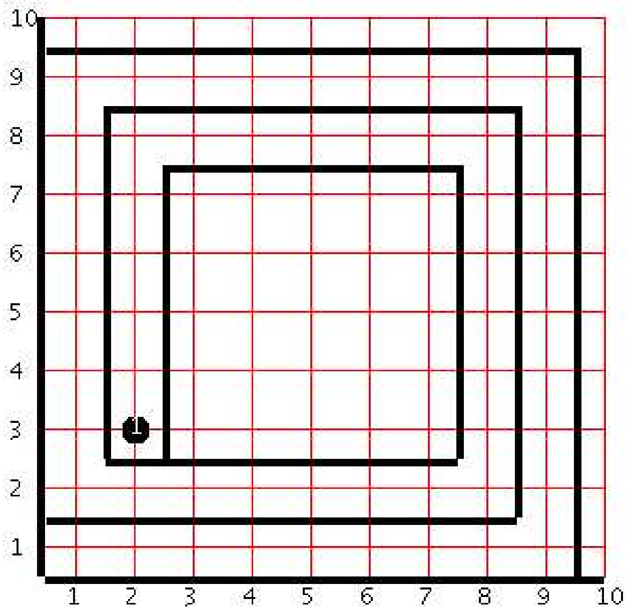
> Use the same map as yesterday (here figure 6.5). Use the pre-written class, and pick up the beepers and then print out on the screen how many beepers karel has picked up.

The solving of this assignment will be discussed in more depth in the chapters about progress of the different groups.

The students worked at different paces on this assignment so we just added some more parts to the same assignment (found in Appendix B) and let them work on this. Right after this we handed out ”Activity 4-2” without saying anything. This activity looked like this:

Look at the map (here figure 6.6). Go through the maze, starting in (1, 1) and pick up the beeper. Karel does not know how far he has to walk or where the beeper is. However, he knows that he only has to turn to the left.

Figure 6.6 Map in Activity 4-2



The students had some starting problems with both assignments and had forgotten some of the things from the previous day. This is not so strange as we had covered many topics the previous day. But after some hints and tips, the students were able to solve the assignments. Activity 5 and 4-2 were the two hardest Karel J assignments we had and used everything they had previously learned except generating multiple objects of the robot. Generating multiple objects was something we saw we needed to incorporate in the assignments on the last day. These two assignments took a while (not unexpectedly) and we finished with a small summary around lunch.

After lunch we had another small repetition before we introduced the students to a new environment: Robocode. I was a bit unsure as to how much they would be able to do with this system. Robocode is much bigger and more complex, and the students had only had about 7 or 8 hours of effective programming. After a short introduction, explaining about making a new tank and handling events, we let the students start. This was suddenly much harder, the main problem for the students was to get an overview of what the classes in the system offered. We encouraged them to look at other tanks to see what happened. One pair of students borrowed and modified code from other tanks. One pair modified tanks that had been written by someone else. Another pair of students had trouble getting an overview and tried a little bit, but they soon got stuck and needed hints from us. After a little help they tried to make some methods and succeeded in making some simple event handling. One student (now alone) managed to make some very nice methods and from discussing with him I see that he understood these events. He for instance made a fire event where the power of his shots was based on his own energy and the distance to target, he received a little help with some of the coding, but he had all the ideas, he just had a little difficulty finding all the code.

Concluding remarks, day 2

This was a day of much frustration for the students. Parts of yesterdays lessons were forgotten, not very strange because of our pace. Robocode became too big and complicated for them to fully get an overview.

However, the students showed that they understood the logics of small programs and were comfortable with using OO terms. Explaining programming on a slightly more advanced level was easier as the students were becoming familiar with the terms. It was also really easy to explain how Java works when the students understood the class and object terminology. Explaining *main* and the class containing *main* also became a matter of simply saying that this class was a special class that Java needs to be able to start. It took us 5 minutes of talk to ”demystify” *main* and the main class. Algorithms were still something the students struggled a little with, but algorithms were never the goal of this experiment. We moved to algorithms because the progress of the students was so fast. Maybe we should have spent more time on repetition? Although these students had not programmed before, they had used computers for some years. Still, their advance was impressive.

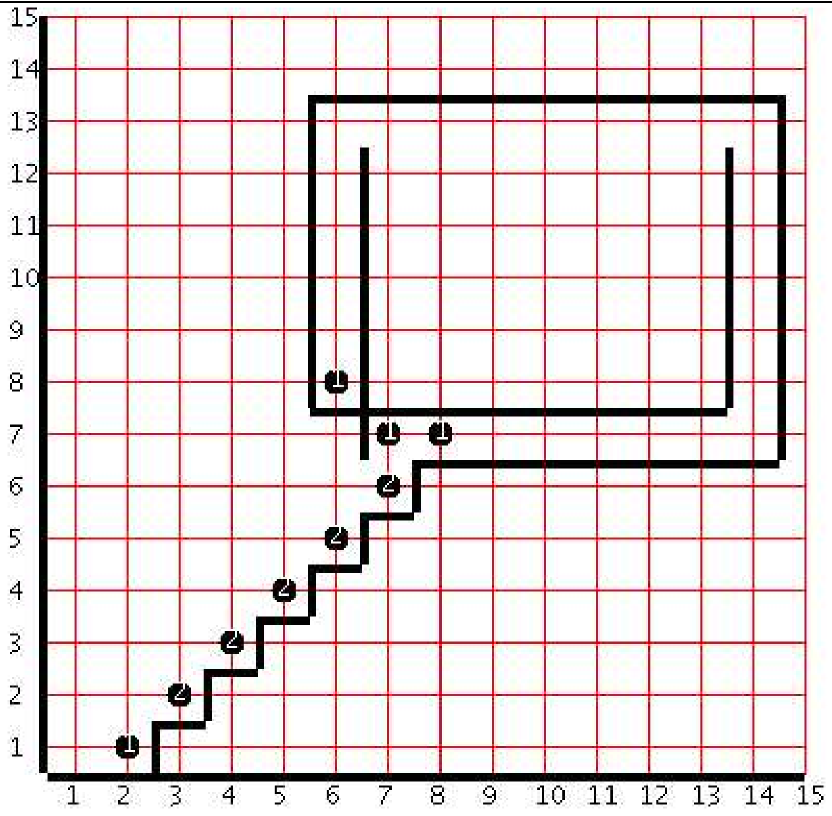
6.2.3 Day 3: Karel J + repetition

On day 3, two students were sitting alone, because someone had not shown up, so we paired them up after a short while. The group, now G2 consisted of one person from the old G2 and one other person. This new person was however the same type of person as the one who was not there, having the same background knowledge. I therefore choose not to make this new G2 into a new group that I compare as the dominating person was still the person from the old G2 and the new addition to the group had the same type of knowledge. It would have been interesting to see what sort of solutions might have come if the newest member of G2 had dominated, but this can only be speculations.

It seemed that the things that we had done on day 1 and day 2 had sunk a bit more in, so the students seemed a bit more comfortable. We did a short repetition about karel J before handing out the final assignment. The two groups of our focus chose different ways of approaching the problem. This will be discussed more in depth further down. The Activity looked like this:

> Look at the map (here figure 6.7). One robot starts in (1, 2), the other starts in (8, 6). Both robots shall pick up 7 beepers and they can not pick up the beeper they are standing on.

Figure 6.7 Map in Activity 6



The thought behind the assignment was for the students to reuse code from earlier assignments, put each solution into a method of a subclass and then call the methods from main. Group 1 did this, but they did not reuse as much code as they could and ”reinvented the wheel” in a larger degree than was necessary.

The assignment they were given was not an easy one, but the groups managed to solve it with a little help. They solved the assignment differently, one group using a object oriented approach and the other group using a more procedural oriented approach, more on that later.

After the assignment was done, we evaluated of what the students had learned. When we asked how they understood what a class was, a student answered that it was a blueprint for a group of objects. That is a pretty insightful observation into OOP after only three days of work.

Concluding remarks, day 3

Having a break of a few days helped so the things the students had learned had some time to mature (there is off course also the chance they forgot some of the things they were told!). This was the first big assignment that really showed big differences between the groups.

The students showed that they understood basic OO concepts, by making methods in subclasses and using pointers. They also show that they have learned some problem solving skills such as pseudo code, drawing, testing and debugging. Here, Karel is helpful as it gives a very visual representation of what is going on in the code. The approaches to problem solving differs from the two groups however, as seen in the next chapters.

6.3 Progress of the main groups

We followed two groups with one video camera on each of the two groups. This was done on day two and three, collecting about 5 hours of video of each of the two groups. It is interesting to see the advances the two different groups did. I managed to follow the two groups pretty well on day one as well, so I have a good view of how they did on the first day. Off course, I have a much better view of day two and three, but I will try to make an as accurate description as I can. However, the most interesting day by far was day 3, seeing how the two groups worked so differently. But it is important to look at how the two groups evolved into day 3. First, a short profile of the two groups:

- **G1:** Group 1. The group consisted of two boys. Both were familiar with computers, but none of them had programmed before. They did not know each other before this experiment started.

- **G2:** Group 2. The group consisted of two boys. Both were familiar with the use of Linux[4] and computers in general. One of them had programmed a bit in Visual Basic. They did not know each other before this experiment started.

6.3.1 G1. Day 1

Both activity one and two were solved by typing and testing, as we had told them to. The members of the group talked well together and discussed. One typed while the other person drew and counted on the maps handed out and told the other what to write, but he always added a *What do you think?* at the end of his instructions to get the feedback from his partner. The one who typed usually pointed at the screen and the two discussed where they thought the robot would go next. This way of working was steady through all the days. After the two first assignments they started using papers more, sketching a solution before starting to program.

It was very easy for them to draw on the maps handed out and make gestures in the air moving like the robot would. In a quasi English-Norwegian[5] language they discussed how to solve the assignments. We constantly tried to encourage them to compile more often as they often got many compiling errors. I believe this is because they solved the assignment on paper first and were very confident in their solution and that it was the right one so there was no need to check it as they went along. They often asked for help when something was wrong. Their main problems were the following: Either they had many compiling errors due to too much writing before testing, or they had been sloppy with their naming and mixed their files, giving the file one name and the class another thus running the wrong files later.

They were the slowest in completing their tasks, and this is due to the following:

1. They solved everything on paper first and discussed the problem.

2. They used a long time fixing compiling errors and naming errors.

However, their solutions were often very tidy and, as the assignments got bigger, object oriented[6]. See listing 6.3 for example code.

[4]Linux is the operating system used here at the Department of Informatics

[5]It was very easy for them to use English words as ”move” and ”put” since these were the method names.

[6]The other groups also solved the assignments object oriented, but G1 wrote more in classes than G2.

