Roboworld
Overcoming the Problem of Cognitive Load in Object-Oriented Programming by Microworlds

Mémoire en vue de l'obtention du DESS
en Sciences et Technologies de l 'Apprentissage et de la Formation

Hanspeter Heeb

(http://tecfa.unige.ch/~heeb)
(hanspeter.heeb@tgnet.ch)

Defended on October 31 2001

Jury

Dr. Daniel SCHNEIDER
Chargé de cours, TECFA, Université de Genève (thesis director)

Dr. Daniel PERAYA
Maître d'enseignement et de recherche TECFA, Université de Genève

Paraskevi SYNTETA
Assistante d'enseignement TECFA, Université de Genève

Acknowledgement

Table of Contents

Abstract

This study is centred on the design and development of a microworld learning environment for object-oriented programming and object-oriented design. Special care was taken to meet three goals: Firstly, to reduce cognitive load. Secondly, to maintain as much as possible similarity with the programming language Java. Thirdly, to facilitate the introduction of object-oriented design by means of the Unified Modelling Language (UML). The environment can be used for visualization (support teacher's presentation), for students training (solve slightly different programming problems based on previous teaching) and for the construction of new problems or solutions to problems by students. The impact of cognitive load theory on instructional design is a major topic of this study.

Keywords: Microworld, Cognitive Load Theory, Unified Modelling Language, Java, Object-Oriented Programming

Introduction

This study has three scientific goals. Firstly, the analysis of the paradigm of object-oriented programming and what this paradigm shift implies for teaching. Secondly, the analysis of the problem of cognitive load related to teaching object-oriented programming and of existing methods to overcome this problem. Thirdly, the design of a microworld that facilitates best the achievement of the knowledge needed for object-oriented programming most likely.

1   Practical Background

The programming language Java is widely used for introducing novice learners to programming. Also, it is used to teach object-oriented programming to expert procedural programmers.

There are several reasons for this. Kim N. King, probably one of the best teachers in this field, writes in his book «Java Programming, From The Beginning» (King, 2000, p. xvii):

> «When Java appeared in 1995, I immediately noticed its potential as a language for beginning programmers. Java satisfies today's need for early instruction in an object-oriented language, while avoiding the complexities of C++. At the same time, Java is similar enough to C++ to serve as a stepping-stone to that language. (At my institution, Java is the first language introduced, immediately followed by C++). With no pointers to cause problems, Java programs are immune to those frustrating crashes that are so common in C++. If a Java program does encounter an error at run time, the interpreter provides a stack trace that can often pinpoint the problem. And, last but not least, the software needed to write and execute Java programs can be downloaded from the Web at no charge.»

At the institution for adult education where I teach Java, we have very ambitious goals. People without any background in information technologies should be able to start a job as a junior software developer after 6 months of education and training. This involves the teaching of very complex concepts and a huge amount of procedural and factual knowledge. We tried several approaches to achieve this goal. The main problem was to reduce the amount of new information presented at the same time (that means the cognitive load). We found that there is—at least in German—no appropriate material available. Although there have been very interesting techniques used in education and training, the problem is that most of them are not well documented.

Our task—designing a short course in Java—is quite frequent. Therefore this thesis is of high practical value. It discusses the problems we encountered and analysis solutions and material to work around these problems. It contains material and techniques we used. It discusses our experiences

and may give some hints for teachers to improve the teaching of object-oriented programming.

I organized the material as follows:

Chapter 1 explains the problem of cognitive load and the reason why it is higher in a Java short course than in a course for Pascal or Basic.

Chapter 2 analyses the impact of the paradigm shift from procedural programming to object-oriented programming.

Chapter 3 discusses three problems in Java education that are very important in practice. The findings in this chapter were used for improvements in the microworld Roboworld.

Chapter 4 analyses different approaches to overcome the problem of cognitive load. It checks whether experience can be explained by cognitive load theory.

Chapter 5 analysis existing microworlds as a promising way to overcome the problem of cognitive load in an elegant way. A way also that allows teachers to introduce important topics, such as problem solving and software design, very early in a course.

Chapter 6 outlines and describes a new microworld—called Roboworld—especially designed to teach Java in a short course. This chapter is to prove that the cognitive load is heavily reduced, and that there is enough similarity that Roboworld can serve as a steppingstone to real Java programming, either to the programming of Applets or of Java applications.

Chapter 7 discusses the metaphoric power of Roboworld so it can be used to explain basic concepts about the development of software.

Chapter 8 explains the importance of sign systems such as the diagrams of the Unified Modelling Language (UML) for the development of software and how Roboworld supports these sign systems.

Chapter 1: Cognitive load in a Java short course

Major objectives in a Java short course are mostly higher cognitive activities. For such activities, the cognitive load theory might be valid. The level of interactivity of the elements is extremely high.

1 Major Objectives in a Java Short Course

For the half-year short course to educate junior software developers, the course designers stated the following key competencies that relate to object-oriented programming:

The first educational cycle (180 lessons) includes designing uncomplicated applications and implementing them; applying the standard procedures of implementation, testing and documentation.

The second educational cycle (160 lessons) includes designing applications with database connectivity and developing, testing and documenting them with Java according to the standards of object-oriented programming.

2 Cognitive Load Theory

What cognitive load theory says and what it implies for instructional material and design. For a detailed 25 pages overview over it see Cooper (1998) and there, the sections «Suggested Readings» and «References».

2.1 What It Says

According to cognitive load theory, the key learning activities are schema acquisition and automation of their usage. After enough training, acquired schemata are stored in long-term memory. They allow high cognitive performance with a very limited working memory. But under conditions where multiple elements of information are interacting, we have to comprehend and learn them at the same time. Such material produces a high cognitive load by itself (intrinsic cognitive load). The instructional design may add to the cognitive load (extraneous cognitive load). As a rule of thumb, the working memory is limited to between 5 and 9 elements. If the material itself is not highly interrelated, it does not produce high cognitive load itself. In this case the cognitive load added by the instructional design is not critical.

To conclude, Learning difficulty correlates with the number of elements that must be learned simultaneously. Therefore, we have to study firstly how interrelated elements in the curriculum of object-oriented programming with Java are.

More information can be handled, when the learner can group information into larger units (schemata). That is, when he knows most of the content and the structure of the information.

For instance the text

```
class MyClass extends JFrame{
    public JLabel label1 = new Label("Name: ");
    MyClass () {
```

means an accumulation of puzzles to the beginner. The expert or even intermediate programmer will easily recognise the beginning of the definition of a stand-alone window for a graphical user interface (GUI).

2.2 What It Implies

For material that has a high level of cognitive load by itself, the cognitive load added by instructional material and course design should be as low as possible. Things that add cognitive load should be avoided.

Tested effects of the cognitive load theory are, according to Cooper (1998):

The goal free effect, the worked example effect, the split attention effect, the redundancy effect, the modality effect.

The goal free effect is studied for mathematics. In classical training examples for math, you have the goal and the given. There is a multiple step procedure from the given to the goal. The goal, the given and the possibilities in between overflow the learners working memory. If the learner has only the given and the task to find out what ever he can he performs and learns much better.

The worked example effect implies to give students worked examples to study. This way, they can build schemata. The teacher uses problem solving only to test if training has been effective.

The split attention effect suggest to avoid that the student has to look up information at different places. Examples for splitting attention are for instance: Working with a textbook and at the computer screen at the same time, or explaining the semantic elements of a diagram in the text outside the diagram instead of explaining these elements directly with notes on the diagram. Footnotes and endnotes.

The redundancy effect suggest to avoid to show similar content twice. Examples of redundancy are for instance: Adding pictures that do not explain the material that should be learned to a text book (Sauders & Solman, 1984). Upgrading a text with vivid, but lengthy examples. See Langer, e. a. (1981, p. 139) who did extensive research on readability in German.

The modality effect says that some portions of working memory are sensory mode specific. That means, you may expand working memory when using different modes of presenting information. The combination of spoken text and graphics would be such a presentation.

Effects under investigation are (according to Cooper, 1998): « (1) the procedural learning effect, (2) the imagination effect, (3) the colour coding effect and (4) the interaction effect.»

2.3 What It Is Not

The cognitive load theory is not a global learning theory. It is not meant to explain theoretically all learning experience. It is meant as a tool to improve teaching and the design of educational material. Compared to the three classical learning theories, there is, for me, a big difference. The classical theories, behaviourism, cognitivism, constructivism, all try to explain what enables learning. They explain what to do. Cognitive load theory explains why learning is hindered. It explains what to avoid. It does not explain how a schema is learned. It explains only why learning does not take place. Therefore, it seems to me complementary to existing learning theories.

To visualise that fact with an analogy: you can improve the overall performance of a racing car by tuning the motor, that is what classical learning theories do. Or, you can improve it by reducing weight and improving aerodynamics, this is what cognitive load theory does. Usually we think of the first and forget the later and as with racing cars we would gain more with the latter.

3 Cognitive Load Theory and Instructional Design

Wilson (1995, chapter «Cognitive Load Theory, A Crossover Theory of Learning and Instruction») writes referring to cognitive load theory:

> «These instructional prescriptions are consistent in spirit with traditional instructional theories. Because they come from a cognitive psychologist, however, they are tightly coupled with a specific learning theory. Conceivably, they could conceivably be taught to designers as rote rules to apply ....»

3.1 Cognitive Load Theory as Rote Rules to Apply

The question is whether cognitive load theory is a powerful tool to analyse existing problems in a field like object-oriented programming and helps to locate and improve the best means to overcome these problems. There is some existing evidence that a theory like cognitive load theory should be powerful. Cognitive load theory has a view, understandable rules. The practitioner should be able to estimate the number of schemata in some part of his course or in his instructional material. So he can locate problems caused by cognitive load. He can think whether he may build up higher level schema as prerequisites of this unit or to split the unit. He can check for additional cognitive load caused by the instructional material.

3.2 Simplicity of the Theory Is its Strength

Studies done in Germany in the early 70ties prove how important the simplicity of a method is if the goal is improvement in the practical field.

Langer e. a. (1981) did comprehensive research about the readability of German text. They found four criteria (factors) for readability and constructed a short course to learn to check for this four factors and improve the readability. Schulz von Thun, e. a. (1974) then checked whether this training was efficient. Well, of course it was. The most remarkable to me

of this check up are two things. Firstly, a short training that only helped trainees to check for the four factors already caused a significant improvement (readability raised about 50%). Secondly, in the control group that studied a style guide readability even dropped insignificantly. This did not happen, because the style guide suggested tips that where not congruent with the four factors. It happened, because the trainees had to consider to many rules at the same time.

By the way: the four factors coincide with cognitive load theory. The first factor is simplicity (length of sentences, known and figurative words). That means each sentence causes no extra cognitive load by unknown words, or to many words in a sentence. The second is text organization (logical flow, introduction to paragraph, mark ups of important words, titles). That means no extra cognitive load needed to bring the ideas expressed in the text into a logical order. The third was length of the text. The text should be as concise as possible. But shortage should not get in the way of simplicity. This means, no unnecessary redundancy. The forth factor was redundancy added to motivate the reader. For instance vivid examples, questions, dialog. This redundancy showed a negative impact, as soon as the organization of the text was not perfect.

3.3 Overcoming the Old Fashioned Way

In practice, some ideas persist a long time, even in case of severe frustration. The findings of the cognitive load theory often prove that these standard practices are inefficient. And, what is important for teachers, changes do not need enormous efforts.

3.4 Conclusion

The cognitive load theory seems to be an interesting, simple and efficient tool to analyse weaknesses and design improvements in a field where you encounter high levels of students frustration.

4 Analysis of the Cognitive Load in a Java Short Course

4.1 Outline of the Curriculum

The objectives of the above short course were:

For the first educational cycle (180 lessons): Knowing the structure of a Java application. Applying the basic Java commands to write programs. Using classes (of the API) and objects correctly. And, lastly, understanding the procedure of refactoring object-oriented software.

For a second educational cycle (160 lessons): Implementing database connectivity with Java. Writing simple applications that provide net services. Knowing sources of current information about Java. Defining software specifications with Unified Modelling Language (UML) and implementing software that complies with them.

6

Most interesting in the view of cognitive load is the first cycle of our Java short course, because it is the most demanding. A lay out of the knowledge needed is shown in fig. 1 as use case diagram. To be able to write an uncomplicated Java application, the trainee needs all knowledge related to the three icons that show person-figures labelled with «basic object-oriented», «intermediate object-oriented» and «Java programmer». A lot of interrelated information is needed, even to write a simple Java application without graphical user interface. There are seven topics interrelated with this first milestone. Usually this topics are teached as follows:

Instruction how to set up the Java Developer Kit, how to enter the text of the application "Hello-World", how to save, compile and run this simple application. This is the prerequisite to start learning. Therefore, this preparations are not in the diagram.

Secondly, the student learns how to store values in variables of primitive type and how to perform basic operations. Sometimes, this is done simultaneously.

Thirdly, the student learns the use of control structures. Usually, This is not possible without variables and Boolean operations. Hence, to study control structures includes variables and basic operations.

Then, comes the hard core of object-oriented programming where it is difficult to decide in which order defining a class, method call and return value and last but not least instantiation should be taught.

Deviation: Short Explanation of Use Case Diagrams

Use case diagrams are part of the Unified Modelling Language (UML).

A use case diagram, as fig. 1 on the next page, usually shows all possibilities how to use an application. For me, it is also a way to outline a curriculum. Every use case describes an activity. Arrows show the interdependencies of these activities. Base activities should be learned firstly, because other activities extend the material of a first activity. Important, the arrow points to what needs to be learned firstly, the basic activity. Outlining a use case diagram, one may split up larger activities into smaller activities, as long as there is still an activity that the trainee can carry out.