6.3.2 G2. Day 1

As one of the members of this group had some experience using Visual Basic and the other member had experience using Linux, this group set off very fast in starting to program. They were very keen on exploring the program and not afraid of trying things like crashing into the walls and placing beepers in the map. They did not have much trouble with names and files as they were more familiar with general use of computers.

They did not ask for much help and were pretty quick with their first assignments. I believe this is because of some experience with programming so the first tasks were pretty easy.

6.3.3 G1. Day 2

The next day started out a bit harder for this group. When they started with activity 4-2 they made many mistakes that made it look like they had forgotten what we had taught them the day before. This is not very surprising as we did cover many topics the first day. This we did because we had that impression that all the students handled the assignments given to them and understood the theory presented. A question raised here will be:

> Did we move forward too fast or was it simply that the group had forgotten?

There are many factors that can have played into the cause of their errors, one being that it was early morning and that the students were tired. This is very plausible as the group perked up much after a break.

When solving the assignments, both students spent some time reading the assignment and drawing on paper trying to understand the assignment as well as they could.

6.3.4 G2. Day 2

The group was given Activity 5 and started by looking through the file, discussing different elements of the file and getting quite caught up in the technical details of the code, for instance, they asked me the meaning of ++[7]. They started very fast with the coding and compiled right away to check if the map was right and if the code they had written worked. There was a short discussion about what class to use (ur_Robot or the new subclass). One of them quickly said that they were supposed to use the new subclass because it was where they got their methods. After a little problem with the file-class-name, they were able to compile. And,

[7] ++ is an increment terminology, meaning one increment a veriable by one

confident that the basic details were in place, set off coding. They first tried just one step (calling the step method once). Seeing that the first beeper was left unpicked, they added a pickBeeper at the beginning of the code. A source of some problems was the *leggTilTeller*() method as they thought that this was supposed to be used for controlling how long the robot was supposed to walk.

They soon remembered however, how to use theory they had learned from Activity 4-1, and used a while loop for controlling how far the robot should walk. The rest of the assignment went fast.

After they had finished with the assignment, they played a bit around with the code to try to better understand what they had done. For instance, they changed the constructor and the parameters in the *super* call to see if that had any effect.

After having solved the assignment, the group was asked to place out the beepers they had collected. They did so without problems.

This done, the group was handed Activity 4-2. They decided to reuse the subclass from the last activity, as one student said: "it has the *turnRight* method". There was actually no need to use the *turnRight* This assignment focused on imperative programming. It was designed to teach the students some imperative thinking so they would have some more code to put into their subclasses later. There was no need for them to use a subclass here, but they used that anyway. The students had a very imperative way of thinking: "We need a while loop for testing the walls, then another while loop for testing the beeper. Or maybe an if". The assignment adds some difficulty as it requires that the students pay attention to two things: The beeper and the walls. This requires both an if-test and a while loop.

The group discussed much and most of it evolved around while loops. They were thinking about using double while loops to solve this task. The added difficulty of two things to be mindful of was a big constraint for the students. The main problem was to figure out a good stop condition. Their final solution looked like this:

Listing 6.2: Code for group 2, activity 4.2

```java
class Karel42 implements Directions {
  public static void main( String [] args ) {
    TrappeRobot karel = new TrappeRobot (1, 1, East, 0, 0);
    while (karel.frontIsClear()) {
      if (karel.frontIsClear()) {
        while (karel.frontIsClear()) {
          karel.move();
          if (karel.nextToABeeper()) {
            karel.pickBeeper();
          }
        }
      }
      karel.turnLeft();
    }
    karel.turnOff();
  }
  static {
    World.readWorld( "eks4-2.kwld" );
    World.setVisible( true );

  }
}
```

6.3.5 G1. Day 3

The group started off as before, showing a very structured way of solving the problem they were presented with. Also as before, it took a long time before they started coding, and when they first began, they headed straight into trouble. The big problem was naming. Their file was called ”Karel5-1.java”, and their class was called ”Karel42” as they borrowed code from last session and forgot to rename the class. This obviously led them into difficulty[8], and it took some time for them to straighten it out. They were very good at borrowing code they saw they could use. As this last activity was a combination of the two previous assignments, the group borrowed code from both places.

Again, they ran into compiling problems, but it was nothing major, only syntax errors. However, due to the amount of code written, they were a bit hard to spot right away.

The activity had four major tasks for the robots to perform: Robot one up the stairs and through the maze; robot two through the maze and down the stairs. and the group put all these tasks in separate methods in a subclass. This made it very easy for them to test the code later by commenting out the method calls they knew worked.

6.3.6 G2. Day 3

Starting off, this group also borrowed code from the earlier project. That meant that the long imperative code in main followed them through to this day. The stair went well after getting the code from Activity 4-1.

The rest of the assignment was very much like the day before: Trying out things, discussing what to do and throwing in a *while*, *if* or another statement whenever they faced a problem, so the code got pretty messy

[8]When a java program is compiled, one or more class files are generated. The names of these class files is what the Java Virtual machine looks for when it runs the program. These files are named after the names of the classes in the program, not the name of the java file.

after a while. This led to trouble each time they tried to test the code. For one, it took a very long time to test[9] and they also had trouble commenting out portions of the code, so they had to run the whole program each time they wanted to test a new add-on.

Their approach was dominated by testing and more testing and looking at old code to try to solve the problem at hand. Their main method is presented in the next section.

6.3.7 The code from day 3 for each group

It is interesting to see how the two groups differed on the last assignments. In this chapter I will bring in fragments of the code, while the whole code will be attached at the end (Appendix C). This code will also reflect the code from Activity 4-1 and 4-2. I only present the main methods here, because this is what shows the organization of the rest of the code.

G1: OO-style

This group's main method was very short and tidy:

Listing 6.3: Code of group 1

```java
class Assignment6 implements Directions {
  public static void main( String [] args ) {
    // Uses a class written in an earlier assignment (5)
    TrappeRobot Karel = new TrappeRobot(8, 6, North, 0, 0);
    TrappeRobot Jon = new TrappeRobot(1, 2, North, 0, 0);
    // Calls methods for Jon first
    Jon.plukkTrinn();
    Jon.turnRight();
    Jon.finnPiper();
    // Then calls methods for Karel
    Karel.move();
    Karel.finnPipe2();
    Karel.turnLeft();
    Karel.plukkTrinn2();
  }
}
```

Here we see that the group will have no problem in testing just some code fragments.

G2: Imperative style

This group's main method was big and messy:

[9]Nested while-loops with if-tests takes a long time to execute.

Listing 6.4: Code of group 2

```java
class Assignment6 implements Directions {
  public static void main( String [] args ) {
    // Writes a new subclass
    ur_Robot2 karela = new ur_Robot2(8, 6, North , 0);
    ur_Robot2 karel = new ur_Robot2(1, 2, East, 0);
    // Use while in main as a controll statement
    while (!karel.frontIsClear()) {
        karel.trapp();
    }
    // Go through the maze
    while (karel.frontIsClear()) {
        if (karel.frontIsClear()) {
            while (karel.frontIsClear()) {
                karel.move();
                if (karel.nextToABeeper()) {
                    karel.pickBeeper();
                }
            }
        }
        karel.turnLeft();
    }
    karel.turnOff();
    // The other robot goes through the maze
    while (karela.frontIsClear()) {
        if (karela.frontIsClear()) {
            while (karela.frontIsClear()) {
                karela.move();
                if (karela.nextToABeeper()) {
                    karela.pickBeeper();
                }
            }
        }
        karela.turnRight();
    }
    // Fix to get the robot on the right track
    karela.turnLeft();
    karela.turnLeft();
    karela.move();
    karela.pickBeeper();
    karela.turnRight();
    karela.move();
    // Down the stairs
    while(karela.frontIsClear()) {
        karela.trapp2();
    }
  }
}
```

From an OO perspective, this is not good, but I am impressed that they have actually made this program running! It shows some heavy imperative programming. It is apparent that OO is not the governing path here. Testing and experimenting and a small portion of luck played a big part in this code.

6.4 Comparing G1 and G2

The two groups differed a bit in their approach and their way of working. The most obvious differences were:

- **Help from teachers:**
  - Group 1 asked for help whenever they faced a new problem.
  - Group 2 liked to solve it on their own, but they usually sat looking for us when they trouble.
- **Problem solving approach:**
  - Group 1 read the text thoroughly and drew possible solution on paper and wrote a few lines of pseudo code before even starting to program.
  - Group 2 started to code right away and solved problems as they came up.
- **Coding style:**
  - Group 1 coded much before they started testing because they felt they had solved the problem satisfactory on paper first. They were also more object oriented in their approach (putting most of the code in subclasses).
  - Group 2 compiled and tested often. They had a more imperative approach to the problems (using while loops and if statements in main to solve problems).
- **Common problems:**
  - Group 1 usually had problems with naming files and classes and mixing these two. They also had compiling problems, as they usually coded much before they compiled and tested.
  - Group 2 also had some naming problems, but the main problem was hard-to-read code because of their imperative way of programming.

But they were similar in some respects as well.

- **Communication:**

    - Both groups communicated well together and used both paper and screen to point and trace the route Karel took. The students seemed very equal, none took specific control. However, as they were all very unsure, the approach that was first presented by one of the members was very often followed as the other didn't have a better idea.

- **Exploring:**
    - Both groups explored solutions and were not afraid of trying something. Karel offers a good environment for exploration and the groups used this opportunity.

6.5 The experiments, do and don't

During the work on this thesis I have conducted two experimental studies, one small and one large.

When planning experiments in an educational setting there are some problems one faces. One is that of when the educational year starts: I was looking for students that were going to start their education at the university. This was not an absolute requirement, but it would make it easier for me to follow them up later. This means that I would have to do this at the beginning of a semester and when I want to use students who are just starting the university, many of these students are trying to get settled in and might find it a bit scary to join an experiment before they know anybody.

6.5.1 Experiment 1

In the first experiment, I had no experience with conducting in-depth studies and this was then a test for me and a way to practice. Looking back, the experiment went well except for a low attendance. This could have been improved with better planning from my side and better information to the students. But I am happy with the content of the experiment itself and what the students did.

6.5.2 Experiment 2

In this experiment, we had 2 and a half day at our disposal (12 hours effective time). We could mainly choose two paths:

1. Use the whole time to get the students to understand the OO concepts very thorough at the expense of not covering all subjects.

2. See how far we managed to come in this time without loosing any of the students under the way.

We chose the second path to test the effectiveness of our program and the tool. The advantage of this approach is that we can see how much students are able to manage on a short period of time and then see if it is hard to teach the OO paradigm. The disadvantage is that with so little repetition, there is a large risk that the students will forget much from day to day and that the knowledge will not be very solid at the end.

6.6 The students after the experiment

I have spoken with the students at later occasions and I asked one of them if the experiment had helped them any, especially when he started with objects in the course he was attending. He said that he had enjoyed the experience much and that he liked the approach. It was just a little sad that it was so long ago, much of what he had learned had been forgotten. This is understandable, we had too short time to repeat subjects for the students and I am not surprised that he had forgotten much of what he had been taught. I have been so fortunate to be the instructor for two of the students from the experiment in the introductory course so I have been able to keep an eye on him. The assignments they have delivered have been good and it is clear that they have a better understanding of OOP than many of the other students on the group and other groups I have visited. If this is due to the experiment this summer or if the two would have been just as good in any case, I can not say.

Chapter 7

Discussion

In this chapter I will summarize my findings and try to link my observations to the theory I have presented in earlier chapters. I have presented some OO theory wherein I mentioned some concepts that are important to understand in order to have a good grasp of OOP. In addition I mentioned some different concepts that are used to facilitate the learning of OOP, as well as some general concepts with regards to the psychology of learning.

In the experiments I observed many of the concepts above and these will be discussed in-depth in this chapter. When I refer to ”Experiment 1”, I am referring to the Robocode experiment while ”Experiment 2” refers to the Karel J experiment. Sometimes I will also refer to the groups from experiment 2 and I will use the same names as I did in the chapter about them, G1 and G2.

7.1 The environments, revisited

In this section I will revisit the environments I presented earlier and look at some differences in the environments that I have observed during the course of this thesis.

7.1.1 Graphical representation

BlueJ, Karel J and Robocode all have in common that they are graphical environments but they differ in many respects.

BlueJ is the environment that separates itself most from the other two. In both Karel J and Robocode, the program execution is the only thing that is represented while in BlueJ, the code can be seen executed by the objects that are generated as well as seeing the program itself. In both Karel J and Robocode, one must keep in mind the setting of the program, either a grid or a battlefield, while in BlueJ it is possible to make any sort of program without having to think of that it has to be compatible with the BlueJ environment.

Karel J and Robocode represent the program written in different ways. In Karel J, the execution is slowed down so that the students can follow the code line by line and see when the program they have written performs an unwanted action. The downside to this is that many students might find the execution being too slow and not test often because this takes too much time. Robocode is much faster in its execution, looking more like a game with engaging action. The downside to this is that the students have a hard time following the execution of the code they have written.

7.1.2 What was difficult

What was difficult with the different environments? Was there some threshold that was high for the students?

Observing the students in the experiments, I saw that they found Robocode a little difficult. The basic idea of Robocode was not hard to understand: Using some methods that made the tank move. It was when the students started to work with many different events (hitting walls, being hit by bullets and so on) that they started to loose track of what was going on. The students easily made the tank move around and shoot a little. It was when the students started to make more intelligent tanks that problems arose: Robocode offers many different events that can be handled: Scanning another tank; getting hit by a bullet; hitting a wall; missing a shot and a few more. Getting an overview of all these events and possible ways to handle them, requires some more experience in programming than the students had.

7.1.3 Usage

The environments are all good to use when teaching OOP, but the areas where they can be used for best effect vary. In the beginning, students will have problems being able to use every aspect of Robocode due to its size. However, as the students develop more understanding for programming, Robocode is very good at illustrating imperative programming and event handling.

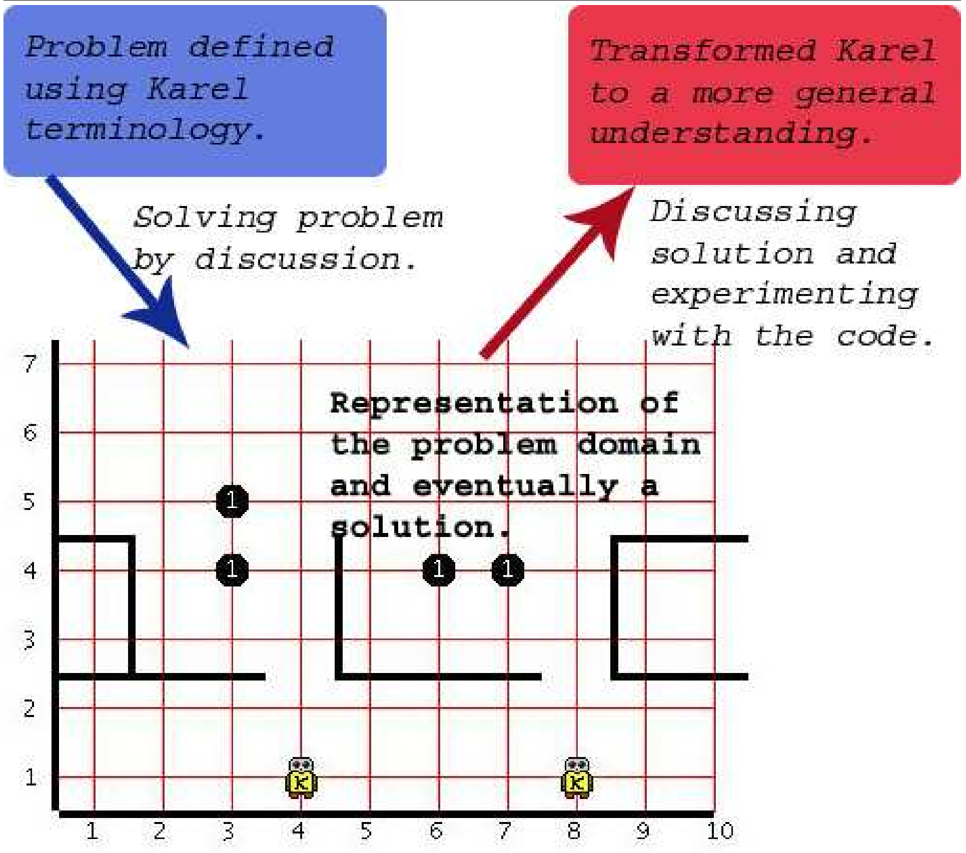
My whole thesis is based on my experiments and the observations that were made there. In Experiment 1 I used Robocode and in Experiment 2 both Karel J and Robocode were used.

7.2 Any OO theory learned?

I believe the best way to observe learning is to observe problem solving instead of asking students what they feel they know. In this chapter I will look at concepts one by one and see how they were presented and whether they were understood and used at all by students. In my experiments I had many conversations with students and this helped me get a better view of their situation.

As can be seen in figure 7.1, Karel played a vital part in the students' understanding of OOP in experiment 2. The blue part of the figure illustrates the observed situation during the experiment. The red part of the figure illustrates a situation we did not observe so well, but we hope was happening. We did see some evidence of experimenting with code, but it was not possible for us to see if Karel J was able to be transferred to general understanding of OOP.

Figure 7.1 Learning situation in experiment 2


Problem defined using Karel terminology.
Transformed Karel to a more general understanding.
Solving problem by discussion.
Discussing solution and experimenting with the code.
Representation of the problem domain and eventually a solution.
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8
9
10

However, when we introduced Robocode to the students in experiment 2, we did see that the students were able to transfer their knowledge from Karel J over to Robocode. This opens for the possibility that the students' knowledge from Karel J can be transferred to general OOP.

7.2.1 Abstraction, understanding the setting

As mentioned in chapter 2, abstraction is one of the main concepts in OOP, along with encapsulation, inheritance and polymorphism (discussed further down). Making the students understand how to sort out the important parts of the reality that is to be implemented in a program, is important no matter what sort of programming language one uses.

In both Karel J and Robocode, the abstraction of the world has already been done by their creators: The functionality of the robots/tanks has been reduced and limited to what is needed in the setting they were designed for. What the students must do, is relate to this abstraction and get a mental model of the environment. When the environment used is graphical, making a mental model is easier. Karel J is a good example of this: In Karel J, in addition to the graphical representation of the robot, the status of the robot is printed to the terminal in this fashion:

```
RobotID 0 at (street: 6) (avenue: 2) (beepers: 0) ( direction: West) on
RobotID 0 at (street: 6) (avenue: 2) (beepers: 0) ( direction: South) on
RobotID 0 at (street: 6) (avenue: 2) (beepers: 0) ( direction: East) on
RobotID 0 at (street: 6) (avenue: 3) (beepers: 0) ( direction: East) on
RobotID 0 at (street: 6) (avenue: 3) (beepers: 1) ( direction: East) on
```

If the terminal print-out had been the only thing the students had seen, they would have had a much harder time making a mental model of the environment. They would have to draw the robot on a piece of paper and trace their way through the print-outs.

In experiment 2, we as teachers also tried to make the students visualize the setting by telling them that the robots were created at a factory and they were given certain functions they could perform.

The students were comfortable talking about Karel in this setting. The following is a conversation between to students in Experiment 2:

> ”Where are we placing the robots?”
> ”On the corner of 2nd street and 1st avenue.”
> ”What way is he facing?”
> ”North.”
> ”He can pick beepers, can he place them out?”
> ”Try placeBeeper or putBeeper”
> ”He has to walk two north, then turn or he will crash into the wall.”

We see from this conversation that the students comfortably used Karel J specific words ”street”, ”avenue”, ”beepers”. These are all words that describe the Karel world in some way, but at the same time are words that the students are fairly familiar with outside a Karel setting. In using these words they are linking their understanding of Karels world with their own knowledge of the real world: Walls must be walked around, streets and avenues cross each other, beepers are things that can be picked up. Linking this understanding is a very good example of assimilation (mentioned in chapter 3), where the complex programming of Karel is linked to familiar concepts that the students know.

At the same time, one can see that the conversation above resembles pseudo code. Put in a Java-like program this would be:

Listing 7.1: Pseudo code from the conversation

```
/**
 * Make a Karel robot in pseudo code
 */
class RobotWalk {
  main {
      new ur_Robot in 1, 2 facing north;

      placeBeeper();
      putBeeper();

      north, north, turn left;
  }

}
```

After a little while, talking in pseudo code became very natural. In chapter 3, I speak about accommodation and assimilation. The way the students talked is a way of assimilation: The students talk about a more complex world (the world of programming) in a way that they recognize (a robot moving up and down streets).

7.2.2 Encapsulation

Encapsulation, a very important aspect of OO, was revisited many times during experiment 2, but also in experiment 1. In experiment 1 it was a bit more subtle, but the students saw each of the tanks on the battlefield as an individual entity or object with some functionality packed into it that could be reused. The students made some local methods and variables used by the tank when it was tested.

In experiment 2, the students made their own classes[1] with local

[1]They only made subclasses but they were treated as independent robots, see more next sub chapter

methods only usable by a specific robot. In Activity 3-2, where the students made a robot to climb stairs, they made a subclass with some functionality (**turnRight** and **climbStep**).

In G2, one of the students had some experience from BASIC programming, a procedural language. This knowledge of procedural programming dominated the group as he then became the one with most knowledge when it came to programming. Remembering listing 6.4 from chapter 6 and the example of conversation between members of G2 below, group 2 used a very imperative coding style. Much of the code was written in the main method as this was the way the student in G2 was most comfortable with.