By the way: the presentation of the material as use cases diagrams may facilitate the learning of use case diagrams. It may also help to give an overall view of the course and feedback on the ground the learners have covered, while proceeding in the course.
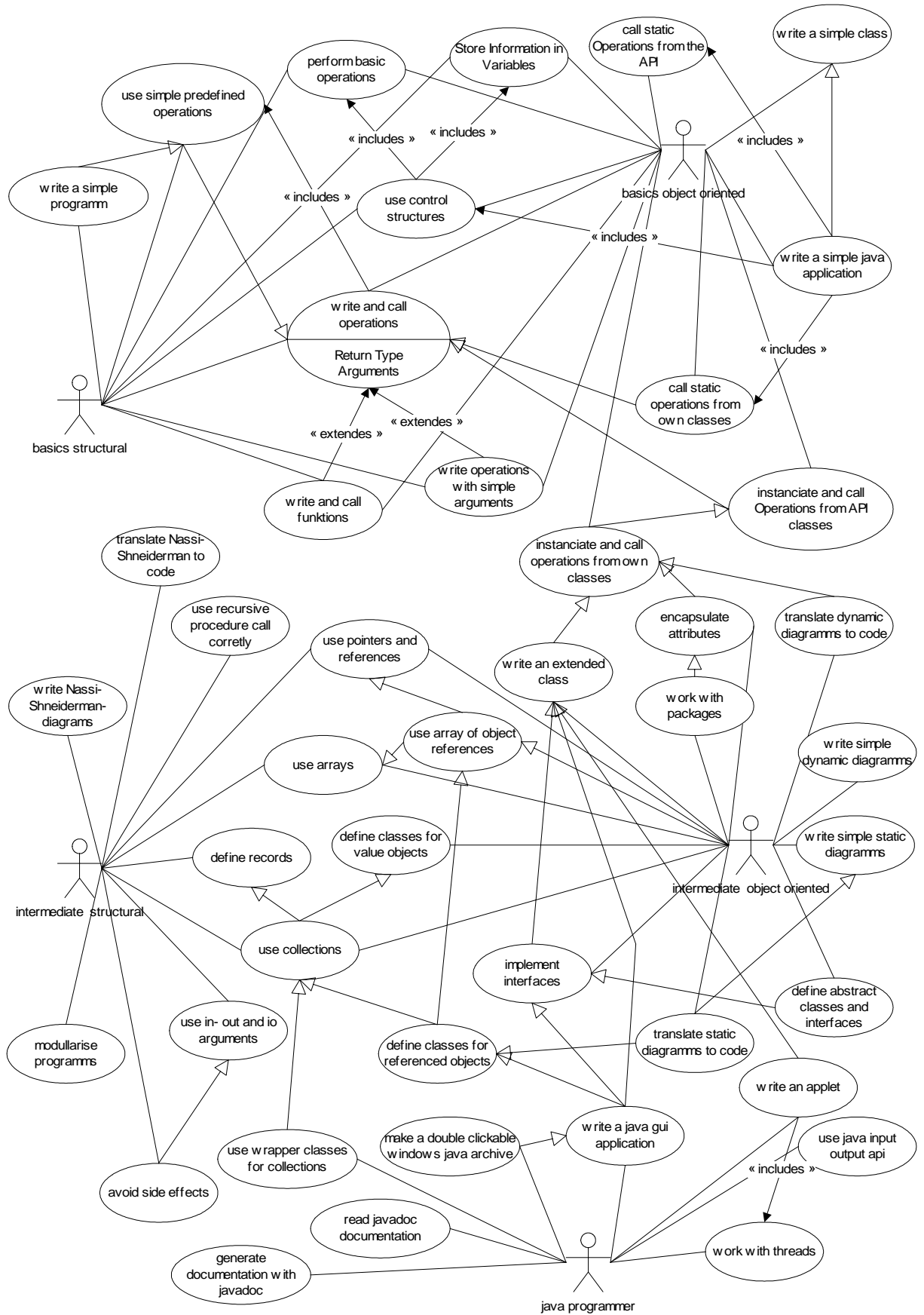
Fig. 1, curriculum and its dependencies as use case diagram

4.2 Reason for the Complexity of Object-Oriented Programming

One main problem for teaching is the complexity of object-oriented programming. You do not understand the code of the most simple Java application or applet (»Hello world!«) as long as you do not know very complex concepts of object-oriented programming. Java is an ingenious language, but you need a lot of Java specific knowledge even for the most basic tasks.

For instance, the «Hello world!»-Application that a teacher in general must use needs:

In a procedural programming language, such as Pascal	In an object-oriented programming language, such as Java
• A simple program structure • The predefined procedure «writeln()» • The string constant "Hello world!"	• A simple class structure • The definition of a static method «main()» • The string object that contains "Hello world!" • An array of objects of class String, even if this argument array is not used • The class System, because this class contains a reference to the standard output in the static attribute out. • System.out refers to an object of class PrintStream. • This object's operation «print()» is called with the argument String, i. e. «System.out.println(String)» • The argument String is upcasted automatically to argument Object. • The operation «println(Object)» of PrintStream calls the operation «toString()» from class String to write the object to the standard output.

Table 1

4.3 Programming Environment

Also, the programming environment adds to the cognitive load. There is no programming without a programming environment. The set-up of the programming environment has become easier with Java 1.2. Still, a lot of knowledge is needed. For instance to use predefined classes. Such classes

are provided as part of some courses, or are crucial for some exercises. Complexity bursts when packages are used or introduced.

The trainer has to decide whether to use a software developer tool like «JBuilder™» that helps in generating and managing projects, or to use a simple editor like «EditPlus». Choosing the first option means teaching the overhead of a sophisticated software. Choosing the second option requires a sound understanding of the file system, text editing and the usage of a command line operating system like DOS or Unix.

4.4 Technical Terms

The amount of technical terms, especially the terms in the feedback provided by the compiler, is high. For trainees who are unfamiliar with English the English language adds to the cognitive load as well.

4.5 Graphical User Interface

Programming graphical user interfaces (GUI) is rather complicated. Setting up a user interface with swing is very well documented in the Java-tutorial. Nevertheless, because of its richness and complexity, it is impossible to understand for the novice. But the novice would like to do real things. This means, he or she would like to write small applications that have a graphical user interface and code that he or she understands completely.

4.6 Command Line Interface

Even the set-up of a command line interface with Java is very demanding. Comparable to the graphical user interface, the command line interface uses advanced features of Java.

5 Conclusion

In a typical Java short course, there is a high level of intrinsic cognitive load. Special care has to be taken to reduce cognitive load added by course design and course material. Therefore, we should take advantage of any means of reducing cognitive load.

Procedural programming and object-oriented programming are based on different problem solving strategies and have different key concepts. This has a heavy impact on education.

1 Motivation for Object-Oriented Programming

Object-oriented programming was created because of shortcomings of procedural programming. Procedural programming processes data input and produces data output. Whenever the structure of the processed data changes, large parts of the application need revision. This happens often, for instance if the user likes to have a field for an e-mail-address in his address book. In such cases procedural programs are difficult to maintain.

In object-oriented programming however, the address book and all operations on the address book are built into a small module, a class. Adding a field means only modifying the methods in this class. Of course, the view and control of the class needs also modifications. But, all other parts of an application that do any transaction with the data defined in this class, will never be affected.

Also, as the class defines data and methods to modify this data in one file, this class might be used for other applications as well. Transferring parts of data structures and related procedures out of a procedural programmed application is much more difficult.

2 Problem Solving Strategies Compared

Procedural programming and object-oriented programming uses two completely different problem-solving strategies. This implies other key concepts. Focusing on the concepts of object-oriented programming may reduce cognitive load.

2.1 Procedural Programming

Starting point of procedural programming is a main task of data procession that should be performed. For instance a banking transaction like buying shares. This main task is divided into subtasks. For instance, checking clients credit limit, checking price limits, getting actual stock market price, accept offer or send an offer to stock market, put shares into clients deposit, withdraw money from clients account. Again, these subtasks are divided into smaller tasks until the tasks meet two criteria: Firstly, the programmer does not have to rewrite the same code twice. Secondly, the task is small enough to be written to code with no further analysis.

This means, the typical problem solving strategy of procedural programming is a recursive task. A procedure WRITE_PROGRAM looks like this:

IF Task is small enough and not redundant THEN write the Procedure for it ELSE do WRITE_PROGRAM.

Besides this top-down strategy there is also a bottom up strategy. It says, implement elements that are relevant for the main task. Chain together the elements to build the solution. In practice, a mixed strategy is used.

2.2 Object-Oriented Programming

In object-oriented programming we build a model of the reality, the problem domain of our application. If we do it with class diagrams, this model looks much the same as an entity-relationship model used for database design. This means, the starting point of the developer is data, not procedure.

The developer treats every element of this data structure as one small application. This small application may act as a server-application and as a client-application at the same time. That means, the application can be called to provide a service (for instance provide the credit limit of a customer) and calls other servers to provide services to them (for instance asks for the customers monthly salary to set the credit limit).
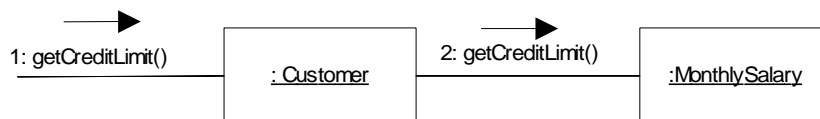
Fig. 2, objects as server and client

The information exchanged between server and client is not directly related to the internal data in the server. For instance, a server, asked to provide the monthly salary could return the average income of the last three years in case of a salesman with a volatile income, or return a value set by the customers contact person in case of an important customer in private banking. There is no need to enter wrong numbers into fields just to work around inflexible data structures. The data structure must be flexible, whereas the messages sent to the server do not change.

Therefore, the strategy is to define the services provided by any element of the application (the responsibility of the classes) and which classes should interact with each other (the collaboration).

For the services that the class should provide, methods are written. The methods either check the information inside the class and give an information (query methods) or that change the information inside the class (update-methods). Usually, when the most basic methods are written firstly, there shouldn't be any complicated methods. That means, after having defined classes, responsibilities and collaborations, writing methods is basically a bottom-up strategy.

3   Key Concepts Compared

Procedural programming and object-oriented programming have different key concepts. To reduce redundancy—that means unnecessary cognitive load—we have to check whether this key concepts of procedural programming are needed for object-oriented programming.

Key concepts of procedural programming are control structures like iteration and conditional branching, recursion, modularisation and side effects.

3.1   Conditional Branching

Conditional branching is a main topic in procedural programming. Writing Nassi-Shneiderman-diagrams should help to avoid spaghetti-code for complex tasks. A lot of effort in design and testing is made to get the branching right.
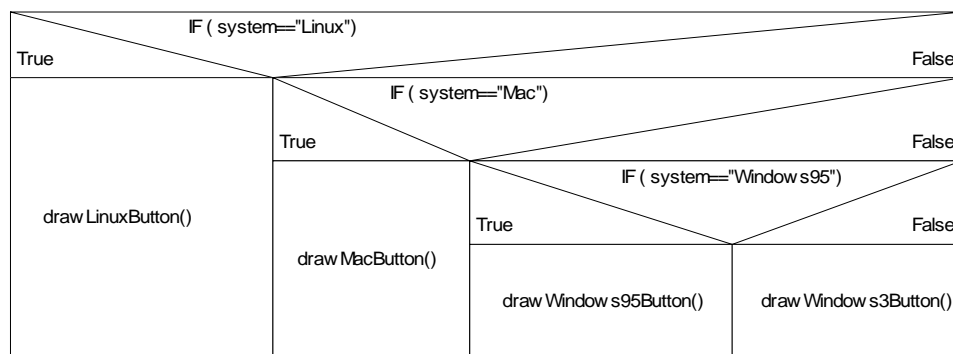
Fig. 3, Nassi-Shneiderman diagram for nested if statements

Object-oriented programming does not make much use of conditional branching, because it uses a much more powerful tool to create diverse behaviour: polymorphism.

Let us study for example the user interface. Our application may run on Linux, Mac, Windows 3.x or Windows 95. In a procedural program a global variable would be set in the beginning due to the system the program is running. For instance:

```
system = "Windows95";
```
Later we would check for this variable like:
```
if (system == "Linux") drawLinuxButton();
if (system == "Mac") drawMacButton();
if (system == "Windows3") drawWindows3Button();
if (system == "Windows95") drawWindows95Button();
```
In an object-oriented program we would have a module (a object as server) that provides us the objects of the user interface. In the beginning we would activate this server:
```
userInterface = new Windows95UserInterface();
```
Interface is now the reference to these objects that serve as elements of the user interface. Now all we have to do is ask this server to provide us with elements of the user interface:

13

userInterface.drawButton();

The server, userInterface, draws for us the button. On Windows 95 a windows 95 button. The code inside of the server does only treat Windows 95 buttons. There is no need for conditional branching.

3.2   Recursion

There is no need to discuss recursion in object-oriented programming. If the instances of a class do their job correctly, they will do it also when called recursively. For instance: In the composite-component design pattern any container calls all its components. As every container is also a component, this is a recursive call. Studying this design patterns that has a recursive structure (association) replaces the analysis of recursive procedure calls. The composite-component design pattern is discussed in Gamma, e. a. (1994).

The expression «recursive call» got another meaning in object-oriented programming. See Object Modeling Group, 2000, 3.60.2. It means that the object calls itself and not that a method calls itself. The later is only a very rare and unimportant sub case of the first. Also when taking a closer look at other books discussions on recursion are rare: Eckel's object-oriented programming bible «Thinking in Java», Eckel, 1998, mentions recursive method calls only twice: on page 342 as something causing a never ending loop and on page 926 he uses it and explains it by the way. Sun's Java Tutorial does not mention recursion; other authors just mention it shortly giving one example, for instance Schildt, 2001, p. 246.

3.3   Modularisation

Obviously this cannot be any topic in object-oriented programming, because classes are already small modules. Instead of modularisation, object-oriented programming uses packaging to produce units that could be regarded as parts of a larger application.

3.4   Side Effects

Avoiding side effects and handling the procedure's parameters with care is a major topic in a course for procedural programming. Therefore, how to avoid side effects is a major topic in a good book or course about procedural programming.

Side effects in object-oriented programming are very rare and can easily be avoided. The increment operator (+ +) has a side effect when put in front of the variable. Then the variable returns the incremented number.

Example:

```
i = 0; while (++i < 10) { .... ; .... ; }
```
where a side effect occurs can be translated to
```
i = 1; while (i < 10) { ... ; ... ; i++; }
```
where there is no side effect. The student sees that the loop starts with 1 and does not need to figure out whether it starts with 0 or with 1. The easiest way to reduce cognitive load here, is just not to mention the first possibility for an increment.

Eckel (1998, chapter A: Passing & Returning Objects) writes about real side effects:

> «In general, you call a method in order to produce a return value and/or a change of state in the object *that the method is called for*. (A method is how you "send a message" to that object.) It's much less common to call a method in order to manipulate its arguments; this is referred to as "calling a method for its *side effects*." Thus, when you create a method that modifies its arguments the user must be clearly instructed and warned about the use of that method and its potential surprises. Because of the confusion and pitfalls, it's much better to avoid changing the argument.»

Most Authors do not even mention real side effects in object-oriented programming.

4 Novices' Problems in Programming

4.1 Novices' problems with procedural programming

Lewis (1980), as reported by Rajan (1990), found for a procedural programming language the following problems of novice programmers in descending order: (1) flow of control, (2) side effect, (3) recursion, (4) procedure call (to many or not enough arguments), (5) variables (non unique names for global variables), (6) modes (of the programming environment).

4.2 Validity of these Problems for Object-Oriented Programming

According to the different paradigm, most of these mistakes can not occur in object-oriented programming. These are side effects, recursion and global variables with non unique names. Flow of control and procedure call are very different in object-oriented programming and cause therefore different problems. As a result, we can not, based on this study, figure out what the main problems of object-oriented programming are.