> ”Damn, he won’t turn.”
> ”Maybe we need another while (loop).”
> ”Let’s try.”

Did the concept of encapsulation not sink in? Why not?

The reason for this can be a few. Bergin says in his article that the shift from the procedural paradigm to the object oriented can take a long time based on the person’s procedural knowledge [10]. The students in G2 did make subclasses with methods and did use them to a certain degree, but when the complexity became a bit higher, they slipped into a more familiar procedural way of solving the problem. Instead of following the object oriented way we had presented, they fitted the problem into a domain that was more familiar to them. The conception of how a program is supposed to be written (in a procedural way) was transferred over into the OO world. This is a form of negative transfer that could most likely have been avoided. Although Pea (see chapter 3) talks about negative transfer from daily life to coding, the same principle applies here: The student was familiar with something, saw a connection and used it. It can also be seen as an example of Constructivism (see also chapter 3): He builds his knowledge on something he already knows and applies old knowledge to new problems. As far as getting the program to work, he was right, but with respect to OOP, it was wrong and not illustrating the point of OOP.

If we had spent a longer time on each activity so that the students felt more comfortable with the use of OOP to solve a problem, they might not have fallen into the procedural paradigm when the tasks became more complicated.

To quote myself (chapter2):

> *The strength of encapsulation is that one has the opportunity to complete parts of the program with methods and variables and then put these parts away and only use the methods required later in the program.*

G1 in experiment 2, did use methods packed in a subclass and only called these methods when it was time to run the program (see code bit 6.3). The pay-off was very evident when they wanted to comment out one of the robots because it took too long to test: They only had to comment out two lines. It is quite evident that the use of encapsulation was positively reinforced with G1 when they saw this.

7.2.3 Inheritance

Inheritance, the use of subclasses was something that was covered in both experiments. There are different ways students can understand inheritance:

1. Subclasses are used to expand some class with new behvior in addition to the old behavior which is inherited by the subclass.

2. Subclasses are used to extend some general class (for instance an abstract class) and use the superclass' behavior modified a bit to fit a specific program (most likely by polymorphism).

3. Many subclasses extends a superclass (figure 4.2) and the superclass contains common data for all the subclasses (like name and license number).

In experiment 1, the students made objects that were subclasses of an abstract superclass. In the classes the students used methods defined in the superclass. The students' understanding of subclasses in Robocode would fit into number 1 in the list above.

In experiment 2, subclasses were covered the first day in Activity 3-1 (making the robot understand turnRight). We then told the students that the new class[2] was a new blueprint of a robot that had inherited the functions from the old robot. The students had some trouble understanding the use of constructors, but this will be explained more in-depth with example in a later sub chapter.

The students were able to write their own subclasses and understood the motivation for using them. Inside the Karel world, the students seemed very comfortable using subclasses, but a big question is whether they would be able to transfer this knowledge to a program independent of the Karel environment. When we tried the Robocode environment on the students, it became apparent that they were able to use their knowledge in a similar environment, and this is a strong indicator that the students would be able to remove themselves from the Karel environment entirely. Hopefully, the use of the Karel environment can work as

[2]We now started using the class word.

a tool for the students when making the shift to general OOP (see my notes on positive transfer in chapter 3).

As mentioned, in experiment 2 we also tried Robocode as a test to see how well the students managed a more complex environment after just about 8 hours of coding. The students understood how to make working tanks in Robocode as well, it was the size and complexity of Robocode that stopped them:

Listing 7.2: G1, Robocode (comments added by me)

```java
/**
 * Some of the code from G1: main run method
 */
public void run() {
  // Behavior of the tank,
  // Using methods defined in superclass Robot
  while(true) {
    ahead(100);
    turnRight(90);
    ahead(200);
    turnLeft(90);
    back(100);
    turnRight(90);
  }
}
```

As seen in the code, the students use the methods defined in the superclass Robot without much problem. It is when faced with all the multiple opportunities that they stop up and have a hard time sorting out all the information.

7.2.4 Polymorphism

Polymorphism was not covered in any of the experiments, but I see ways it could easily have been done.

Maybe we should have covered polymorphism instead of some of the imperative programming in experiment 2? I can say I felt drawn to introduce imperative programming so we could make the assignments more interesting, but this might have been the wrong approach.

The concepts mentioned above are concepts that are important for the understanding of OOP. Other concepts that come as a consequence of these four are discussed below.

7.2.5 Reuse of code

Reuse of code was illustrated very clearly in experiment 2. The students made subclasses that could be reused in later assignments. We did not specifically tell the students to reuse the classes from earlier, although we sometimes told them not to reinvent the wheel.

G2, the imperative group, did not reuse classes to the same extent as G1. They did reuse subclasses from earlier to aid them some with for example turnRight. But the overshadowing paradigm was the procedural. In our three day experiment we did not have much time to repeat and imprint the concept of reuse well enough. In a conversation from G1, regarding the final activity we observed. In the conversations students mostly referred to the karel robot as "he":

> "OK, so now he works, now what?"
> "The other does the same thing, just a little reversed."
> "So now we write that..."
> "No, let's just copy, I will show you."

The conversation over is a clear example of positive transfer mentioned in chapter 3: The students are faced with a task similar to something they have seen before and are able to apply earlier knowledge to this new problem.

7.2.6 Object generation and pointers

In OOP it is important to learn to use and work with objects both by generating one or multiple objects and using pointers. In Karel this starts right away. The first two lines the students see when starting with Karel are:

> **ur_Robot karel = new ur_Robot(1, 1, North, 0)** and
> **karel.move()**

One can say that this is like the **HelloWorld** of Karel, but instead of just writing a line of code on the screen, it captures much more. The students get introduced to their first object and use a method associated with it by using a pointer or dot-notation as it is also called. And it produces a visual result: A robot that moves in a map. This opens up for trying out some more lines like moving additional steps, then turning.

This was the case for our students. After they had been presented to these two lines, they started experimenting on their own and it was not long before all the groups had solved the first task of the course.

The students in experiment 2 did not fully understand pointers, they saw the references to the objects as objects. However, they were able to access methods in the objects. We never referred to the object names as references but the students still used the names as references with the dot notation.

7.2.7 Active objects

Robocode illustrates active objects very well. In this environment, instances of different tanks are created when they are spotted (see earlier chapter, example 4.2). This is very much like the first understanding of active objects mentioned above.

Karel J has no communication between objects built in the environment, so this type of communication is not easy to implement. It is possible, but it requires some coding. I think it would be a very big advantage for Karel if communication between objects had been simpler to implement. However, Karel offers the opportunity for the robot to adjust itself based on its surroundings:

Listing 7.3: Active Karel robot, written by me

```java
/**
 * Karel moving based on its environments
 */
class KarelMove implements Directions {
  public static void main(String [] args) {
    Walker karel = new Walker(1, 1, North, 0);
    karel.walk();
  }
}
class Walker extends Robot {
  // Constructor
  public Walker(int s, int a, Direction dir, int b) {
    super(s, a, dir, b);
  }
  // Karel walks until he finds a beeper, turning every time
  // he faces a wall.
  void walk() {
    while(!nextToABeeper()) {
      if(frontIsClear())
        move();
      else
        turnLeft();
    }
    pickBeeper();
  }
}
```

This type of example can easily become very procedure oriented and steer some of the focus away from OO, but this type of procedural programming has to come in at some point, and it is important then to put the methods in the objects instead of in main.

7.2.8 Constructor

As mentioned in an earlier sub chapter, the students had some problems understanding constructors the first time it was introduced. Because we did not introduce methods before subclasses, finding constructors difficult was a natural reaction as the students had not seen a method yet (with the exception of the main method, but that had not yet been discussed in depth). Looking at the code example above (7.3) the students had some trouble understanding the need for the constructor:

> Why do we have to write s, a, dir and b so many places? It seems so redundant.

To begin with we had some trouble convincing a few of the students of its use. The general explanation we used was that the constructor made the object unique and the **s**, **a**, **dir** and **b** is information that make that particular robot special and the constructor using letters and words (s, a, dir and b) was helpfully if we made more objects of the same class. They accepted that explanation, but most likely they did not understand it fully, at least not how it was called.

On the second day, we introduced an assignment where the constructor was expanded (activity 5), the conversation in G2 developed thus. Remember that the expanded constructor dealt with counting the beepers Karel had in his bag:

> ”OK, so where do we start?”
> ”Ummm, 1, 6. And north. And write 6 here.”
> ”If it says 6 here, he starts with 6 beepers.”
> ”Yeah...?”
> ”But he has 0 beepers. So it’s 0 here.”

As we see from this conversation they have started to understand what a constructor does and how parameters are passed to it by its call. I am quite impressed.

Constructors was the hardest part of the course, because it combines three elements: Methods, parameters (discussed below) and objects. When the students started understanding how constructors work, they started to get a good grasp of OOP.

7.2.9 Methods and parameters

As mentioned in chapter 2, many researchers and teachers have claimed that methods and parameters are very hard to understand.

There are different aspects to methods:

1. Calling methods without parameters.
2. Calling methods with parameters.
3. Making methods without parameters.
4. Making methods with parameters.

In experiment 1, the files that the students' code are subclasses and therefore do not need static. Most methods in Robocode, like turnLeft, move and fire, all take parameters. However theses parameters ”made sense” to the students: A method call like $move(200)$ looks like something one would say in a natural language, ”Move 200 units”. The students in experiment 1 were also used to using methods with parameters, so it was not hard for them to understand what was going on. Parameters were also quite easy to understand for the students in my experiments as most of the movement of the tanks are based on parameters.

In experiment 2, we waited with introducing methods until after we had introduced the students to subclasses, so we avoided the word static all together. We did not introduce methods with parameters because we did not introduce variables. However, looking at the sub chapter on constructors, it is obvious that the students had a grasp on how to use parameters.

As mentioned, teachers and researchers have claimed that methods were difficult for students. My experience in the experiments was that students had no problems with using methods. In experiment 2, the students used methods with and without parameters and made methods without parameters. In our experiments, methods were limited to the environments, using method names that came very natural. Method calls like $move()$ and $turnLeft()$ becomes words instead of method calls and therefore easy to apply to the program.

One might argue that outside these environments, methods might still be hard for these students. However, looking back at code listing 6.3, we see that the students have made well written methods and call them without problem.

7.2.10 Variables

The students of experiment 1 were used to the use of variables as they had had one semester of programming.

We never introduced variables to the students in experiment 2. They used variables briefly in one of the assignments (Activity 5) and it seemed like they had no problems using it there.

7.3 Ways of working and learning

In chapter 3, I mentioned some different methods or ways of working that are said to have a positive effect on learning to program. An interesting aspect of the experiment was to see if any of these ways of working were used by the students.

7.3.1 Pair programming and discussion

In experiment 1, the students all sat at individual computers and programmed, so there was no pair programming in this experiment.

In experiment 2, the most important reason for pairing up students was to be able to observe their conversation. We were unsure how well they would talk together, but it became apparent that we had nothing to worry about. The students communicated very well and discussed the assignment and Karel's movements with much enthusiasm. It is clear that Karel worked as a catalyst for these conversations. Having something that is visual to talk about helped the students:

> ”OK, so Karel walks up here.” (points at the screen)
> ”Then we know he is facing north.” (points up on the map)
> ”Yeah?”
> ”Then we have to turn to the right once and start the labyrinth.”
> ”Yeah.”

The students were also very good at working as equals, no one taking more control of the team than the other:

> ”Do you have any ideas?”
> ”Not really.”
> ”Me neither really, but... (explains a possible solution)
> ...what do you think?”
> ”Sure, let's try.”

Did we see any other benefits apart from good conversations?

We focused on two groups mainly and they were both programmed in pairs. There was one student that sat alone one of the days, but he had many conversations with us. I am not sure if the fast progress of the groups can be entirely contributed to pair programming, but I am sure that it was helpfully for the students to have someone of their own skill level to talk with.

As mentioned in chapter 3, pair programming leads to better quality of code, fewer bugs and faster production of code. There are numerous examples where one of the group members corrected the other when he typed something wrong. The students corrected each other, discussed and produced good code in a relatively short time. Not all of the code held the same quality, but all in all, I must say that it was pretty good.

We saw clear signs of pair programming in this groups, as can be seen in the conversations above. The benefit of being able to discuss problems with another person in the same situation as oneself is clear. Pair programming and discussion is a good example of contructivism: The individual learning through interaction with the environment and other people in the same situation plus building his experience on others' experience. In a study on the benefits of pair programming in a beginners course in programming the following was reported [14]:

> *Students reported that they benefited from being exposed to their partner's ideas and suggestions, and that they therefore broadened their understanding of the assignments' requirements. The students indicated that it was easier and quicker to complete their work and there was an overwhelming belief expressed that it helped them identify errors more readily and consider alternative approaches to problem solving.*

The results corresponds with what we saw in our experiment. On day three, two students started out alone and spent about 20 minutes getting nowhere. After that time we paired them up:

> "Have you gotten far?"
> "Yeah, well, no not really, just been fiddling around. You?"
> "Not very far no."
> "I think we are supposed to..."

And from there the discussion went on with many suggestions for solution and a much quicker pace in solving the problem. It is quite clear that sitting in pairs helped the students in Experiment 2 get more out of the experience and learn more.

7.3.2 Playing a game

In experiment 1, the game factor was high: Robocode is based on winning over other tanks in a battlefield. Was programming overshadowed? Did Robocode illustrate what it was supposed to?

When the students in the experiment wrote their programs, two of the students, knowing each other started a friendly bet that one was going to beat the other. This inspired them both to make the best robot and therefore used all the help they had: The API, looking at other robots and theory learned in class. The students did not make this robots thinking Java, but they still learned coding techniques such as reading the API and understanding code written by other programmers. Without actually thinking about learning any code, the students did learn to code.

From conversations with the students, it was apparent that the students viewed each code as a separate objects:

> Me: ”What if you had to make a file for the battle, how would you get all the tanks in the battlefield?”
> S: ”Make an array of the tank objects with 'new' and then the name of the robots?”

Robocode did illustrate the communicating objects for the students and they were able to make some interaction between the tanks. Given more time, the students would have made good tanks with many features.

In experiment 2, the main focus was on Karel J. Karel J also resembles a game, but in another way than Robocode. In Karel J each assignment is a small game or puzzle, like climb a stair or find a beeper in a maze.

In chapter 3, I presented MOOSE Crossing, a game where children could learn a simple scripting language in a virtual setting. The particular story I mentioned tells about one thirteen year old girl helping twelve year old girl to learn this scripting language through cooperating and playing around in this virtual world. We see some of the same examples in experiment 2 where two students help each other to learn by discussing and playing around.

7.3.3 Testing and running code

Both Karel J and Robocode are visual environments that give a clear representation of the code written. This encourages the students to test their code often, thus seeing the result of their code.

Karel J's representation of the code runs very slow for a reason (to give a representation of the code that is slow enough for the students to follow). This makes Karel J slow to test if the program is big and tend to put the students off testing too much. To take one example from G1 (experiment 2):

> ”This is taking so long. We know this part works.”
> ”OK, let's comment out these two lines and place the robot there.[3]”
> ”No, it's not facing that way.”

Here we see the students using a good method for testing the parts of the code they were unsure of.

[3]The students commented out two method calls and moved the karel robot to the coordinates they knew it would end up when those two methods were done.

In experiment 2, G1 tested more seldom than G2. This was because they did the whole assignment on paper before starting to code, so they had great confidence in their solution. Because of this great confidence they saw no need to test a lot along the way.

Still, Karel J gave an output that could be understood by the students: If the robot suddenly was facing the wrong way, the students could easily backtrack through their code and see where things went wrong. Removing one line and adding one line also gives clear output to the students and help them see better the effects of one line of code. As I said in chapter 3 about the same matter:

> This is a good way of learning how programs works and gives very good hands-on experience that is invaluable in learning to program.

The students also read through some of their old code, trying to use old knowledge to solve a problem:

> ”This problem looks like the one we did last time.”
> ”Find the program we did last time, what did we do there?”

This can be seen in parallel to reuse of code, discussed above.

In experiment 2, the students had a good chance at seeing what went on in the code. When the robot crashed into a wall, the students had little problem finding out where in the code things went wrong:

> ”Hmm, he isn’t facing the right when he reaches the top.”
> ”OK, we just put in a turn command to fix it.”
> ”About there then (points), that’s where he reaches the top.”

7.4 The good progress in experiment 2, all us?

The progression of the groups in experiment 2 was beyond all expectations. A good question to ask oneself is: Was this progress all thanks to Karel J and our way of structuring the course?

Karel J and our structure might have had something to do with this. One thing helping the progress was the structure of the assignments, starting very simple but still complicated enough to illustrate OOP very early on. This is a very important point: Course administrators using objects first all agree that a sufficiently complex first example is needed to illustrate the basic concepts of OOP, a thought that has been advocated by Kristen Nygaard and later a part of the COOL philosophy[4]. In Karel J,

[4]http://heim.ifi.uio.no/~kristen/FORSKNINGSDOK_MAPPE/F_COOL1.html

the first example is complex enough to show the most basic of OOP (using objects and pointers to methods) but is easy enough to understand[5]. At the same time, this first example is a good start for building more complex examples.

In addition to pair programming, discussed in depth above, there were other factors that could have affected the progress.

7.4.1 Selection of students

One factor that played in was that these students had volunteered for this experiment. This means that they had a special interest for programming and were very motivated to learn. The students' interest for computers also helped the experiment start off faster than what we would normally have expected. The fact that the students volunteered may also have meant that these students were very outgoing and easy to connect with, making it easier for them to cooperate about the assignments.

In large programming courses, students attend for different reasons: Some of them might need the course, others are curious, while others are very interested. The biggest challenge for lecturers is the first group of those who are not motivated at all. In Experiment 2 we did get students that were interested in computing. It is harder to get students that are not interested to volunteer and therefore the results of the experiment will be skewed. The course we used in experiment 2 should be tested on students that are not motivated or have much knowledge of computers.

7.4.2 Size of the Class

Another important factor was the size of the class. With so few students as we had, it was much easier to follow up on each of the students and spend time with each of them. It was impossible for them to hide their problems when each of the students were asked questions that they had to answer. In a larger class, the course instructor will have problems to following the progress of each student as well as for a small class and there is a greater possibility that some students might be left behind.

Another advantage of having a small class, is that a small class offers a more social environment. In this environment, the students can discuss among each other and share aspects of the assignments they have solved (problems, surprises, solutions and so on).

An investigation into the size of classes[6] reports on the benefits of reducing class sizes in lower grades schools and that among other things it leads to higher academic performance.

[5]I can only speak for the students in Experiment 2, of course.

[6]http://www.asu.edu/educ/epsl/EPRU/articles/reducing_classsize.htm

7.5 Structure of the coursework

One factor for the good progress can be contributed to us: The structure of the coursework and the tasks given. We took great care when designing our tasks and the maps, not to leave the possibility for many different problems to arise at the same time. This was an important reason why the learning curve was so good. We always made sure that the students would not suddenly be faced with any difficulties they had no qualification for knowing.

There was one concept we forgot to introduce, and that was that of negation (the use of ! in front of a logical expression). The result of this was that the students had much more problems with one assignment than if they had known this from the start.

All in all, we managed to keep the facing of multiple problems very low.

We did however, encounter a small problem on day 2 as the students seemed to have forgotten some of the material they had been taught the day before. We should have repeated more of the previous days' material to allow the students to recall and better remember what we wanted them to have learned.

7.6 2 days of Karel vs. 3 months of traditional OOP

I had my expectations before the projects started, and some of them came true, but there were some surprises as well.

The biggest surprise for me was the progress of the students in experiment 2. I had not expected them to be able to manage so much in those short days. We had set the pace for starting day 2 with subclasses. When we started subclasses halfway through day 1, I was surprised indeed, and at the same time pleased. It showed that Karel J had some potential and that our structure of the course had worked. Above, I have discussed reasons for this, and although part of the reasons for our success lies outside Karel J, I am of the conviction that Karel J played a big part in our course and therefore in the good results of the course.

In experiment 1, I had expected the students to perform better with Robocode. However, it seems that Robocode is a bit more complex than I thought. The students (both in experiment 1 and 2) understood the basics of Robocode but drowned in the many possibilities.

That was another surprise: The students from experiment 1 did not do much better in Robocode than the students in experiment 2, despite the fact that the students in experiment 1 was at the end of their first semester. This was not expected, but I do not want to jump to conclusions about it. Robocode does look a little bit like Karel J with its movements commands and the threshold from making the tanks in Robocode just move around to make them intelligent is quite high. Many questions are raised here that I can not answer but would like to mention:

1. **Were the topics covered in experiment 2 coincidentally the topics needed to understand Robocode?** This might explain why the students in experiment 1 and 2 were so similar in their use of Robocode. If this is the case, it leads to the next question:

2. **If both Karel J and Robocode require the same kind of knowledge - this being much OOP knowledge, since that is what we covered - had the students from experiment 1 been taught too much "un-OOP like" programming or are Karel J and Robocode just unrealistic reflections of what sort of OOP knowledge is important?** I do not feel I can comment too much on it. Maybe with some more studies I can answer this question better.

3. **Were we particularly lucky with the students we picked out for experiment 2?** This is probable, and something I discussed a little above.