5 Conclusion

Teachers can avoid some advanced topics like side effects, extensive conditional branching, recursion and modularisation in a Java course. Nevertheless, this fact does not reduce the heavy cognitive load in the beginning of a Java course as discussed in the previous chapter. Novices' problem studied for procedural programming give us no exact guideline for problems we will encounter in object-oriented programming, because the main problems of procedural programming can not occur in object-oriented programming. These problems are, therefore, studied more deeply in the next chapter.

Chapter 3: Key problems of teaching object-oriented programming

There are some problems that make object-oriented programming harder to teach than procedural programming. In this chapter, we take a closer look at these key problems. They are: variables, pointers (object references in Java), static attributes and operations, and last but not least the collaboration of objects.

1 Understanding Variables in Java

1.1 Experienced Problem

As most people nowadays work with spreadsheet application like «1-2-3» or «Excel» you would not expect that people have difficulties with the concept of variables. Therefore I underestimated the difficulty to make the transfer from a spreadsheet cell to the concept of a variable in Java.

1.2 Analysis of the Problem

In a spreadsheet application a cell address like B4 is rather an instruction to look up information in another cell but a variable. This is evident with relative addresses where you look up the value in a given distance. As this addressing is most common, users might have a notion of cells as a look up instruction rather than a variable. Absolute addressing is only some special case of a relative look up.

Most learners—as long as they are not mathematicians—are unfamiliar with variables. Variables are therefore something unfamiliar.

1.3 More Cognitive Load Related to Variables in Object-Oriented Programming than in Procedural Programming

But, why do variables cause almost no problems in procedural programming but can become a burden in object-oriented programming?

In procedural programming there are two kinds of variables: global variables and local variables. They share the same name space. The end of block limits the scope of local variables and they may hide global variables or local variables of an outer block.

In object-oriented programming there are some other components of complexity added:

Firstly, there is the name space of the variable. Every object has its own name space. Therefore, different variables with the same name are relevant in many situations. Trainees have to learn that assigning a value to a variable does not mean that every variable with that name holds now that value. Plus, he can use the same name in different classes for different

values and even different types. When you work with variables in object-oriented programming it is very confusing for learners that the same name of a variable holds different values (information) and that the same data (information) is stocked in variables with different names, when being sent from object to object.

Secondly, objects and classes share name spaces. Inside a method we can access the variables of the object, called the object's fields without referencing the object. We can also refer to the object with the key word «this». Sometimes we have to refer to it, because a local variable has the same name as the object's field. Resulting in code like

```
this.balance = balance;
```

where the value of balance (a local value) is assigned to the field called balance. If there is no field balance there could even be a class variable balance. These fields and class variables might be defined in any basic class of the current class.

Even if you have not understood this explanation because you are unfamiliar with object-oriented programming, you have seen that the cognitive load is heavily increased compared to procedural programming.

Sometimes learners are so confused that they even doubt whether value passing goes from left to right or from right to left.

1.4 Usual Attempt to Solve this Problem

Make learners to send the content of variables to the standard output «System.out.println()» throughout the program flow.

Instruct them to use different names at different places for variables that handle the same information (data) in different classes and their operations. This way they learn at least that value passing is independent from variable names.

1.5 Conclusion

It is preferable that the schema of objects and classes and the collaboration of objects are generated in trainees mind, before variables as a major topic are introduced. Otherwise, the cognitive load is overwhelming. A microworld should support the acquisition of the most important type of variables, objects variables.

2 Understanding Pointers or Object References

The concept of object references (pointers) is known as complex. In object-oriented programming it is needed pretty early.

2.1 Problem Description

If you do not enter technical aspects only interesting for expert programmers, pointer and object references are the same. In fact, a pointer in the procedural programming language Pascal has more similarity with a Java object reference than with a pointer in the procedural language C. From the conceptual point of view, pointer and object references use the same schema; they apply to the same problems. Pointers are difficult, because

naming a place where information is stored (a variable) is more natural than naming the reference to that name (a pointer). Learners may even be familiar with the concept of variables from mathematics. But there is nothing like pointers in the real world.

2.2 Metaphors for Pointers

To explain pointers, teachers created metaphoric explanations. The metaphor that a pointer is like the address of a house, where different people may live at different times is widely used. At address nr 14, there might live family Miller. They move out and Brown moves in. So nr 14 may point to Miller at one time, Brown at another time. This metaphor works fine in procedural programming.

I am not very enthusiastic about such metaphors. They build extra item to be remembered. As long as metaphors are still needed to scaffold the concept, they use up working memory.

2.3 More Technical Explanation

Therefore, I explain my students that objects have sequential numbers. Usually these numbers are hidden. The variable that holds a reference holds the object's number. The only way to call up an object's attribute or operation is by this number.

When explaining objects, I often use these pseudo serial numbers instead of objects name. For instance in a object diagram instead of showing an object like:

| myObect : ClassName | or | : ClassName |

I show it like:

| #00462 : ClassName |

If learners understand the concept of assignment, they do understand the concept of assigning a serial number to a variable. The only additional information to learn is the fact that serial numbers cannot be used directly in program code like:

```
#00462.toString();
```

But you can use it directly for further programming without assigning the serial number to a variable. You take the serial number returned from an operation and use it as a reference to the object.

```
(new Integer(205)).toString();
```

This is understandable as soon as you imagine that (new Integer(205)) returns a serial number of the new object.

2.4 Conclusion

Objects references are very important. Visualising the reference number of objects can support this acquisition.

3 Understanding Argument Passing with Messages

3.1 Description of the Problem

The exchange of information between the object, acting as client and the object, acting as server, is one of the main concepts in object-oriented programming. The client sends a request to the server by calling one of the servers operations. It may send values or object references as arguments when calling an operation. The server object uses the arguments within the called operation. The called operation may return a value or object reference.

Also, as this seams to be the same as calling a procedure in procedural programming, there are elements that add complexity. The called object (the server) may be the object itself, but it may also be another object. This other object is defined in a different class. Therefore there are more elements to be remembered at the same time than in procedural programming. The split of attention towards two files (two classes) is inherent to the call of methods.

There are some existing solutions that address this problem.

3.2 Solution: Procedural Programming at First

Procedural programming is basically equivalent to object-oriented programming concerning procedure or function call and return value. This justifies teaching this part of procedural programming first. This often means to teach method calls within the «main()»-method of a class. Because, using the «main()»-method in teaching is the most common technique of teaching procedural programming first in Java. After the schemata of methods are acquired with one class (no splitted attention), courses usually start with existing classes of the Application Programming Environment (API). This can be done also without attention split, when the used existing classes of the API and their usage are explained first. After that one starts with two classes and their objects exchanging messages.

3.3 Solution: Step-by-Step or Spiral Approach

Training that goes smoothly from the call of simple methods to more sophisticated method invocation do lower the cognitive load. You may start with simple methods firstly and post-pone sophisticated method invocation to later in the course (Spiral Approach).

3.4 References (or Pointers) as Arguments and Return Type

References as arguments and return type do not cause problems only in object-oriented programming but also in procedural programming.

Firstly, references are like pointers to records. We discussed this topic already earlier. Secondly, references in Java are the only way to have output arguments and input-output-arguments. That means arguments do not only provide additional information to be processed within a method but can also receive information during the execution of the method. In proce-

dural programming, developers take care not to alter other variables than the out- and in out arguments to avoid side effects.

The structure of object-oriented programs is different. We will discuss this in the next chapter.

In Java the argument passing by value is a very important topic. It is well worth to invest several lessons to provide trainees with enough experience. In several books String and StringBuffers are used to explain value passing in Java. As Strings are immutable they cannot be used for output or input-output arguments. StringBuffers are mutable and can be used for it.

3.5 Conclusion

Messages and argument passing are a very demanding task with a heavy cognitive load. It is interrelated to other topics with high cognitive load like variables and object references. Approaches that do not overload the learners working memory are crucial.

4 Grasping the Idea of Object-Oriented Programming

The most challenging part in teaching object-oriented programming is getting the students to understand the idea or paradigm of object-oriented programming.

4.1 Main Problems in Teaching the Paradigm of Object-Oriented Programming

Let us assume the trainee knows procedural programming or we taught the parts of object-oriented programming that are congruent to procedural programming, first. Now, what impedes trainees to work according to the paradigm of object-oriented programming? —I experienced that three problems are most cumbersome in practise:

1. Falling back into the paradigm of procedural programming's overall control and therefore abusing static attributes and static methods.
2. Collaboration and encapsulation: Working with object references or pointers.
3. Inheritance and Polymorphism

I will analyse these problems in the following three sections.

5 (Miss-)Understanding Static Attributes and Operations

This is a small but nasty problem. Because, trainees mix up the concepts of classes and objects. Plus, ideas of overall control dominate their thinking. Concentrating on objects can help. I introduce the notion of static object to clearly separate the concept of class and object.

5.1 Description of the Problem

In object-oriented programming we set up classes. These classes define objects. These objects shall do the job. Therefore, there is not much need for static operations and static attributes. But, in the beginning, students need to store information sometimes and they want to retrieve this infor-

mation without worrying too much about the design of objects and their classes.

So this is what you will see: Students use static attributes where you would expect object's attributes. This occurs if there is only one object of a class in a specific project. Students will defend themselves: «Well, it works. Firstly, I tried an objects attribute, but then there appeared any compiler error. I tried the static attribute and it worked.»

This means: Students have not understood the process of instantiating an object. They try to work with an object before they constructed an instance of it. As the compiler's error message says something about static, they try using static attributes. Also as you worked before in the static «main()»-method you might have introduced attributes there. Therefore, if you introduce attributes before you introduce instantiation, you will run into big problems. Your students will remember static attributes as something you use regularly. They do not see it as something very special, you use only for specific tasks.

5.2 A Good Remedy: Treating Static Elements as a Special Object

Besides teaching the process of instantiating well, so students don't run into compiler errors, or, teaching them how to fix this problem properly, I found the following explanation of static attributes and operations accurate.

I visualise static attributes in an object diagram. For instance in the account example, I show classes and the content of their attributes as objects.
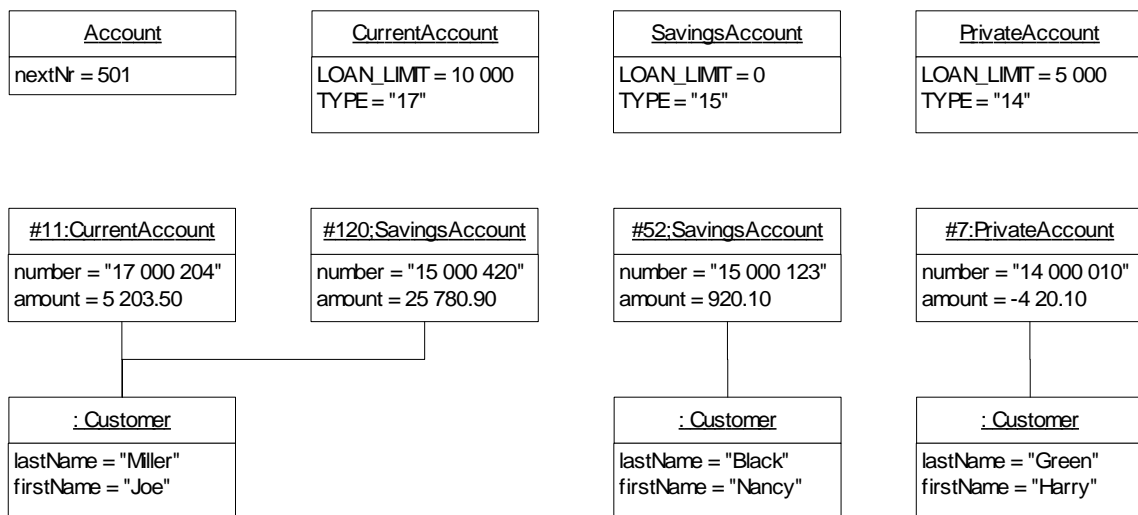

Fig. 4, classes displayed as static objects

I explain them that any class holds definition for two kinds of objects:

5.3 The Static Object of the Class

From one kind, there exists only one object. This object is automatically constructed, when the class or one of its objects is used for the first time. You refer to the attributes and the methods of this object by the name of the class. This object is defined by the static attributes and methods in the class definition. Let us call it the static object. As the static object needs no instantiation, it is not correct to name it an instance of its class. I acknowledge that the notion of a static object proposed here is new. But it is powerful and it might overcome unnecessary argumentations that static methods and attributes are not really object-oriented. Eckel (1998, p. 162, chapter 4) writes about this issue: «Some people argue that static methods are not object-oriented since they do have the semantics of a global function; with a static method you don't send a message to an object, since there's no this. This is probably a fair argument, and if you find yourself using a lot of static methods you should probably rethink your strategy. However, statics are pragmatic and there are times when you genuinely need them, so whether or not they are "proper OOP" should be left to the theoreticians. Indeed, even Smalltalk has the equivalent in its "class methods."»

To sum it up, static objects provide a service like any other object. They are available all the time throughout execution of the program.

5.4 Instances of the Class

The second kind is the regular objects, the instances. From this kind of object, there can be no instance then it is a utility class like class Math, there can be one and only one instance, then it is a singleton, or, there could by many instances. The latter is the usual case.

To use the concepts of object-oriented programming correctly we can use object, as super category of instance and static object. Therefore it might help learners when we draw object diagrams with the static object and the instances of the class. This is not foreseen in the Object Management Group's (2000) Unified Modeling Language, Specification, version 1.3. But it might help to avoid a lot of confusion between classes and objects. The Object Management Group suggests mixing classes and objects in static diagram. This might be convenient to experts, it is confusing to novices.

5.5 Shared Name Space of Instances and the Static Objects of its Type

In the code inside a class, you can refer to the attributes and operations of all the static objects of the basic classis without referring to the class. In this case, the program looks for an attribute or operation first in the static object of the current type of the variable that refers to the object, then in the static object of the parent class of that type, then in the static object of the parent's parent class and so on. In the example above, inside of object #11, a CurrentAccount, TYPE refers to TYPE in the static object CurrentAccount that is 17 and nextNr to the static attribute nextNr in the static object Account.

5.6   Conclusion

Static elements augment complexity. They are needed in a normal learning environment, but are unknown in microworlds. Using microworlds decreases the cognitive load therefore.

6   Collaboration and Encapsulation: Working with Object References (or Pointers) in Messages (Operations)

6.1   Description of the Problem

What is the kernel of object-oriented programming? —In object-oriented programming you think of small units (objects) that do a small part of the job. Every single operation is trivial. The complexity lies in the collaboration of all the objects (instances) of these classes. You may discuss this problem from the point of view of program design, then you see it as a responsibility-collaboration problem; you may define it as the technical goal of encapsulation to get small and stable interfaces. Schildt (2001, p. 11) identifies Encapsulation as the first principle of object-oriented programming. I see it more pragmatically for beginners: The objective is to be able to set up collaborating objects and this means to understand how object know about each other and how they refer to each other. It means to know how such references are set up in a program. Because, collaboration is only possible if several objects know each other.

In the view of object-oriented design, collaboration is often seen as aggregation. Instead of defining one object that does the entire job, you design several objects who work closely together. There may be one object that acts as a contact to the outside of the group and receives all messages first. But then, this objects sends messages to other objects that are closely related to it.

For the beginner, aggregation or collaboration, is difficult to understand and even more difficult to set-up and implement. There are several ideas to help beginners with setting up aggregations of collaborating objects.

6.2   Visualization of Collaboration in Class Diagrams

The association in class diagrams are a good visualisation of the relationship between objects. Students understand usually associations in class diagrams. But, the transfer from drawn association in a static diagram to code is not evident. You have to teach how to transform associations to code. For instance, to teach collections like Vector does not help. There is a relationship between the kind and number of objects at each end of an association and the type of collection best suited to maintain this relationship. The trainee must learn this relationship between association drawn in a chart and the collection used to maintain references.

6.3   CRC-Cards and Role-Play

Working with CRC-Cards to test program flow is an excellent way to experience the way collaboration in object-oriented programming works. Joseph Bergin calls this «Design by walking around». I would call it an objec-

todrama, because I do not only use CRC-Cards for designing new interaction, but also to replay existing interactions. For instance I replay with my students the event processing of the graphical user interface.

6.4   How to Use CRC-Cards and Role-Play

Students tend to use interaction sequences were one object takes control and the other objects stay more or less passive. That means they do more or less procedural programming. One object takes the part of the structured program the other play the role of records that store information. You end up with programs where one class is extremely long all other classes are short and have only simple update and query methods. Even trainees that never learned procedural programming show this tendency. To prevent this structured thinking, you may give your students different restrictions for the way they should construct the interaction. Tell them for instance, that they should find a way that all persons (objects) involved in the interaction have equivalent tasks to do.

To visualise parameter passing, I use envelopes and paper. I instruct trainees to pick a sequence number for any object created. The trainee writes the number of the object that receives a message on the envelope. He writes on small pieces of paper first the name of the operation, then the value of the first argument, the value of the second argument, and so on. The trainee puts the pieces into the envelope. They should do so even with messages to «this». The pieces of paper inside the envelope make up the signature of the operation. The receiving objects will choose the appropriate method based on the signature and use the values passed to complete the method. The envelope will be returned to the calling object (person) when the operation is completed. The envelope will be returned void or with a piece of paper with the return value in it.

This role-play helps the trainee to build up the schema of collaboration without unnecessary additional information, such as, Unified Modelling Language diagrams or code syntax.

6.5   Conclusion

Collaboration (technically, aggregation and encapsulation in point of view of design) is one of the key concepts of object-oriented programming. With role-plays, it can be teached avoiding cognitive load.

7   Polymorphism, Types and Overwriting Methods

Inheritance, polymorphism, the overwriting of methods in sub-classes is regarded as the essential thing in object-oriented programming. Good and simple examples are therefore crucial.

7.1   Teaching Polymorphism With The Account Example

A good example to demonstrate sub-classes in action is the account example. For different kinds of accounts there are different rules for overdrawing the account and for redrawing money from the account. It is a good exercise to build together this hierarchy of accounts and to overwrite

methods to implement the rules for the different kind of accounts. The hierarchy of accounts should have more than one level. For instance:

First level:

Account {abstract}

> Second level:

> CurrentAccount for businesses, where there are different limitations on overdrawing, according to customers credit limits. No limitations on withdrawal.

> PrivateAccount, limitations on withdrawal, overdrawing is limited to the monthly income (rent, salary)

> Third level:

>> SavingsAccount, like PrivateAccount but overdrawing is not possible.

If you constrain your example to two levels students will speculate, that only two levels are possible.

Obviously, in this account example, there must be an operation to check the amount of money the customer may withdraw. This operation is abstract on the first level and has different methods on the other levels.
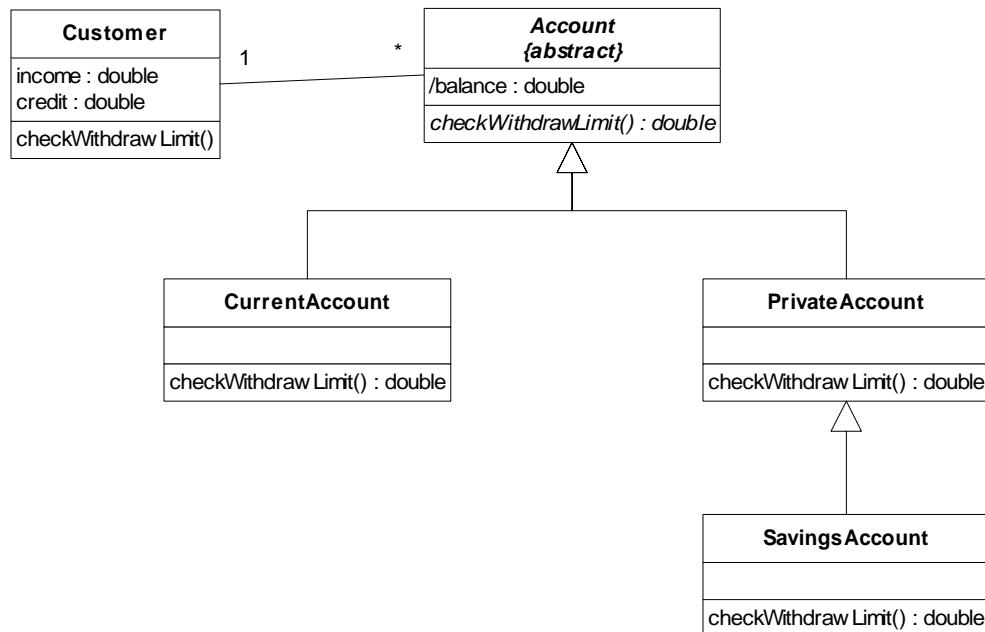
Fig. 5, the account example

Avoid using static constants (static final) to define limits for withdrawal or overdrawing. You may do so later on, when teaching the relationship between types and class variables. But when you teach inheritance, polymorphism and the overwriting of methods, then, your students encounter already enough new information.

In the account framework above you can demonstrate a useful polymorphism by programming «checkWithdrawLimit()» in class customer. You

should get the overall limit from all accounts of the customer. You must refer to the accounts by an array or by a collection («Vector» for instance).

Later one may even use an interface «creditable» with operations like «checkWithdrawLimit()». Before you have such an interface, «checkWithdrawLimit()» is only realised in class customer. Afterwards it is a type. You may demonstrate, that you can mix customers and accounts for instance in an array of type creditable to get the overall withdraw limit of any objects of type creditable you wish.

7.2 Conclusion

There are some evident examples to introduce polymorphism and inheritance. Polymorphism is a complicated and abstract topic. However, it is not as difficult to teach as collaboration.

8 User Interface Related Problems

8.1 Because of its richness, a graphical user interface is difficult to implement. Books and courses avoid it, but DOS and Unix-like code line interfaces also problematic.

8.2 Description of the Problem

Graphical user interfaces (GUI) are very complicated. Setting up a user interface with swing is very well documented in the Java-tutorial. Nevertheless, because of its richness and complexity, it is difficult to understand for the novice. But the novice would like to do real things. This means, he or she would like to write small applications that have a graphical user interface and a code that they understand completely. How can we scope with this wishes and circumstances?

8.3 Need for Visual Feedback—Confusion between Objects and Related Views and Controls on Screen

One goal of graphical user interfaces is to give the user feedback. The user will get feedback, not the programmer. For the novice programmer, the graphical user interface (GUI) is a burden. Because, in a graphical user interface the visual representation should be separated from data. Therefore there is a multi-step process to transform the settings in a visual representation (called control (a text, a slide-bar) into data and data back into representations on screen. Not knowing this fact, novice programmers expect that they can manipulate directly data from TextAreas, Lists or JTables. For instance, they expect that they could use directly a TextArea for a calculation or that they work with a JTable without defining a data model for the Table. How to scope with this problem?

8.4 The «System.out.println()»-Approach

Extensive use of sending information to standard-output (a text-window) helps users to see what is going on. You may later even instruct your people to first set up a data model that you test only by observing the

line-output produced with «System.out.println()». Only when this model works fine, you integrate it in a graphical user interface.

While doing the integration you take care to explain the process from control to data and from data to view.

8.5 Making Input and Output Easier with Java

Java is a GUI-oriented language. Access to input and output is difficult. It is all-natural to lower the cognitive load by avoiding the GUI-overhead and hiding the complexity of input and output in the beginning. Several trainers or authors do this. Judy Bishop (1997) uses a class Text to ease access to files and keyboard. Not all readers appreciate her way to do it. One reader on Amazon.com even says: «We learned Java.gently not Java.» But it cannot be the fault of the approach. K. N. King (2000) who gets for his book an amazing suite of ten enthusiastic five star ratings uses exactly this approach: Avoiding the graphical user interface and using a class to ease input from keyboard and output to screen. One reader writes: «All Java beginner books skip the input function in Java applications. You spend half your time wondering how you input anything into a Java program. Mr King solves the problem by writing a Simple Input class that you import like any other class and makes Java application input simple and fun. This increases your interest in Java and makes it more fun to learn. You can actually write useful Java programs from page sixty three.» King comments his three classes in appendix E (King, 2000 p. 759). He restricts himself to three classes, because as he writes: «I'm not a big fan of instructor supplied classes, so I've kept the use of these classes to a bare minimum.»

Besides writing an input class and importing it, it is also possible to write a class that provides basic services and subclass this class. An example of such a class can be found in Appendix B: Service Class for easy input, output in Java.

As written in the section Read Before Write versus Hiding Details, p. 48, I strongly recommend either to design and comment this Service class well or to hide the content of the class. Joseph Bergin suggests in his teaching pattern for object-oriented programming to work out with the students a toolbox (a set of basic classes for their own work) during the course. Such input or service classes could be part of such a toolbox. While giving the trainees as an exercise the task to work out the input or service class they used in the beginning of the course, you avoid certainly the impression to have learned Java.gently and not Java.

8.6 The Unit-Testing Approach

More demanding is the use of unit tests. In a unit test you test the functioning of all singular methods in a class. Does the method meet your expectation (its specification)? You send to standard out all information about malfunctioning of methods. The framework Junit helps to set up unit tests. Unfortunately this instrument is a bit to complicate for novice user to use it. In a course with intermediate students you may use it. It is

very helpful to visualise what design by contract, specifications and interface means.

With the unit tests you verify by assertions that specifications are met. You make tests to verify whether preconditions and post-conditions are met.

Instead of using the framework Junit one can easily set up his own unit tests. A class with a main procedure that instantiates objects and calls up the operations of these objects is sufficient. Novice programmers will send a feedback line to standard out under all conditions. Experienced programmers will do so only in case of failures to reduce output.

Unit testing is not something trainees do without instructions. Testing must be part of programming permanently. This unit testing gives trainees the feedback they need. They will spend less time with unnecessary functional testing. That means with tests of entire use cases through the graphical user interface.

8.7 Conclusion

Programming Java's user interface is too difficult for beginners. Avoiding graphical user interfaces in the beginning and providing classes that help with input and output lower cognitive load dramatically.

9 Richness of Contemporary User Interfaces

There are ample of components in contemporary graphical user interfaces. For instance with Swing all elements of a modern graphical user interface are available. Avoiding teaching all elements but concentrating on the design aspect of the user interface helps learners.

9.1 The Use of Aggregation in Java

Java does not provide multiple inheritance. Instead you use aggregations. For instance in the graphical user interface of Java:

Every component has its graphics (representation on screen) associated. It is not a subclass of Graphics. The missing multiple inheritance in Java is the reason for this structure.

Every container has its LayoutManager. Other than the relationship between component and graphics this structure allows to combine Container and LayoutManager dynamically. Also you can extend the abilities of all subtypes of containers (frame-windows and dialog-windows, panels) by defining new types of LayoutManagers.

Therefore, there are good reasons to prefer aggregation to multiple inheritance. There are good reasons to regard it as an advantage of Java not to allow multiple inheritance. But aggregation is difficult to understand in the beginning. Therefore, there is a stiff learning curve for beginners starting to early with graphical user interface programming.

9.2 Other Knowledge Inherent to the Java Graphical User Interface

After the learners have acquainted basic knowledge in object-oriented programming then the Java graphical user interface is a good example for

aggregation. It is also a good example for other design pattern. Further it is good material to work with the Java tutorial and the Java documentation. In fact the graphical user interface of Java is an excellent subject to study after the basic knowledge is safely implemented in your students.

So when you start teaching the Java graphical user interface, take care to explain the design pattern used for it. Do not explain all the possibilities of the Abstract Window Toolkit (AWT) or even of Swing. The trainees will not have any difficulties using them once they have understood the basic design pattern used in the Abstract Window Toolkit. There is the Java-tutorial and there are ample of books that help once the basics are understood.

9.3 Conclusion

User interfaces are an interesting field for studies, once the trainee has learned basic design patterns and concepts.

10 Chapter Summary

Compared to procedural programming, object-oriented programming causes not only different problems. It causes also much more problems in the very beginning of a course. Java adds even more problems by its complicated graphical user interface and its poor support of a code line user interface.

Chapter 4: Existing solutions to overcome the problem of cognitive load

There are several ways to work around the problem of cognitive load when teaching Java. You can ignore object-oriented concepts when teaching Java in the beginning. Give a sound theoretical background first. Start with software design with the Unified Modelling Language. Start with single classes and test them against test cases the teacher provides.

The usage of microworlds is compared to such alternate possibilities.

1 Ignoring Object-Oriented Concepts When Teaching the Language Java

There are three big pros for this approach: the common basis of procedural programming and object-oriented programming is the one, the availability of tested training material the other, the possibility to do interesting projects in Java without object-oriented programming, the third.

1.1 Common basis of procedural programming and object-oriented programming

Firstly, there are only advanced topics of procedural programming that are not relevant for object-oriented programming. So learning procedural programming with Java or C++ gives a good base for programming with Java. After the student is familiar with the basic principles of Java, there is less cognitive load when moving to object-oriented afterwards. The highest rated book written in German for learning Java (Boles, 1999) follows this approach. Even adepts of object-oriented programming suggest this book for education at technical universities before plunging into object-oriented programming with a book that gives a sound technical background for object-oriented programming (Gool, 1999). Also if you take a closer look on most basic books about Java programming, you find that they cover elements that procedural programming and object-oriented programming have in common. They program procedurally inside the class operation main() ignoring most of object-oriented concepts.