4. **If I had waited another two weeks before recruiting for experiment 1, would the students have come over the hardest threshold in Robocode and been able to program very good tanks?** Learning to program often goes in leaps at the time when suddenly certain "light bulbs" lights in the heads of the students. The students in experiment 1 might have been on the verge of a better understanding of OOP and therefore would have clearly separated themselves from the students in experiment 2.

Below follows one code from each of the two experiments. They are quite short, but that is just as much a reflection on the complexity of Robocode and the amount of reading and understanding needed before starting to code more serious. In both experiments the students only spent a couple of hours on this environment.

Listing 7.4: Code from Experiment 1, just the innards of the class is displayed. This is one student with about 3 months of programming experience.

```java
public void run() {
  while(true) {
    // Replace the next 4 lines with any behavior you like
    turnGunLeft(360);
    }
}
```

```
 7  /**
 8   * onScannedRobot: What to do when you see another robot
 9   */
10  public void onScannedRobot(ScannedRobotEvent e) {
11    double d = e.getDistance();
12    x = e.getBearing();
13    turnRight(x);
14    ahead(100);
15    //kjøre mot fiende
16    if (d<200) {
17      while(d<200) {
18        turnGunRight(e.getBearing());
19      }
20    }
21  }
22  /**
23   * onHitByBullet: What to do when you're hit by a bullet
24   */
25  public void onHitByBullet(HitByBulletEvent e) {
26    ahead(50);
27    turnGunLeft(e.getBearing());
28    fire(30);
29    back(50);
30  }
```

The student is starting off with trying to calculate the distance and bearing to the other tank (line 11 and down). He only handles two events, *ScannedRobotEvent* and *HitByBulletEvent*. He starts with using *if* and *while*, but I am not quite sure where he is going with this. The students shows that he has a grasp on the different methods for communicating with other robots, but there is something lacking. With a little more time this could have been a promising tank.

Listing 7.5: Code from Experiment 2, just the innards of the class is displayed. This is the code from group 1 in the experiment at the end of day 2.

```
public void run() {
  while(true) {
    // Replace the next 4 lines with any behavior you like
    ahead(100);
    turnRight(90);
    ahead(200);
    turnLeft(90);
    back(100);
    turnRight(90);
  }
}
/**
 * onScannedRobot: What to do when you see another robot
 */
```

```
15 public void onScannedRobot(ScannedRobotEvent e) {
16   turnGunRight(360);
17   fire(10);
18 }
19 /**
20  * onHitByBullet: What to do when you're hit by a bullet
21  */
22 public void onHitByBullet(HitByBulletEvent e) {
23   turnRight(e.getBearing());
24 }
25 public void onHitWall(HitWallEvent e) {
26   out.println("Ouch, I hit a wall bearing " + e.getBearing() + " degrees.");
27   turnLeft(180);
28   ahead(100);
29 }
```

The students are trying to get the hang of the events. They are starting to show promise in the last method *onHitWall*. It is clear they have understood the use of movement, a parallel they have drawn from Karel J. We see that the students have put more in the *run* method (lines 1 - 11) and this is probably because of earlier experience with Karel J. The students are not quite comfortable with the methods that are offered by the RoboCode API and tries to dictate more of the movement of the tank instead of letting the tank react to events.

Chapter 8

Conclusion

As stated in the introduction, the focus of this thesis has been to examine the use of graphical environments for supporting the teaching of object oriented programming and with a main focus on Java. The conclusion here is supported by two experiments, one with Robocode and the other mainly with Karel J.

Along the way, I have looked at other aspects of teaching OOP that are important: Mainly pair programming and an objects first approach. These aspects were not the main focus of my thesis, but I have seen the importance of them as I have conducted experiments.

I will go through the most important aspects and make some concluding remarks about them.

My original research question was:

> **In an Objects First approach to learning OOP and specifically Java, will the use of graphical environments be a good tool?**

The experiments indicate that:

> If students are to learn OOP using the objects first approach, the use of graphical environments as a support will help this approach greatly. Collaboration between students and a well structured course work without too many problems introduced at once, is vital for a successful introduction to OOP.

8.1 The Experiments

The main drawback of my experiments is the lack of participants. There were too few subjects to observe to make any proper conclusions, but I saw many indications that the graphical environments had a positive influence on learning.

For experiment 1, I see I should have planned it better and allowed more time for people to participate. Despite this, the experiment went well and I could observe I wanted: How the students interacted with Robocode and how well they understood it. And whether the students learned anything from the experience. Robocode encouraged students to play around with the code and it seemed to help them visualize the movements of the tanks and how their code affected this.

Experiment 2 went very well, but there are changes I would like to make if we were ever to conduct another experiment like this again. First of all, it would be best to have more time or to use some of the time available to repeat many of the concepts we introduced. The subjects covered in the course can be easily forgotten as we did not spend much time repeating and making sure everything had stuck. Polymorphism was never covered. Polymorphism is an important aspect of OOP and we should maybe have covered this subject as well. Still, we were able to cover the most basic of OOP, as mentioned in the discussion and the experiments and after this we introduced some imperative programming instead of polymorphism. In order to make a bit more advanced programs and assignments it was necessary to introduce imperative programming to have something to put into the methods in the subclasses. We could however have repeated more about objects and subclasses, but we did the right thing in choosing imperative programming in stead of polymorphism.

The structure of the assignments was also important. As mentioned in chapter 7 we used the same structure on the course as Bergin suggested on his website. Bergin, in his approach introduce polymorphism before imperative programming. We chose not to focus on polymorphism, due to limited time. The structure in broad terms was as follows: Make one object, make multiple objects, make subclasses, some imperative programming.

To summarize what the students learned from this experiment, I would say they learned many new concepts. They got an understanding for OOP, learning about objects and making their own objects before learning more imperative coding.

The students learned to discuss assignments and that it helps to plan ahead and test their programs often; all good qualities to have when programming.

Karel J helped us as lecturers to develop a well structured course program because we had a framework that was easy to understand and handle to build our tasks and lectures around. However, I am sure that other graphical environments could have done the same job, provided they were easy to use and not too complex when it comes to functions. Robocode is an example of an environment that is complex in that the tanks have many functions for each tank, such as $hitByBullet$, *onHitWall* and so on. When we tried Robocode in experiment 2, we saw that the students had no problem transferring their knowledge of Karel J over to Robocode, but the progress soon stopped due to the sheer size and multiple possibilities that Robocode offered.

Pair programming, size of the class and the type of students we had also played a part in this. With a little better time, I am certain we would have managed the same good result with 20 students as well. I saw this as a very good and desirable learning situation when learning OOP.

8.2 The environments

I chose to focus on BlueJ, Robocode and Karel J because they offer three different views on OOP: Models of objects, active objects and simple graphical representation of basic OOP. There are many other environments that could also have been used, but here I see a problem: Most of the environments developed are not explored well enough in an educational setting and many of them are only tried once on a local institution without anyone ever hearing about it. Many would benefit from a deeper research into the effects of certain environments. On this field BlueJ is the one most explored; much literature exists on this environments. That is one of the reasons I have not looked too closely at BlueJ, other environments needed more research.

Robocode is a very engaging and fun environment that encourages playing with code. It also visualizes objects as independent objects that interact with each other. From the experiments conducted, it was apparent that students had some problems with overcoming the size and amount of features that Robocode offered. From this it is evident that Robocode would not be the best place to start introducing programming. Provided that the course structure of a beginner course has been an objects first approach, Robocode would be a good tool to introduce when the students were moving over to imperative programming, use of **if**, **while** and **for**. The students showed that they had problems following everything that happened. This is because the tanks in Robocode has very fast continuous movement that is hard for the students to observe.

Karel J works very well for an objects first approach to programming. Many argue that the best way to introduce OOP to beginner students is to circumvent the use of the *main* method. Karel J does introduce the *main* method right away, but we found in our experiment that we had no problems explaining the real use for the *main* method after only a few hours of programming. Karel J offered a good setting for explaining OO concepts and giving us as tutors a framework where it was easy to explain OO concepts. We are able to right away start with OO concepts and let these be reinforced many times over the course of the experiment while teaching new topics that could be attached to the old knowledge. This is consistent with what Bergin writes about Karel J in [9]:

> One of the main advantages of the Karel philosophy, however, is that it gives the students simple versions of a large number of tools that enable sophisticated problem solving. Thus, Karel forms the first cycle of a Spiral teaching approach, and students will have the ability to successively deepen their knowledge of many topics while having the tools to build exploratory programs from the very beginning of the course.

BlueJ offers an approach to OOP that focuses on modeling the world and letting the students plan the program they are about to write in detail. This is something that few beginner students see the value of as their programs are too small for this sort of planning to be an obvious benefit. Careful planning of a small program will only be viewed as a cumbersome and time consuming task.

It is possible to combine Karel J with BlueJ. This could be a union that displays many sides of OOP [1]. An objection to this union is that two environments can be a lot to learn in one semester.

8.3 What was hard and what was easy?

Looking at the two experiments and environments, what was hard for the students and what was easy?

- **Understanding objects:** In both Karel J and Robocode, it was easy for the students to see the objects present and understand how they worked. They saw that the objects had certain properties and methods and that these objects could work independently of each other. In Robocode the aspect of objects as active entities affecting each other was also clear.

- **Understanding methods:** In both Karel J and Robocode, methods were used. The students in Experiment 1 had much experience with methods. The students in Experiment 2 had no problems making methods without parameters and reusing methods in Karel J. In Robocode it was a little more complicated, but they managed to ”scratch the surface” of methods there as well.

- **Understanding constructors:** Constructors is only covered in Karel J. Although constructors are similar to ordinary methods, there is the added complexity of when the method is called and usually

[1]http://www.csis.pace.edu/~bergin/KarelJava2ed/usingKarelJRobot.html, bottom of the page.

with parameters. The students had some problems with constructors and seeing the use of it. One of the reasons for this can be that we used too few objects for the constructor to become apparent.

- **Reuse of code:** The students in Experiment 2 were good at reusing code, even though they sometimes wrote things from scratch when they had written the same thing before. They became better with some training and more experience.

8.4 Top ten reasons, revisited

In the introduction, I mentioned a list of top ten reasons why OO should not be taught in CS1, I would like to list these and comment on them based on my personal experience. Some of the points mentioned in the article [17] are outside the areas of my research but I will still mention them, but I will not comment on them.

1. **OOP is just a fad!** When this article was written in 1994, OOP was gaining popularity and many thought this popularity would fade. Now, in 2003, it is quite clear that OOP is not just a fad.

2. **OOP is too hard for my CS1 students!** The experiments conducted in this thesis clearly show that OOP can be learned in two days. A good environment and well written tasks made OOP understandable for the students in the experiments. In experiment 2 we were able to give the students an understanding for OOP and had a good foundation to build on.

3. **The dreaded paradigm shift!** This point was concerned with the trouble of making the transfer from procedural to OO. This will not be a problem for students not familiar with any of the two paradigms, but for students already familiar with procedural programming. In experiment 2, one student had some experience from using Visual Basic. We clearly saw that when solving the largest of the assignments (appendix) the student (and his partner) used a much more procedural oriented approach. However, some of the methods were put in subclasses so he was not a stranger to the idea of OOP, it was just very easy for him to fall back into a more known procedural approach when things got harder. After only 2 days of work this must be expected, but he and his partner showed that they were on the way of making the shift quite easily.

4. **You still need algorithms!** No matter what kind of language on programs in, one must learn to make algorithms and use them properly. The article was concerned with how the use of algorithms

would fit into the Object Oriented paradigm. In experiment 2, the students put simple algorithms into subclasses and saw the robots execute these algorithms. The students learned simple algorithms and were able to place them in the correct classes, giving them a clear sense of OOP and algorithms.

5. **The ”OOP-ish” overhead!** A main concern was the amount of formalism explaining OOP, spending many pages in a book explaining the theory before getting to the code. In a graphical environment such as for example Karel J, there is no need for many pages of theory when much of the theory can be explained through showing the robots moving.

6. **OOP languages are ugly!** This point was very C++ specific and I will not comment on this here.

7. **There’s already too much material in CS1!** Learning OOP adds many concepts that must be taught in addition to program statements. However, in experiment 2, the students had no difficulty getting both a taste of imperative programming and learning about objects in a little over two days. In addition, it seems that the author perceives OOP as an ”extra feature” to add to a programming course. This is a misconception, an object oriented language is not a procedural language with some objects flying around. A course in OOP must therefore be a course in OOP, not procedural programming with an added feature.

8. **It screws up the rest of our curriculum!** Is this the reason why many educators wait with introducing objects in today’s Java and C++ courses? Because there is the fear that once the curriculum starts down the path of objects, most of the term will be spent explaining this instead of showing any code? In experiment 2 it helped us much that the students got an early understanding of OO. In this way we did not have to hide many of the long wrapping names (public static...) in a veil of magic.

9. **It fits perfectly with CS2 and data structures!** I am not entirely inclined to disagree. OOP best shows its strengths when the students start building larger programs. For simple CS1 programs, OOP might not be the best. However, [10] reports that programmers used to the procedural approach paradigm use a long time making the shift. This is a good argument for starting with OOP as students in a shorter time period is expected to learn OO and therefore do not have a whole semester to spend on making the shift. In addition, the main languages used in CS1 today are object oriented and the course administrators must take the consequences of this

choice and structure the CS1 course based on an object oriented approach.

10. **OOP is too hard for us!** The main concern was that switching to OOP would require much work from the lecturers as they were not familiar with OOP. This did not apply to us.

8.5 Are graphical environments the ultimate answer?

I have looked at three environments and tested two of them. It is obvious that the benefits of using graphical environments are many:

- A graphical representation of code.
- Changes in code are clearly seen.
- Students can live more inside the code and act with the environment.
- Educators have visual examples to hang their theory on.
- The environments are often small worlds with their own stories where it is easy to use analogies to illustrate code.
- Many complex aspects of programming are hidden so that the students can focus on using the functionality offered and then dive deeper into the matter later.

So why not just use some graphical environment since it solves all problems? There can be some problems with graphical environments as well:

- It can be hard to design an environment that illustrate the concepts that are important for programming, and especially OOP where the design phase is very important.
- If the environment is too complex it will take very long to learn.
- If the environment is too ”game-like”, it might overshadow the purpose of the aim: Learning to program.
- If the environment is too far removed from real life, it can be hard for students to move over to the domain of general programming.

The final point mentioned is a crucial one and something to be aware of. The environment might work very well for teaching students to program in that specific environment, but it might be impossible for students to move outside this environment.

The benefits I listed are of course dependent on a good design of the environment. A poorly designed environment with no thought of how students think or how concepts should be illustrated will just confuse.

In addition I want to mention that a graphical environment can never take completely over in the teaching process, but more work as an aid for the teacher and students. A well constructed course with good assignments, texts and lectures will always be important.

In my introduction, I looked at an article concerning BlueJ ([5]). In this article, the author present BlueJ as a helpful tool to support an Objects First approach. This is an example of how a graphical environment should be used: To be a tool that both the students and the lecturers can use to get a better illustration of the theory presented.

Graphical environments can help reduce the areas the students have to focus on in an educational setting (Introduction chapter). The students in experiment 2 had few problems getting right to the core of the problem solving and programming. One reason for this can be that the students were very familiar with the use of computers and therefore managed this transition quickly. Another cause of the quick transition may be that the students were introduced to an environment where there were fewer areas the students had to turn their attention to and therefore they could focus on the programs. We saw that they sometimes got a little confused by the problem with names of files (covered in chapter 6), so there was still some "noise" that distracted the students. But all in all, using Karel J helped the students focus on the code and it helped us as teachers in the way that we did not have to spend a long time removing "noise".

8.6 Future work

It would be interesting to conduct another experiment like experiment 2 with more students and more time. There are some changes I would like to try, and I am confident that with small changes, our course plan could be a very good one. The ultimate would of course be to have a whole semester at my disposal, but I think that would be hard to get the opportunity to do.

Based on the knowledge I have acquired during the work on this thesis, it would be interesting to make my own environment and try that out on a group of students. I must admit that many thoughts of such an environment have buzzed in my head and if I had the time, I would tried to shape such an environment. A new environment, coming from the same people that developed BlueJ, called Greenfoot, is under development. It is a 3D version of Karel J with many other features, and from what I have seen so far it resembles what I have envisioned as a good environment, so I guess there is no need for me to make something of my own now. I await the arrival of Greenfoot with excitement and hope it is as good as I expect.

There is a need for more comprehensive research into the teaching of OOP to find a framework consisting of one or more graphical environments that tutors can base their course structure on. I have enjoyed doing this kind of work up to now, and it would be fun continuing this kind of work.

At the 7th workshop for pedagogies at ECOOP 2003, everyone agreed on the Objects First approach, and this is encouraging. I see that there are strong roots in the way programming should be taught, but it is my hope that the trend is now shifting more and more to wards a new approach and maybe the use of graphical environments is a helpful tool for using this approach. It is my hope that this thesis can shed some light on this question.

Bibliography

[1] J. Atherton. Learning and teaching: Assimilation and accommodation (on-line). URL:
http://www.dmu.ac.uk/~jamesa/learning/assimacc.htm.

[2] D. E. Avison and M. D. Myers. *An Introduction to Qualitative Research in Information Systems*, pages 3–12. Qualitative Research in Information Systems, Sage Publications, 2002.

[3] D. J. Barnes and M. Kölling. *Objects first with Java - A practical introduction using BlueJ*. Prentice Hall, Pearson Education, 2003.

[4] R. L. Baskerville and A. T. Wood-Harper. *A Critical Perspective on Action Research as a Method for Information Systems Research*, pages 129–145. Qualitative Research in Information Systems, Sage Publications, 2002.

[5] J. Böstler, T. Johansson, and M. Nordström. Teaching oo concepts - a case study using crc-cards and bluej. *Proceedings of 32nd ASEE/IEEE Frontiers in Education Conference*, 2002.

[6] B. W. Becker. Teaching cs1 with karel the robot in java. *ACM SIGCSE Bulletin*, 2, 2001.

[7] I. Benbasat, D. K. Goldstein, and M. Mead. *The Case Study Research Strategy in Studies of Information Systems*, pages 79–99. Qualitative Research in Information Systems, Sage Publications, 2002.

[8] O. Berge, R. Borge, A. Fjuk, J. Kaasbøll, and T. Samuelsen. *Learning Object Oriented Programming*, pages 37–47. Norsk informatikkonferanse NIK'2003, Tapir Akademisk Forlag, 2003.

[9] J. Bergin. Introducing objects with karel j. robot. URL:
http://www.csis.pace.edu/~bergin/karel/ecoop2000JBKarel.html.

[10] J. Bergin. Why procedural is the wrong first paradigm if oop is the goal. URL:
http://csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html.

[11] Birtwistle, O. J. Dahl, and K. Nygaard. *Simula Begin*, page 47. Studentlitteratur, Lund, Sweden, 1973.

[12] S. Booth. *Laerning to program. A phenomenographic perspective.* Acta Universitatis Gothoburgensis, 1992.

[13] R. Borge and J. Kaasbøll. What is oo first? URL: http://www.intermedia.uio.no/cool/docs/ECOOP-2003_B&K.pdf.

[14] J. C. Brougham, S. F. Freeman, and B. K. Jaeger. Pair programming: More learning and less anxiety in a first programming course. URL: http://gemasterteachers.neu.edu/resources/pairprog03.pdf.

[15] S. A. Conger and I. Vessey. Requirements specification: Learning object, process and data methodologies. *ACM Comm.*, 5, 1994.

[16] O. J. Dahl and K. Nygaard. How object oriented programming started. URL: http://folk.uio.no/kristen/FORSKNINGSDOK_MAPPE/F_OO_start.html.

[17] R. Decker and S. Hirshfield. The top 10 reasons why object-oriented programming can't be taught in cs1. *ACM SIGCSE Bulletin*, 3, 1994.

[18] Jeremy Gibbons. Structured programming in java. *ACM SIGPLAN*, 4, 1998.

[19] L. J. Harvey and M. D. Myers. *Scholarship and Practice: The Contribution of Ethnographic Research Methods to Bridging the Gap*, pages 169–179. Qualitative Research in Information Systems, Sage Publications, 2002.

[20] J. Kaasbøll. *Exploring didactic models for programming*, pages 195–203. Norsk informatikkonferanse NIK'1998, Tapir Akademisk Forlag, 1998.

[21] M. Kölling and J. Rosenberg. Blue - a language for teaching object oriented programming. *ACM SIGCSE Bulletin*, 2, 1996.

[22] J. Kölling and, M. Rosenberg. Bluej - the hitch-hiker's guide to object orientation. *Technical Reports 2002, University of Denmark*, 2, 2002.

[23] R. R. Kessler and L. A. Williams. All i really need to know about pair programming i learned in kindergarten. *Communications of the ACM*, 43, 2000.

[24] S. Papert. *Mindstorms: Children Computers and Powerful Ideas.* Basic Books, Inc., 1960.

[25] R. D. Pea. Language-independent conceptual ”bugs” in novice programming. *Journal of Educational Computing Research*, 2, 1986.

[26] S. Schaub. Teaching java with graphics in cs1. *ACM SIGCSE Bulletin*, 2, 2000.

[27] M. C. Temte. Let’s begin introducing the object-oriented paradigm. *ACM SIGCSE Bulletin*, 2, 1991.

[28] P. van der Linden. *Just Java.* Aner ikke, 1999.