1.2 Availability of material and workforce

Secondly, and for long time most important, you can entirely rely on material that you used earlier for teaching a procedural language like Pascal. Fore instance, if you do not have experienced object-oriented programmers as teachers and sufficient learning materials at hand you run into troubles with a course for object-oriented programming. I acknowledge that I myself have been working with procedural programming for years and I haven't made the move to object-oriented programming completely. One of my colleagues at our institution for adult education made for a

case study in object-oriented programming a model solution. Analysing the code you can tell that this colleague hasn't made the transition from procedural programming to object-oriented programming either. I guess the shortage of material and teachers will continue for some years.

1.3 procedural programming projects with Java

It is possible to create all-in-one-class procedural solutions for smaller projects. For instance, with an applet, the student can create interesting applications. He takes the inherited methods as if they were predefined procedures in a procedural language.

1.4 Draw backs of this approach

There is some drawback to this approach. Object-oriented programming is still quite different from procedural programming. When setting up a time budget for your course you might carefully plan where you can spare time for teaching the most important concepts of object-oriented programming. You may check existing course books for topics related to procedural programming like decision tables, Nassi-Schneiderman diagrams, recursion, side effects and modules.

Experience shows that learners show a big affinity for a structural approach to solve some programming problems. With this approach you may reinforce this behaviour. Also learners are bewildered by this entire «unnecessary» object-oriented overhead around the operation main().

Joseph Bergin, 2000b, in his online pamphlet «Why Procedural is the Wrong First Paradigm if OOP is the Goal» mentions several reasons, why this approach should be wrong. He mentions that developers used to procedural programming need one or more years to grasp the paradigm of object-oriented programming. He also mentions the different strategies to design structured programs and object-oriented programs.

1.5 Conclusion

In fact strategies for procedural programming and object-oriented programming are different as discussed in «Problem Solving Strategies Compared», page 11. But why not teaching the basics of Java programming as procedural programming, like Dietrich Boles, 1999, does, during the first months of education, before making the move from procedures to methods, from overall control to client server thinking? —"Procedural programming first" promoted from Gibbons, 1998, for instance might not be absolutely the best way, but it might be the best way under circumstances that are not unlikely. I would therefore suggest this approach only if you meet two prerequisites:

1. You can really relay on good and tested material on procedural programming but not on object-oriented programming.

2. You have enough time to do the second step from procedural programming to object-oriented programming.

In a Java short course, there is not enough time usually. Therefore, this approach is problematic.

2 Giving a Sound Theoretical Background Firstly

2.1 Description

Giving a sound theoretical background first is done in lots of books and courses. Either After teaching some basic programming in Java or in the very beginning. In any case, it should be done before instantiating objects and defining entire classes. To explain objects technically to novices is almost impossible. Therefore most authors rely on to metaphors to explain objects, classes, inheritance, polymorphism or objects in action. Therefore, I discuss at this place metaphors comprehensively.

2.2 Judgment of this Approach

According to the theory of cognitive load this should be a good approach. The learner is not forced to split attention and there is no redundant information.

But, practical experiences are somewhat disappointing. Even if the teacher uses good metaphors, he may not gain more then having taught some basic words. Whether he uses metaphors or rather stick to more technical explanations, there is little hope for successful transfer of the knowledge to a situation where the trainee actually has to write code. This means code that uses techniques of object-oriented programming, like instantiating objects and exchanging messages. The main problem is that trainees stay inactive. Therefore, there is little likelihood that they acquire new schemata.

2.3 Why and When Metaphors Do Not Work

Jean Piaget first recognized the power of metaphors. He made a link between the struggle for survival in biology and the struggle for survival in learning situations. Firstly, beings are trying to implement their existing strategies for survival in new situations. They pick the strategy for the most likely situation they know. This is known as assimilation. So the main reason to pick a metaphor is to make the trainee do most likely the right thing or something that is not completely erroneous. If he does something wrong, the trainee must get a feedback that makes him adopt the right behaviour. This is called adaptation. Therefore metaphors without activity are little help. Metaphors will only do their job if the trainee is active and gets enough feedback so that adaptation takes place.

2.4 There Is No Evident Metaphor

The high variety of metaphors used in object-oriented programming shows that there is no evident metaphor to explain objects and inheritance. Some of the metaphors used are listed below:

- In the Java tutorial (1996, through online 2001) in chapter «Object-Oriented Programming Concepts: A Primer» the metaphor of bicycles is used.
- Eckel uses a light bulb (Eckel, 1997, p. 29) to explain an interface.

- The popular German online documentation SelfJava (Schröter 2001) uses cars and different types of cars as a metaphor when explaining inheritance. Similar Niemann (2001, p. 71) uses vehicles in general divided into ships and land vehicles.
- Bishop, 1997, uses Nature as basic type for Birds, Trees and Animals (with subtype Carnivores and Herbivores).
- Goll, 1999, uses SteamBoat (p. 38), Person and Student (p. 40), Machine with subtype FaxMachine, Computer (p. 44).
- Lemay, 1999, uses first trees, like Bishop does, but switches to monsters, with subclasses FlyingMonster and WalkingMonster.
- Bergin, 2000, in his «An Object-Oriented Bedtime Story» uses the metaphor of people doing services for other people. People are objects of different types. Bergin's Bedtime Story is especially interesting, because in his metaphor he uses aggregation and object's interactions. He explains the object's role as client (a patient) and server (a doctor).

2.5   About Effectiveness of Metaphors

Iding (1997, p. 249) reports based on several studies that analogies foster learning. Analogies can be good if students receive information about their limits. This lets students better map the parts of the metaphor that are applicable to the new situation. Not surprisingly, for semiotic researchers, she reports three things: (1) metaphors do support the transfer of knowledge, (2) it is preferable to use several analogies instead of one, (3) teachers should explain how to map the analogy to the actual problem. Sign systems and semiotic morphism are discussed more deeply in the section, Semiotic Research on Mathematical Sign Systems, p. 89.

2.6   My Personal Preference

As there is no really evident metaphor, authors follow their personal preferences. I always use companies that provide services as metaphor for object-oriented programming like in the doctor–patient example of Bergin (2000), whereas old-fashioned industries are a metaphor for procedural programming.

In companies that provide services, there are always user-client situations. People are exchanging messages. They keep their data at their work place. Usually, you do not use files directly from other people's desks. You rather ask them, to provide information. Or to process a request based on the information they keep in their files or on their desks. Employees have responsibilities and they are collaborating.

In procedural programming you have data, before it is processed (input, basic or half fabricated material). Then you process the data or material. This process may be split up into sub processes. At the end of the fabrication process you got the final material, the processed data or output. You are free to change the fabrication process. But when you change input or output, you are forced to rather large adaptations in the fabrication process.

I prefer these metaphors to show that procedural programming is not obsolete. It is quite efficient, as long as there is a large and stable demand,

like in the production of cars or in banking transactions. It is rather inconvenient, when customers demand and the knowledge in the field is changing permanently.

2.7 Metaphors and Microworlds

Bergin mentions that metaphors brake at some point. He writes Bergin, (2000):

> «Metaphor is important since it helps you with a framework for thinking about something unfamiliar. Every metaphor breaks down at some point, however.»

You may think of the robots in the microworlds of Joseph Bergin as objects. I will discuss this metaphor more deeply later in the text, see Chapter 7: Explaining basic concepts with Roboworld, at page 84. I personally believe that the robot (object), beeper (primitive data), microworld (external storage) metaphor will brake at almost no point. Plus the trainee works in the metaphoric situation. There adaptation can take place.

Only, there still is some transfer to teach, when moving from a microworld to real programming. But this transfer is a feasible step not a jump over the ocean.

2.8 Books That Do Not Use Metaphors and Explain Objects Directly

Eckel (1998, p. 27, online 2001 chapter 1) refers to Alan Kay and explains an object directly as: «Think of an object as a fancy variable; it stores data, but you can also ask it to perform operations on itself by making requests.» But this book is not regarded as a book for beginners. This is at least what the feedbacks of beginners at amazon.com say.

2.9 Basic Programming Knowledge before Getting to the Paradigm of Object-Oriented Programming

As already mentioned, some books and teachers give first an introduction to parts of Java that are not related to object-oriented programming (data types, expressions, statements, control statements, etc.). This is an intermediate approach between giving a sound theoretical background firstly and teaching procedural programming firstly. The cognitive load is lowered a bit this way, but the problem how to introduce the paradigms of object-oriented programming still remains unsolved. So this approach is only a variation to giving a sound theoretical background first. Among the books that follow this track are Judy Bishop's Java Gently (Bishop, 1997), Herbert Schildt's Java 2 A Beginners Guide (Schildt, 2000) or in German Guido Krüger's GoTo Java 2 (Krüger, 2000).

2.10 Conclusion

Despite, all the good ideas and efforts to find convincing metaphors, there is little success in this approach. The level of learner's activities is too low and the space of time to writing code is too far, so that either learning of schemata does not take place or that the transfer to implementing code

gets hindered. Therefore, trying to find the ultimate metaphor is not worth the effort.

3 Cookbook Approach

To get a fast success, a lot of books use the cookbook approach. The cook-book approach uses different elements to support learning.

3.1 Overview of this Approach

In a book following this approach, the text explains—usually with one example—how to do complete programming tasks. The book gives a code model, how to complete the task efficiently. Therefore, there are two elements: example code and how-to-do prescriptions. The Java Tutorial (Campione, 1997) or the very popular German written GoTo Java 2 (Krüger, 2000) are examples for such an approach. Also, there often is complete background information about the most important aspects of the implementation.

3.2 Example Code

The most important part of this approach for learning is the example code given. The reader or trainee can either reconstruct this example, or—based on the model—construct his variation to the model given in the book. I, personally, copy the chunk of code into my project and replace the names of the identifiers.

3.3 How-to-Do Prescriptions and Background Information

Often, things cannot be done with just one or two chunk of code. This multiple steps the student need to complete often involve several classes. Sometimes they involve file handling or even installation procedures. Some of this steps are highly interrelated with other questions. Therefore, often, a high amount of background information is needed. Some on this information can be found in the text.

3.4 Advantages of this Approach

There is often a quick practical success. If the new information involved in the model stays small and the background information covers what is new for the trainee, this approach is the choice. For instance, if the trainee understands the basic concepts of a graphical user interface and its implementation in swing, he or she can easily use most elements of the swing package with the help of the Java tutorial.

Trainees learn to read manuals and to read and adapt code examples. This is an important technique in object-oriented programming. Joseph Bergin (2001b) points out in his pedagogical pattern Read Before Write:

«You are teaching an elementary course that has a strong programming or design component. You want to help them learn to eventually create large and complex programs. Creating anything is hard work, even for the skilled. Novices, on the other hand lack these skills. However, as with natural language, stu-

> dents have an ability to read and understand larger artifacts
> than they can be expected to create. They can also learn about
> structure, scope, and aesthetics from reading great works. In
> an English course, for example, students read and analyze
> Shakespeare's plays, but are not expected to produce such
> works.»

If chunks of code are not to large, this Read before Write element, can be recognized as the worked example effect mentioned in cognitive load theory. Bergin suggests rather using large case studies than a textbook with a cookbook approach. It seems to me to be an important question how the worked example effect is best integrated in programming: Giving large project to read, which show the important chunks of code in context, or using very small examples that give a minimum context to understand the single chunk of code that the student should study?

In any case, this Read before Write (or worked example) effect is a big advantage of this approach.

3.5 Software Learning and Chunk of Code

Learning how to program is highly related to learning code elements that are used repeatedly. Some examples of important chunks in Java:

Usage of the chunk	how the chunk looks
| Starting point of an application | ```java
public class NameOfClass {
        public static void main(String[] args) {
                <application starts here>
        }
}
``` |
| Query method | ```java
typeA variableNameB
typeA getVariableNameB () {
        return variableNameB;
}
``` |
| Update method | ```java
typeA variableNameB
void setVariableNameB (typeA variableNameC) {
        variableNameB = variableNameC;
}
``` |
| Looping for a fixed number of times | ```java
for (int i = 0; i < times; i++ ) {
        <repeated task>
}
``` |

Table 2, typical chunks of code in Java

This chunks represent a medium level schema, which is stored in the programmers long term memory. Such chunks are at about the same level as often used sentences in natural languages.

3.6 Major Disadvantage of this Approach

In a rapidly moving course, there is a great risk that trainees jump from topic to topic without understanding any fundamental concept. They even not stock any chunk of code. The teacher may have the feeling that trainees are doing well, because they succeed in generating nicely looking applications by copying large part of code, and imitating code models with

some support of front runners and the teacher. I remember a course, where I did some repetition on remote method invocation (RMI). The main problem was, that students did not understand method invocation with the passing of parameters and return values. Sixteen trained students had no idea how to pass more than one value of a primitive type back as return value. They did not know what query and update methods were and how they really worked.

3.7 Psychological Reason for the Weakness of this Approach

In the view of cognitive load often, the attention is not splitted. Crucial code elements are commented directly in the code example. So one would expect that this approach should do fine. But, there is very often a large amount of unnecessary information. Usually, the code examples demand a basic understanding of a lot of concepts. If these concepts were not understood, learning is hindered by redundancy.

Redundancy might be one reason for the weakness of this approach. The main problem, as it seems to me, lies somewhere else: There is a high risk that trainees stay inactive: just read theories and do not try to apply their own knowledge to new situations. Or, that their activities remain on a low level: typing and copying code, trying to fix code without exactly knowing what is going on, receiving solutions from teacher and peers, rather than fixing code themselves.

3.8 Conclusion

Be careful! —Learning how to use cookbook solutions is certainly an important objective. Also to learn from case studies (Read before Write) is a good idea. But struggling the way through an endless series of cookbook examples is definitely not the way to success.

4 Spiral Approach

4.1 Definition of the Spiral Approach

Bergin (2001) explains this approach as follows:

> «Topics in a course are divided up into fragments and the fragments introduced in an order that facilitates student problem solving. Many of the fragments introduce a topic, but do not cover it in detail. Just enough detail is given initially so as to form a basic understanding that can be applied to problem solving. Additional cycles contain reinforcing fragments that go into more detail on the topic.»

So, there are two key elements that are crucial for the Spiral Approach:

1. Divide the topics into fragments. These fragments do not cover a topic in detail as a reference book does.

2. Problem solving should be possible with these fragments.

Microworlds as discussed in this thesis are a typical example for a spiral approach. But, one can find other excellent examples.

4.2 Kim N. Kings Spiral Approach

An excellent example of the application of the Spiral Approach for teaching Java can be found in King (2000). King motivates his approach (King, 2000, p. xviii) as follows:

> «I've employed a spiral approach to many topics, gradually adding detail over the course of several chapters rather than covering each topic in a single place. The treatment of objects and classes is spread over Chapters 3, 7, 10, and 11, for example. Control structures are covered in Chapters 4, 5, and 8.»

In fact in his curriculum he often turns back to important topics, see on next two pages, Table 3.

Chapters in King's textbook	Difficult topics of object-oriented programming				
	Common topics for structured and object-oriented programming		Topics of object-oriented programming		
	Variables	Method call and return	Objects and object references	Inheritance and polymorphism	Objects Collaboration
1. Getting Started					
2. Writing Java Programs	2.4 Using Variables, 2.5 Types 2.6 Identifiers, 2.8 Constants	2.9 Methods			
3. Classes and Objects	Declaring instance variables in (3.3 Classes)	Declaring instance methods, method overloading, constructors in (3.3 Classes) 3.5 Calling Instance Methods 3.9 Java's String Class (Common Methods, Chaining Calls of Instance Methods)	3.4 Creating Objects 3.7 How objects are stored		
		3.8 Developing a Fraction Class			
4. Basic control structures					
5. Arrays			5.1 Creating and Using Arrays 5.5 Using Arrays as Vectors 5.6 Using Arrays as Databases 5.7 Arrays as Objects		
6. Graphics		6.1 Creating a drawing 6.4 Combining text with graphics	6.2 Drawing in colour: Construct a Color object 6.3 Displaying text: Construct a Font object		6.2 Drawing in colour 6.3 Displaying text 6.4 Combining text with graphics
7. Class Variables and Methods	7.5 Class Variables	7.1 Class methods versus instance methods 7.2 Writing class methods 7.3 The Return statement 7.4 Parameters 7.7 Writing helper methods 7.8 designing methods	10.4 The this keyword		
8. More Control structures		8.1 Exceptions			
9. Primitive Types	9.1 Types 9.2 Integer 9.3 Floating-Point 9.4 Char 9.5 Type Conversion				

| Chapters in King's textbook | Difficult topics of object-oriented programming | | | | |
| | Common topics for structured and object-oriented programming | | Topics of object-oriented programming | | |
	Variables	Method call and return	Objects and object references	Inheritance and polymorphism	Objects Collaboration
10. Writing Classes	10.2 Choosing instance variables 10.4 The this keyword (using this to access hidden variables)	10.4 The this keyword (using this as an argument or return value) 10.10 Debugging	10.4 The this keyword 10.5 Writing constructors	10.1 Designing Classes	
	10.7 Adding class variables and Methods				10.3 Designing Instance Methods 10.8 Reusing Code
11. Subclasses		11.4 Polymorphism		11.1 Inheritance 11.2 The protected access modifier 11.3 Overriding 11.4 Polymorphism 11.5 The Object class 11.6 Abstract classes 11.7 Final classes and methods 11.8 Inheritance in the AWT	
12. The Abstract Window Toolkit		12.7 Creating and using components		12.2 Frames 12.3 Event Listeners (how to implement an interface)	12.2 Frames (adding Components) 12.4 Inner Classes 12.5 Attaching Listeners to multiple components 12.6 Layout
13. Data structures	13.3 Wrapper Classes	13.7 The StringBuffer class	13.1 Multidimensional Arrays		13.2 The Vector class 13.6 Sets
14. Files		14.4 Advanced exception-handling		14.5 Reading an writing data types 14.6 Reading and writing characters	
			14.7 Reading and writing objects		

Table 3, Topics mapped to chapters in a book that uses the spiral approach.

Joseph Bergin (2001) emphasises the problem solving aspect of the spiral approach. Students should be able to address real world problems with the knowledge they learnt in every iteration. King (2000) tries to help his readers (students) to do so with: Teaching problem-solving skills, some program design, code writing style, and give real world examples that are interesting enough and not to large. Some of his case studies are: Unit conversion, course average, checking an ISBN number, decoding social security number, a phone directory, printing a one month calendar, playing blackjack.

4.3 Cognitive Load of this Approach

This approach reduces dramatically the cognitive load. A lot of care is taken, to avoid redundant information. King defines in his book exercises that enforce the students to learn the schemata they need later in the book.

4.4   Conclusion

The example shows that a Spiral approach is possible also without microworlds. But, the price is high. King needs 370 careful written pages before he gets to the main topics of object-oriented programming (writing classes and subclasses). SAMS famous Java in 21 days covers the same topic in the first week. Nevertheless, I would highly prefer Kings careful approach to the way other books about Java treat the topics. Besides Bergin's description of a microworld that is discussed later, this is the only book that complies with the implications of cognitive load theory. A tremendous rating of this book by amazon.com's readers shows some evidence for the validity of cognitive load theory.

There is a draw back to the Spiral approach in general. It needs a lot of planning and elaboration for the teacher/writer. King needed fore years to do the job properly. As Joseph Bergin (2001) emphasises: «This pattern (the Spiral approach) cannot be used in a small way. A commitment needs to be made to it. If this is not possible or desirable, avoid it entirely.»

5   Teaching Object-Oriented Thinking with UML Firstly

The Unified Modelling Language (UML) is a way to visualise object-oriented problem solving and design. To these methods one can also count the Class Responsibility Collaboration Cards (CRC-Cards). CRC-Cards are an entry level to both the static UML diagrams like class diagrams or the dynamic diagrams like statechart or collaboration diagrams.

5.1   Class Responsibility Collaboration Cards (CRC-Cards)

Kent Beck and Ward Cunningham presented CRC-Cards first at the OOPSLA '89 Conference (Beck and Cunningham 1989). They write:

> «The most difficult problem in teaching object-oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their task.
> … It is in this context that we will describe our perspective on object design, its concrete manifestation, CRC (for Class, Responsibility, and Collaboration).»

CRC-Cards are used as follows:

1. You write down on a paper card (A5 or smaller, i.e. 10*15 cm) the classes you find relevant to your project (in top). For instance in a project for a bank: Account, Customer, Current Account, Value, Performance, Shares, Portfolio might be things you find relevant. Beck and Cunningham 1989 mention a banking machine and suggest in their solution: Account, Transaction, CardReader, Dispenser, RemoteDataBase, Event, Interface FSM, Screen and Action.

2. On the left hand side of the cards: You write down the tasks (responsibilities) you think are natural for this class. For an account natural responsibilities might be: deposit, withdrawal, check current value, calculate interest, show interest rates history, show history of transactions.

3. Opposite on the right hand side, you make a note that tells which classes work together to accomplish this task (collaboration). For instance to show the history of an account you need all the transactions on this account. You do not mention collaborations if a class can fulfil some task without another class but may work together with another class. For instance Beck and Cunningham 1989 mention that in the model-view-control-pattern the model has no collaboration with the model or the control. Collaborations with them are not a must, only a possibility. The model may be driven without a graphical user interface.

Afterwards, CRC-cards can be used in three ways:

1. You can derive static class diagrams from these cards.

2. And most important: you may set up a team in which everyone plays an object of one of these classes. Then you try to accomplish a task. From this role-play you can derive dynamic diagrams as statechart or most likely collaboration diagrams or sequence diagrams.

3. Experienced programmers can transform CRC-Cards directly into code.

It is in the second way where CRC-Cards may be used in education.

5.2 Statechart Diagrams

Learners are confused by a high number of objects working together. A statechart diagram studies only one object in different states of its lifecycle. Learners understand simple statechart diagrams long before they completely grasp the sense of attributes and operations. Working with statechart diagrams is therefore a powerful tool to guide learners from their real live understanding of the behaviour of objects to the functioning of objects in object-oriented programming.

A fuel tank is a good example for an object with states that students understand easily. The tank may be full, empty, filling and ready. It has two attributes: capacity and level. Operations are consumeFuel() and fill().

Even with little programming knowledge, learners may set up the code for the fuel tank and see the tank in action later in the course.
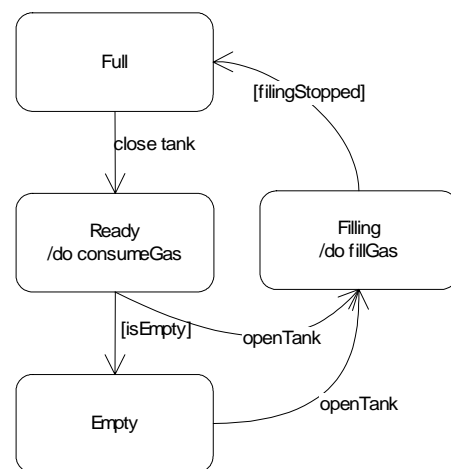

Fig. 6, statechart diagram of a fuel tank

5.3 Static Diagrams

Static diagrams such as class diagrams and object diagrams are the most abstract to understand. They only seem to be easy for experienced pro-

grammers, because they hold about the same information as entity-relationship diagrams in database design, and because they are close to object-oriented code. But without understanding object-oriented programming, these diagrams reflect a lot of concepts of object-oriented programming that the learner does not know yet. Books, therefore, use metaphors to explain the concepts. It seems unlikely that understanding these metaphors really helps understanding object-oriented programming. It is easy to understand that there is a class (category) vehicles and that in this class (category) fall boats, cars and motorcycles as subcategories (subclasses). No problem—expect for students that do not speak English well enough—to see that myMotorCycle, myMotorBoat, myRowingBoat fall in one or several of these categories and that they are instances (objects) of vehicles, motorcycles and/or boats.

No problem to see that a car is a composition of motor, trunk, passenger's cell and 4 wheals, etc.
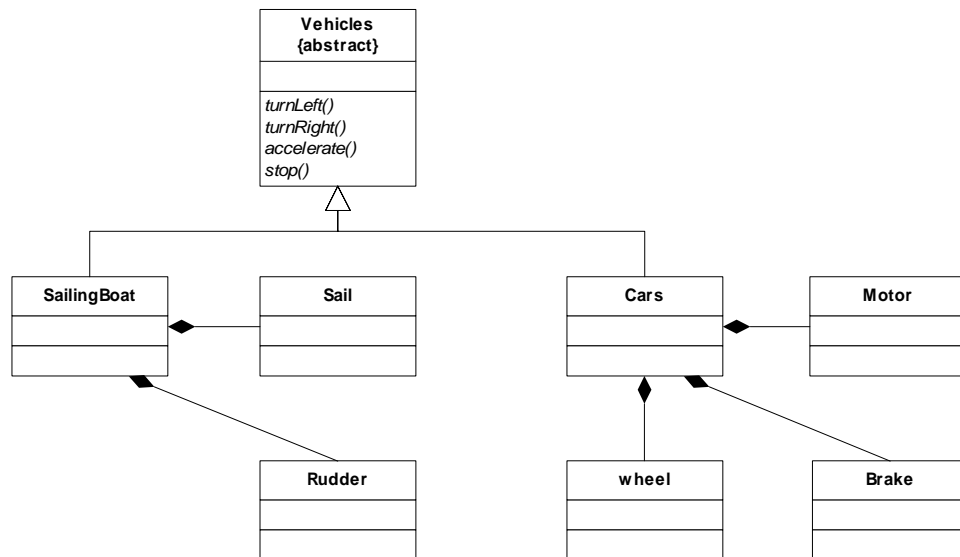

Fig. 7, class diagram, figurative example, but assigning behaviour does not work

Problems start when transferring to real object-oriented behaviour. You see arise big question marks when you explain that all vehicles must know the operation turnLeft() and turnRight(). Or, when you explain that they use different methods to fulfil these operations.

The main benefit of static diagrams is that people learn the vocabulary well. But still without coding, the transfer from static charts to code and the real understanding of the behaviour of objects in object-oriented programming is not sure at all.

5.4 Other Diagrams

It is a little dangerous to use other diagrams to visualise elements of Java code. Judy Bishop (1997) tries it in here book «Java Gently». Some readers find her specific object modelling technique fine, because the diagrams

give them an overview of the example programs at one glance. For most others, they mean another burden added to the heavy cognitive load.

5.5 Judgement of this Approach

We tried this approach in our institution for adult education. It is not a bad approach. We needed one week (30 lessons) to do it. CRC-Cards and statechart diagrams are most useful when you follow this track.

With UML and especially with CRC-Cards one can involve learners in activities.

5.6 Good Activities for Working with CRC-Cards and UML

Reading comprehension of diagrams

- Explaining what you see in a diagram in your own words.
- Given a diagram and different statements, trainees decide which statements are true and which are wrong.

Drawing diagrams

- Given some statements in natural language trainees design diagrams.
- Given a class diagram trainees draw object diagrams that are compatible to the class diagram or vice versa. This is very important to understand the difference between objects and classes.
- Given a sequence or collaboration diagram trainees draw class diagrams that contain as operations the messages sent between objects.

Playing objects involved in the program flow

Given is a task to complete and the objects (or classes) that collaborate with each other. The trainees have to find a program flow for the collaboration. The rules you give for this exercise:

1. You communicate only with messages.
   A message contains: the addressee, a command and additional information.
2. The addressee takes over complete control and keeps it until he gives it back to the sender of the message.
3. The addressee may call other objects (or classes). He may also call himself.