[29] A. Zeller. Making students read and review code. *ACM SIGCSE*, 32, 2000.

Appendix A

Robocode handout

On the following pages are the document that was handed out to the students in experiment 1.

Intallere:

Gjør følgende:

```
¿ mkdir robocode
¿ cd robocode
¿ cp  richared/robocode-setup.jar .
¿ java -jar robocode-setup.jar
¿ ./robocode.sh
```

Dokumentasjon

Dokumentasjon finnes på
http : //robocode.alphaworks.ibm.com/docs/robocode/index.html,
men her er et sammendrag av noen enkle funksjoner:

Funksjoner som tilhører Robot

Disse funksjonene tilhører Robotklassen. Det vil si at dere kan kalle disse rett i programmet deres, siden deres robot er en subklasse av Robot og har derfor arvet all disse metodene.

ahead(double strekning)

Tanksen din flytter seg sa langt rett frem som er spesifisert i strekning.

Eksempelvis: $ahead(300)$ vil føre til at tanksen deres kjører fremover 300 (her er strekning gitt i 'steg' relativt til spillebrettet)

back(double strekning)

Det motsatte av ahead.

fire(double styrke)

Skyter en kule med en gitt styrke.

Eksempelvis: $fire(3)$ skyter en kule med styrke 3.

turnLeft(double grader)

Snur tanksen deres et gitt antall grader til venstre.

turnGunLeft(double grader)

Snur kanonen på tanksen deres et gitt antall grader til venstre.

onScannedRobot(ScannedRobotEvent ev)

Din tanks ser en annen tanks (i denne metoden blir da denne tanksen betegnet med en *e*). Du får i denne metoden mulighet til å programmere hva din tanks skal gjøre med dette. ScannedRobotEvent har en del metoder, som kommer under.

onBulletHit(BulletHitEvent bh)

Du treffer en annen tanks (med en kule som her har fått betegnelsen *bh*). Du får nå en mulighet til å programmere hva du skal gjøre når dette skjer. BulletHitEvent har en del metoder som kommer lenger ned.

ScannedRobotEvent

Hver gang din tanks ser en annen tanks, kalles denne metoden: $onScannedRobot(ScannedRobotEventev)$. I dette tilfellet fikk tanksen du så navnet ev. Nå kan man kalle en mange metoder ved å si:

$ev.metodenavn$. Metodenavnene kommer her:

getDistance()

Returnerer avstanden til den andre tanksen i forhold til din egen. La oss for eksempel si at du vil krasje med den andre tanksen (husk den andre tanksen har i dette eksempelet navn ev, se over). Da kan du skrive:

$ahead(ev.getDistance() + 1)$. Dvs. kjør din egen tanks så langt som avstanden er + 1 slik at du krasjer.

getEnergy()

Returnerer livet til tanksen du har fått øye på.

BulletHitEvent

Hver gang man treffer en tanks, kalles denne metoden (dersom du velger å definere noe i den), f.eks.:
$onBulletHit(BulletHitEventbh)$. I dette tilfellet fikk eventet (den andre tanksen) navn bh. Nå kan man kalle en mange metoder ved å si:

$bh.metodenavn$. Metodenavnene kommer her:

getName()

Returnerer navnet på tanksen du traff. Eksempelvis:

$System.out.println(“Ihit“ + bh.getName() + “!!!“)$. Ikke bruk System.out.println (dette var bare et eksempel, bruk egne meldings metoder.)

Noen andre småting

Metodene over var bare noen eksempler, alt står på webadressen gitt (dette gir forresten glimrende trening I å lese en API!). Her følger noen ting dere burde kunne, men som jeg tar med så dere slipper å sitte å slå opp i boka deres i tilfelle :)

if-tester

If-tester brukes for å sjekke sannhetsverdier og utføre instruksjoner basert på disse sannhetsverdiene. Sagt på en mindre akademisk måte: Vi skal gjøre noe dersom det vi tester på er sant:

```
if(test = true) {
    gjør alt inne i klammene;
}
else if(denne testen == true) {
    gjør dette i stedet for;
}
else {
    ikke noe av det andre slo til, gjør dette;
}
```

En if-test trenger ikke å følges av en else. Det vil si at dersom ikke if-testen slår til, skal vi ikke gjøre noe spesiellt i det hele tatt. Eksempel på en Robocode if-test:

```
void onScannedRobot(ScannedRobotEvent ev) {

    // Vi er for langt unna, flytt nærmere
    if(ev.getDistance() > 500) {
ahead(getDistance()/2);
fire(1);
    }
    // Vi er for nærme, dra lenger bort
    else if(ev.getDistance() < 10) {
back(50);
fire(3);
    }
    // Avstanden er fin, bare skyt.
    else {
fire(2);
    }

}
```

for-løkker

For-løkker er fine når man skal gjøre noe et bestemt antall ganger. En for-løkke kan se ut som dette:

```
for(int start = 0; start < maxverdi; start++) {

    Gjør alt her inne;

}
```

Det vil si, så lenge start er mindre en maxverdi, gjør alt inne i krøllparantesene, og øk start med en for hver gang.

while-løkker

While-løkker passer bra når man skal gjøre noe et uvisst antall ganger. Man har en eller annen testverdi, men denne er usikker på når slår til. En while ser typisk ut som:

```
while(så lenge dette er true) {

    gjør alt inne i krøllparantesene;

}
```

Vær forsiktig med while. Dersom man aldri får verdien false inne i while-testen, får man en evig løkke.

Appendix B

Karel J handouts

On the next pages follows the handouts given to the students in experiment 2. They are put in chronological order. That means that they appear as they were handed out to the students, with assignments and summaries.

Appendix C

Code listings from day 3

Below is the code from group 1 and group 2 from day three of Experiment 2.

First, group 1:

Listing C.1: Code from group 1, Assignment 5

```java
// Lager egne pakker for bedre oversikt
package kareltherobot;

/*
 * Her er innpakningen til programmet. Husk å bytte ut navnet
 * under fra "DinFil" til noe annet.
 */
class Karel52 implements Directions {
    /*
     * Under her starter programmet ditt. Når vi starter
     * programmet vil datamaskinen lete etter "main" og
     * starte alt derfra, så våre kommandoer må komme inni
     * der.
     */
    public static void main( String [] args ) {
        // Her kommer din kode:
        TrappeRobot Karel  = new TrappeRobot(8, 6, North, 0 , 0);
        TrappeRobot Jon = new TrappeRobot(1, 2, North, 0, 0);

        Jon.plukkTrinn();
        Jon.turnRight();
        Jon.finnPiper();
        Karel.move();
        Karel.finnPipe2();
        Karel.turnLeft();
        Karel.plukkTrinn2();
    }

    // Gjør kartet synlig, sett inn navn på kartet under
    static {
        World.readWorld( "eks6-1.kwld" );
        World.setVisible( true );
    }
}

class TrappeRobot extends Robot {
    // Holder rede på hvor mange trinn vi har gått
    int antTrinn;
    // Konstruktør
    public TrappeRobot(int gate, int aveny, Direction dir, int piper, int antallTrinn) {
        super(gate, aveny, dir, piper);
        antTrinn = antallTrinn;
    }

    // Gå ett trinn
    void ettTrinn() {
        move();
        turnLeft();
        move();
        pickBeeper();
        turnRight();
    }

    // Snu til høyre
    void turnRight() {
        turnLeft();
        turnLeft();
        turnLeft();
    }

    // Økker telleren for antall trinn med 1
    void leggTilTrinn() {
        antTrinn++;
    }

    // Skriver ut antall trinn
    void skrivUtTrinn() {
        System.out.println( "Vi har gått " + antTrinn + " trinn.");
    }

    //Legger ut piper så lenge det er i sekken.
    void leggPiper(){
        while (anyBeepersInBeeperBag()){
            putBeeper();
            move();
        }
    }

    void plukkTrinn(){
        while (frontIsClear()){
            move();
            turnRight();
            move();
            turnLeft();
            pickBeeper();
        }
    }

    void finnPiper(){
        while(!nextToABeeper()){
            if (frontIsClear()){
                move();
            }
            else {
                turnLeft();
            }
        }
        pickBeeper();
    }

    void finnPipe2(){
        while(!nextToABeeper()){
            if (frontIsClear()){
                move();
            }
            else {
                turnRight();
            }
        }
        pickBeeper();
    }

    void plukkTrinn2(){
        while (frontIsClear()){
            move();
            turnRight();
            pickBeeper();
            move();
            turnLeft();
        }
    }
}
```

And group 2:

Listing C.2: Code from group 2, Assignment 5

```
// Lager egne pakker for bedre oversikt
package kareltherobot;

/*
 * Her er innpakningen til programmet. Husk å bytte ut navnet
 * under fra "DinFil" til noe annet.
 */
class Karel8 implements Directions {
    /*
     * Under her starter programmet ditt. Når vi starter
     * programmet vil datamaskinen lete etter "main" og
```

```java
     * starte alt derfra, så våre kommandoer må komme inni
     * der.
     */
    public static void main( String [] args ) {
        // Her kommer din kode:
        ur_Robot2 karela = new ur_Robot2(8, 6, North , 0);
        ur_Robot2 karel = new ur_Robot2(1, 2, East, 0);
        while (!karel.frontIsClear()) {
            karel.trapp();
        }

        while (karel.frontIsClear()) {
            if (karel.frontIsClear()) {
                while (karel.frontIsClear()) {
                    karel.move();
                    if (karel.nextToABeeper()) {
                        karel.pickBeeper();
                    }
                }
            }
            karel.turnLeft();
        }
        karel.turnOff();

        while (karela.frontIsClear()) {
            if (karela.frontIsClear()) {
                while (karela.frontIsClear()) {
                    karela.move();
                    if (karela.nextToABeeper()) {
                        karela.pickBeeper();
                    }
                }
            }
            karela.turnRight();
        }
        karela.turnLeft();
        karela.turnLeft();
        karela.move();
        karela.pickBeeper();
        karela.turnRight();
        karela.move();
        while(karela.frontIsClear()) {
            karela.trapp2();
        }
    }

    // Gjør kartet synlig, sett inn navn på kartet under
    static {
        World.readWorld( "eks6-1.kwld" );
        World.setVisible( true );
    }
}

class ur_Robot2 extends Robot {
    public ur_Robot2(int gate, int aveny, Direction dir, int piper) {
        super(gate, aveny, dir, piper);
    }

    void turnRight() {
        turnLeft();
        turnLeft();
        turnLeft();
    }

    void trapp() {
        turnLeft();
        move();
        turnRight();
        move();
        pickBeeper();


    }

    void trapp2() {
        turnLeft();
        move();
        pickBeeper();
        turnRight();
        move();
    }

    void walkLong() {
        while(frontIsClear())
            trapp();
    }
}
```