5.7 Design Pattern and Analyse Pattern

Design pattern are pattern that are independent from a problem domain, and therefore can be used in different applications. Such pattern proofed to be too abstract for beginners. Even obvious patterns like composite-component (described at Gamma, 1994, p. 163) are difficult to understand. I used the visualization in the book of Fowler (2000, section 6.2) that shows a object diagram and the related class diagrams and asked to transfer the idea to the example of another type of organisation: unions and their members. The trainees had sever difficulties to do this transfer.
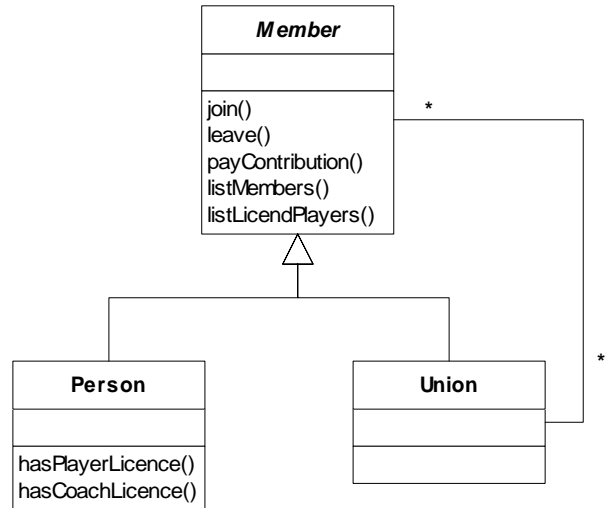

Fig. 8, example of the composite-component design pattern
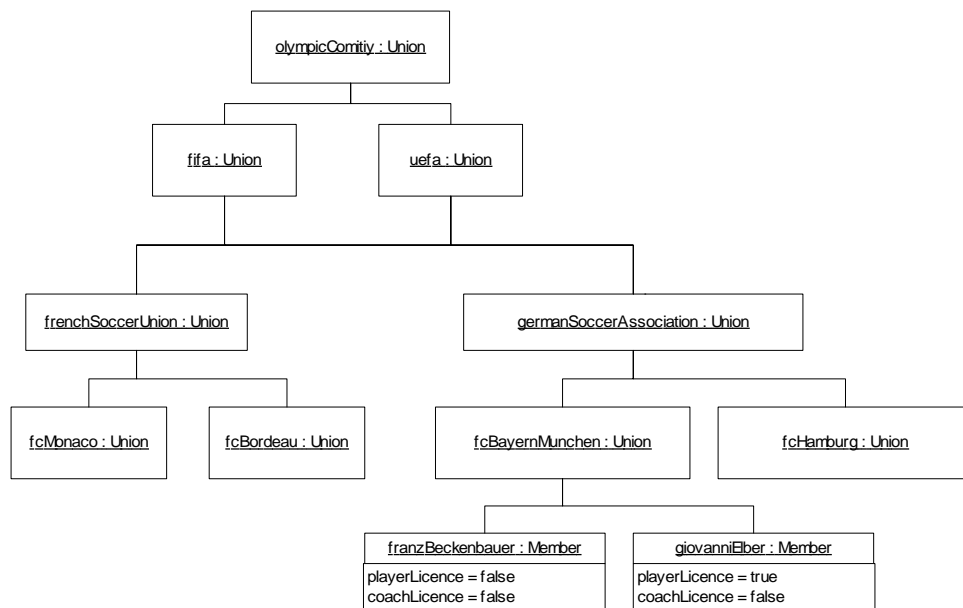

Fig. 9, object diagram to the class diagram in fig. 8

Analyse patterns with simple real live problems like customer, account, transaction showed to be preferable for exercises.

5.8   Examples and Exercises

A teacher should provide examples that are easy to understand and use the same examples for all the different kinds of diagrams.

Good examples are not easy to find:

- The example of the different vehicles for instance works fine with static diagrams or even a statechart of a vehicle, but it is rather abstract when you are going to set up CRC-Cards and test some program flow. What is the responsibility of the wheel? Which classes got the responsibility for a turn? What is the program flow of a turn?
- Examples like a library, or a bank work fine but need some understanding of data handling and tend therefore to become too complex. For trainees with a background in database programming, such examples should work fine.
- We thought that simulations should work fine. We worked out an example with a washing machine. But the example was not specific enough. Therefore, the trainees could not find a solution. Games might be good examples. I have not tried this yet.

5.9   At our Institution for Adult Education We Gave Up this Approach for Two Reasons

1. The feeling that the knowledge is too abstract for beginners.
2. The experience that learners were not able to learn the diagrams without the background knowledge of an object-oriented programming language.

5.10   Starting with CRC-Cards and Unified Modelling Language in the Light of Cognitive Load Theory

CRC-Cards have only three semantic elements and the procedures to use them can be acquired quickly. The usage goes step-by-step. Important to note, you use them in a goal free process. The first instruction is to write down objects classes. Then you think about their responsibility. Then you think about with whom the should collaborate to fulfil this task. Therefore cognitive load is low. There is no split of attention, no goal, and no redundancy. The learners can play the interactions so that a high degree of involvement and learners activity is guaranteed. Bergin (1998) and my own experience are good what gives some support to cognitive load theory.

The charts of the Unified Modelling Language show more, but not an overwhelming amount of, semantic elements. Our experiences were good with statechart diagrams, fair with static diagrams (objects and classes), and mixed with dynamic diagrams like sequence and collaboration diagrams. The latter may be explained with the redundancy when introducing two types of diagram for the same purpose in the same cycle of a course. The better experience with statechart diagrams may be implied by the fewer interacting elements. Statechart diagrams show only one object. So, the experience we made with Unified Modelling Language also supports cognitive load theory.

We introduced also use-case and activity diagrams. The experience with these two types of diagrams were poor. They are not so closely related to the paradigm of object-oriented design. I would avoid presenting this material in the same course cycle in the future. I would use user stories instead of use-cases, because students always tried to define several use-cases even for very small projects.

5.11 Conclusion

Based on reported and own experience I can say that CRC-Cards support the learning of basic concepts of object-oriented programming. This gives support to cognitive load theory, mainly because of CRC-Cards goal free usage. Some diagrams of the Unified Modelling Language can help to learn schemata of object-oriented programming. This is especially true for statechart diagrams. Other static and dynamic diagrams show more interrelated elements. Introducing three kind of dynamic diagrams, statechart diagrams, sequence diagrams and collaboration diagrams, was redundant and might be a reason for the poor performance with this kind of diagram. Also these experiences give support for cognitive load theory.

6 Giving a Single Simple Task within an Existing Object-Oriented Framework

There are three main approaches on this promising track. One is to take an real simple project and ask the trainees to solve one single problem within this project. The second is to write unit tests first and ask the trainees do define classes or part of classes that comply with these tests. The third and most interesting are microworlds where you ask the trainee to solve problems in the microworld. These three possibilities are explained in the next three sections: Trainees Solve a Single Problem within a Project, Setting up a test class, and Using Microworlds for Training.

7 Trainees Solve a Single Problem within a Project

7.1 Key Idea

In object-oriented programming responsibilities are shared. Therefore your students may work on a small part from a larger project. Joseph Bergin calls a predefined application or framework as basis for learning an artefact. He calls the technique of fixing minor flaws in this artefact a Fixer Upper. He mentions several ideas of how to use it. You can implement these ideas simultaneously sometimes. Bergin (2001) lists and describes:

«

- When the artefact is introduced at the **beginning of the course** and introduces the **key ideas** of the course, then it is also an **Early Bird**
- When the artefact is **very large** then it is also an instance of **Larger Than Life**
- If the artefact is also **illustrative** of some **important topic**, then it is also an instance of **Toy Box**

47

- When the artefact **illustrates the overall topic** of interest in the course, it demonstrates **Lay of the Land**
- If the repaired artefact is **useful in some larger** context it can also be a **Tool Box** instance.
- This has some of the same goals as **Mistake**, though it is approached differently. Here we find errors. In Mistake we **make specific errors.**
- This pattern is a way to **achieve Read Before Write.**

»

For all this ideas (teaching pattern) Bergin explains when and how to use them. An example of a FixerUpper could be a class Person where the attribute for the last name is missing. The student has other attributes with their query and update methods as worked examples already written in the class. Now, he can add the attribute for the first name with the related methods with very little cognitive load.

7.2   Read Before Write versus Hiding Details

Bergin emphasises that learning to read (code) before you write (code) is very useful in teaching object-oriented programming. An adverse experience of Java trainers is that students loose a lot of time studying code details they do not understand yet.

Cognitive load theory suggests rather hiding details that students do not understand completely, because they add to the cognitive load.

Experience with language teaching may give some idea why it may be all right to reveal details. The technique of Total Immersion (Listening without understanding firstly, before taking, that means the way children learn languages) is a good way to learn languages. It might be a good way to learn computer languages too.

The diverse experiences depend most likely on the material you provide. If the material is excellent Total Immersion (Read Before Write) works fine. According to my experience the quality is based on:

- Consistent naming and coding
- Code comments where needed
- The context is evident, for instance the responsibility and the collaboration of the class are explained.
- The code is concise but avoids abbreviations that execute two steps in a single command. For instance, it should rather reed
  « i++; while (i < max) … » than
  « while (++i < max) … ».
- Methods are not too large

The Java Tutorial of Sun Microsystems contains a good example of useful code as stated above.

A lot of material is not good enough for Read Before Write. Therefore, you would better not give access to the source code of this part of the project. Otherwise, trainees loose lots of time studying code that does not

have the quality to be used for Read Before Write. This might be very inefficient. Plus, it gives trainees a bad model on how to write code.

7.3 Support for Cognitive Load Theory

Evidently, giving the trainees a single problem within a larger project, he or she can focus on specific problems and the schemata that may be applied to them. They do not only learn the schema to solve the problem, but also to recognize the schema within a real live scenario. Both things do not involve unnecessary redundancy. The first schema is to scan for the specific problem, the second to solve the spotted problem. If the training concentrates on one and only one problem at a time, it is an almost perfect application of cognitive load theory.

A little bit puzzling is the good experience reported by Bergin of the teaching pattern Read Before Write. To reveal details should cause problems according to cognitive load theory. Experiences of my colleagues suggest rather the hiding of details (providing classes only compiled, and without access to source code). Bergin rates the validity of his pattern with zero to two stars Read Before Write has a one star rating, which means medium validity.

7.4 Conclusion

Focusing on one problem within the framework of a larger project is an excellent example of the application of the cognitive load theory. It can be associated to the worked example effect. Giving access to the complete source code of the project might cause lots of problems if the quality of the code is not excellent.

8 Setting up a test class

8.1 What is a Test Class?

A test class is a class that executes automatically several tests on another class without interruption and reports any unexpected behaviour (failure) or run time errors.

There is a special free tool available to construct test classes. It is called JUnit (http://www.junit.org). The tester can execute as many test-suites as he likes in one run. The tool assembles all failures and errors in the test result. The test result can be checked in case of unexpected behaviour (failure) or of run-time errors. Java's complete stack trace is recorded and can be checked for any failure or error.

8.2 How Trainers Can Use Test Classes

The trainer defines the specification for a class. He sets up one or several test classes, so that all mistakes that he expects from the trainees are checked. It is important to give accurate feedback in case that a failure or error occurs. The trainer asks the trainees to write the code of the class. Trainees do their coding so that the specification set up by the trainer is met. The JUnit-Framework is very convenient for such trainings. It gives the trainee a visual feedback; complete access to the stack trace in case

of failures and the trainer can define a feedback text for any failure he expects.

8.3 Advantage of the Test Class Approach

The trainer does not need a whole project. The idea of a single class and its specifications are sufficient. If the class should not use the existing API, classes like «Temperature» that store the value of temperature, «Fraction» that stores two integer values, or «Account» that stores account numbers and the balance are good examples.

By setting up a test class, you can show what specifications are and that specifications must be met and tested. This can be done by the way without loosing time for long theoretical discussions about specifications and testing.

8.4 Experiences with this Approach

I used an example that was too difficult (Direction). The class «Direction» should store a value between 0 and 2π to indicate a direction. Besides static operations like the conversion of degree formats, it provided operations like turnLeft(double degrees) and turnRight(double degrees). The mathematical complexity discouraged the trainees and distracted them from learning Java. But I would do it again with a more suitable class such as «Temperature» or «Fraction».

8.5 Avoid Mathematics in Examples

Trainees outside technical universities usually have no mathematical knowledge. Even trainees with a fair mathematical background do not like to integrate mathematics in programming. There is no need for extended mathematics anyway. Joseph Bergin and other authors and trainers show that instead of using the utility class «Math» and mathematical operations one can operate on String and StringBuffer. Integer operations, as in Fraction, that students usually learn in their sixth school year seem to be alright.

8.6 Cognitive Load Theory and Test Classes

According to cognitive load theory, writing classes against a test class should be even better than fixing one problem within a large framework. The trainee starts a class from scratch. Therefore, there is no redundancy.

8.7 Conclusion

This approach is an interesting alternative to fixing one problem within a large framework. It asks for different key-skills as doing fixes in an existing project. It ask for skills such as understanding specifications and meet them, writing code from scratch, instead of scanning for a problem and fix it.

9 Using Microworlds for Training

Microworlds have been widely used for teaching programming procedural programming. Dietrich Boles (1999) adapted this earlier work for procedural programming with Java. Joseph Bergin (1997) to C++ and to object oriented programming with Java (Bergin, unpublished).

9.1 Short Overview of the Three Existing Worlds

Karel++, the robot, uses a microworld with robots that do tasks with beepers. They can walk, turn left, pick up beepers and lay down beepers. Walls make tasks more complicated. As its name implies, Karel++ uses a simple code with similarities to C++.

J-Karel, is an adaptation of Karel++ to Java. It uses pure Java code. This means the robots are coded as regular Java classes. The world looks the same as in Karel++ and the textbook is based on the textbook for Karel++. It has not been published officially yet. Trainees can use all the power of Java when sub classing the Basic Robot classes. They may experiment even with multithreading and solve problems with several robots running in several threads.

Written in German, there exists a Java-programmed world that uses a hamster instead of a robot and nuts instead of beepers, but the possible trainings remain the same. This Hamster Model explains only procedural programming. An extension is planned that teaches object-oriented programming. But this extension will not be finished before 2003.

9.2 There are Several Obvious Advantages of Microworlds Like Karel++ or the Hamster Model

1. The trainee can solve a given problem with a small amount of knowledge. For the first problems, the trainee has only to know two commands «pickUp()» and «move()».

2. There is not a big overhead of file handling, programming environment or DOS/Unix-command-line instructions. In Karel++, the user sets up a world or opens a saved world with a given problem. He edits the program code and starts the program. The program may be executed step-by-step or non-stop. Two windows are giving feedback about the robots, their state and about errors or completion. The user can combine several classes by the C++ keyword #include.

Basic example of Karel++ code (left) and microworld with problem to solve

```
/* Class and Method Declarations */

/* Begin Task */

task
{ /* Declarations */
  ur_Robot Karel (1, 2, East, 0);

  /* Statements */
  Karel.move();
  Karel.move();
  Karel.pickBeeper();
  Karel.move();
  Karel.turnLeft();
  Karel.move();
  Karel.move();
  Karel.putBeeper();
  Karel.move();
  Karel.turnOff();
}

/* End Task */

/* End Program */
```
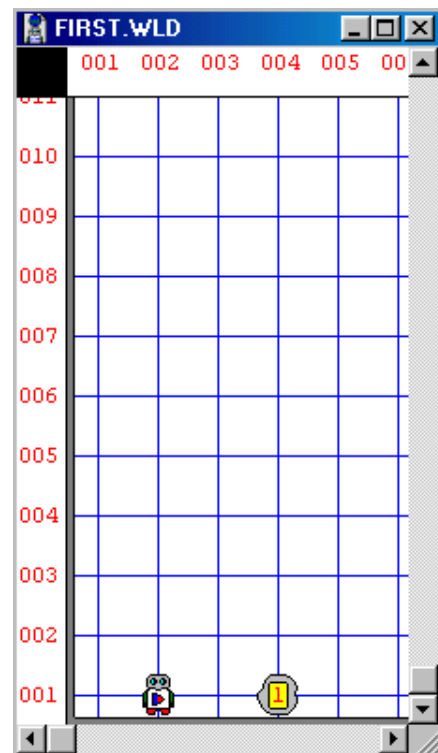

Fig. 10, Karel the robot in action

3. The microworld is a closed world, not an visible small part of a bigger problem. Therefore the trainee does not feel the need to see behind the curtains or around the corner.

4. The trainee gets sensible tasks to complete. With a given small amount of new knowledge, he is able to complete the task. He can even invent his own problems and solve them.

5. The trainee gets immediate feedback. He sees where his code runs successfully and where it fails.

6. The API of the microworld is very small. In the Hamster Model the trainee works in a predefined class «Hamster», in Karel++ the trainee subclasses one of two basic classes or one of his own classes. See the following figure.
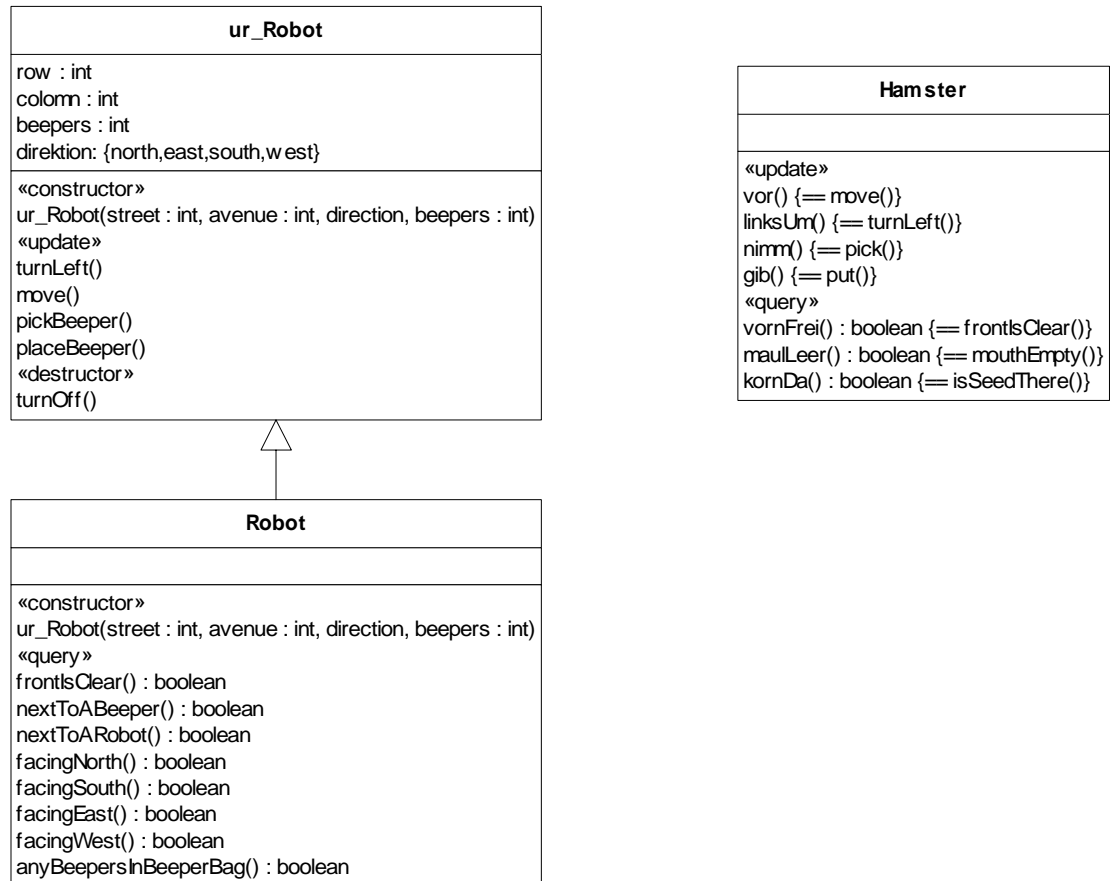
ur_Robot
row : int
colomn : int
beepers : int
direktion: {north,east,south,west}
«constructor»
ur_Robot(street : int, avenue : int, direction, beepers : int)
«update»
turnLeft()
move()
pickBeeper()
placeBeeper()
«destructor»
turnOff()

Hamster
«update»
vor() {== move()}
linksUm() {== turnLeft()}
nimm() {== pick()}
gib() {== put()}
«query»
vornFrei() : boolean {== frontIsClear()}
maulLeer() : boolean {== mouthEmpty()}
kornDa() : boolean {== isSeedThere()}

Robot
«constructor»
ur_Robot(street : int, avenue : int, direction, beepers : int)
«query»
frontIsClear() : boolean
nextToABeeper() : boolean
nextToARobot() : boolean
facingNorth() : boolean
facingSouth() : boolean
facingEast() : boolean
facingWest() : boolean
anyBeepersInBeeperBag() : boolean

Fig. 11, the Application Programming Interface (API) of Karel++ and the Hamster Model

7. The user can experiment with the microworld and learn there the basic knowledge of object-oriented programming. I will discuss this in this study on the following pages.

9.3   Teaching Polymorphism with the Microworld Karel++

Bergin (1997) shows in chapter 3.6 p. 35 how to override inherited operations with different methods and how to call the inherited method to use it within the new defined method's body.

In chapter 6.7 (Bergin 1997, p. 166-170) shows how to use one pointer to a Robot to address Robots with different behaviour. However the example Bergin uses is not convincing. Unlike for instance the Account example, it misses to demonstrate an obvious advantage of polymorphism.

9.4 Teaching Collaboration with The Microworld of Karel++

Bergin only devotes a small chapter (Begin 1997, 3.11 Robot teams, p. 55–58) to the topic of collaboration. He demonstrates with this small chapter, that you can introduce key concepts of object-oriented programming right at the beginning.

For me, the collaboration of objects is as crucial for the paradigm of object-oriented programming as polymorphism is. If you check for instance the design patterns in the famous book «Design Patterns» (Gamma E. and others, 1994) you will see that in most patterns collaboration of several objects and sub-typing goes hand in hand. So if there is a serious criticism about Karel++, it is the fact that it does not do more effort in teaching how to set up aggregations of Robots that work together.

9.5 User Feedback about Working with Microworlds

The feedback for the German version that covers only procedural programming is enthusiastic on www.amazon.de. The feedback for the English version is mixed on www.amazon.com. Both feedbacks are not significant (only 3–4 ratings). Obviously, the English version with Karel++ did not meet the reader's expectations completely.

Despite the German Hamster Model only covers procedural programming, it was highly estimated. The enthusiastic reader comments prove that the approach has been appreciated.

The English Robot microworld shows that you can teach also object-oriented programming with microworlds. Some shortcoming of the book Karel++ does not lay within its general approach. The book is not as well edited as the German book about the Hamster Model. Examples are lengthy. The visual organisation of the book is fair but not good or excellent. I read some comments criticizing that the book was too basic or almost too gentle as positive, not as negative feedback.

Microworlds are used at colleges and universities with success.

9.6 Design Flaws of the Microworlds

There lies a minor inherent design flaw in these microworlds. Usually, in object-oriented programming, you have either update or query methods. What is the difference between update and query? —Objects contain information. For instance the position of a robot in Karel++ and its direction are information contained in the robot. Position and direction are called attributes or fields of the robot. Either, you update this information by methods like move() or turnLeft() or you query this information by methods like isFacingNorth(). Query methods return a value, in Karel++ the Boolean values true and false.

In object-oriented programming, usually, you do not use update methods if you define new query methods. Bergin (1997), page 73, explains how to define new predicates (query methods). For defining an operation «rightIsClear()», he uses the update method «turnLeft()» in this method. Therefore after three turns to the left and checking «frontIsClear()» {== «rightIsClear()»} the robot is facing to the right. So he has to turn it again to the

left to avoid a side effect. A side effect is an unintended update of information contained in an object (an attribute) within a query method.

In a usual project, you would program a method «rightIsClear()» directly by querying the position and direction of the Robot and checking whether there is a wall or the borderline of the microworld to the relative right of the robot. Side effects are not covered in a good curriculum for object-oriented programming. But, implicitly, you teach, «Never ever use an update-method for a query method!» as the golden rule for avoiding side effects.

9.7 Solutions to this Design Flaw

Boles (1999) uses this effect to explain the concept of side effects, which is important in procedural programming. As Boles explains in Chapter 2 procedural programming before introducing object-oriented programming in the following chapters, which will not be available before year 2003, this work-around is adequate. Side effects are an important topic in procedural programming. In procedural programming, this problem cannot be addressed by the simple technique of not using update methods for a query.

Bergin (1997) uses this unintended effect to explain the fact that the return statement terminates immediately execution. And, therefore, you cannot put the last turnLeft() operation after the return statement. As side effects do not occur in object-oriented programming usually, Bergin (1997) does not discuss side effects.

9.8 Microworlds in the View of Cognitive Load Theory

By reducing the problem domain to a microworld, the intrinsic cognitive load is reduced dramatically. The trainee can focus on the algorithmic and design schema he has to learn. The immediate visual feedback discharges the brain from the task of relating unintended results to coding errors.

Karel++, with its C++ like code, adds some redundancy. The display of the executed code, not on the acting robot or hamster, but in a separate window does split attention.

9.9 Conclusion

By dramatically lowering the amount of intrinsic cognitive load, microworlds seem to be the only way for a quick introduction to the key ideas of object oriented programming. Still, considering the cognitive load theory, the design of existing microworlds needs improvements.

10 Programming in Pairs

Two persons working at the same computer (programming in pairs or pair programming) is used in the software industry to improve efficiency. It has a teaching, coaching effect that might be used for training, too.

10.1 Definition of Programming in Pairs in Software Industry

Programming in pairs is described in the web (see http://c2.com/cgi/wiki?ProgrammingInPairs):

«This was called "Programming In Pairs" in its first published description (by JimCoplien; see **Historical Note** below). It's more commonly called PairProgramming these days, notably by the ExtremeProgramming community.

Context: You have several people working on a project. Everyone's received the basic training necessary to do the job. Some are (inevitably) more or less experienced than others. You have PairProgrammingFacilities.

Forces: You want to get more done than your most productive person can do. You want your less experienced people to learn from your more experienced people. You want to structure your organization to avoid needing "n squared" communications channels.

Therefore: Pair up your people. When applicable, each pair should have a relatively experienced and a relatively inexperienced person. For work being done at a computer, put the relatively inexperienced person at the keyboard, so everything the experienced person says has to flow through the novice to the computer. The point is not for the guru to dictate to the greenhorn; on the contrary, putting the novice at the keyboard is meant to keep him or her more in the loop. When (as happens occasionally) that doesn't provide enough bandwidth, the experienced person will say, "Let me drive" (that's not a prescription, that's what's frequently said), and will show something instead of teaching it at a more leisurely pace. When one member of the pair is not following what's going on, he or she can ask to drive. »

10.2 Impact of Programming in Pairs

Collaborative Learning is studied well and shows in general a positive effect. A negative effect of collaborative learning is the fact that low achievers progressively become passive (Dillenbourg, 1996, p. 8–11). Therefore, it is a good idea that the less experienced should drive (sit at the keyboard).

Cognitive load theory has two explanations for the effectiveness of programming in pairs. Firstly, because of the assistance of a second trainee the cognitive load is reduced. Two trainees hold more information than one. So, together, they are able to fill gaps in the knowledge one trainee alone would suffer. There is less back and forth (attention split) between screen, textbook and online tutorial. Secondly, according to the modality effect, the aural explanations of the second student might expand working memory while studying the textual information on screen.

10.3 Practical Problem in Education

The main practical problem are the trainees themselves. As nowadays every student has a PC, usually, it is not easy to motivate him or her to work together at one PC.

Several precautions help to overcome this problem:

- Make teams where a better student works together with a weaker learner.
- But do not make the difference too big. Do not assign the best trainee to the weakest.
- Instruct them to use one PC for browsing manuals and the other for programming.
- In case of non native English speaking students, one of the trainees should know English well. This reduces the cognitive load added by the English language.

Experiences at our institution for adult education

When we managed to motivate students to program in pairs we found it very efficient. As a rule of thumb, the more we managed to get the students to program in pairs, the more they learned and the less we got students that did not learn at all. This experience gives some support to the cognitive load theory.

10.4 Conclusion

Programming in pairs is an efficient way to improve teaching without any additional effort. Team building causes some problems. The effectiveness of this approach gives some support to cognitive load theory. However, other learning theories predict this effectiveness, too.

11 Review Questions

Cognitive load theory implies to prefer offline teaching. Giving sound theoretical background and working with CRC-Cards and diagrams of the Unified Modelling Language was discussed in previous sections. Using review questions is the offline technique that is used regularly.

11.1 Review Questions

You may find a good example for check up questions in the book of Schildt (2001). Schildt inserted a so called One Minute Drill when ever he explained three units of information that he can ask for in a one line question and give a one sentence answer. He repeats this check up questions at the end of each module. With the One Minute Drills he avoids to overload the mind of his readers. With the module check up he forces his readers to a second repetition. The comments on amazon.com show that these checks ups are highly estimated.

11.2 Flash Cards

In a live educational setting the trainer could use flash cards or mind maps for repetitions of basic knowledge. Flashcards have a review question in front and the answer to it on its back. Cards should be small: about 3 times 7 cm, so that the question and the answer are not longer than one to three sentences. Often they hold only key words. The flashcards are provided by the teacher or written by the students themselves what needs training.

11.3 Mind maps

Mind maps hold only keywords. They are a visual representation of all the nodes of the technical terms and the knowledge of a topic.

There is special software for mind mapping. Students need instructions when they should use mind mapping for themselves.

11.4 Evaluation of Control Questions, Flashcards and Mind Maps

The knowledge training of this kind remains on a low cognitive level. The trainee only has to recall what he has learned. Nevertheless, it is important for the trainee to name things correctly, to recall code elements by heart and to remember the most important problems and advantages. This helps him to acquire the complex schema. Every name for a nod in the knowledge tree of object-oriented programming reinforces long-term memory and helps to reduce the cognitive load. As these techniques are integrated in a course or book, it would be bare speculation to say anything about their efficiency based on experience or reader feedback on amazon.com.

11.5 Conclusion

Review questions, flash cards or mind mapping should be used regularly in a course to help acquire complex schemata by reinforcing long-term memory.

12 Analysing Code and Check with Multiple Choice Questions

An offline training on a higher cognitive level than review questions are multiple choice questions where the trainee studies code examples.

12.1 Examples of Such Questions

The multiple choice questions used in the exams for a Sun-certified programmer or developer, may also be used in training. You may find such drill question for instance in the Book of Boon (2000). The repetition or check up looks like this: Firstly, the trainee gets a question and a code example of up to 15 lines as prompt. Then, he gets a question with a set of four answers. Among the answers there are always one ore two choices why the code should not compile. There may also be one choice that the code does not compile for any reason. An example looks like this (Boon, 2000, p. 76):

Specifying this line at the top of your source file

```
package awt;
```

a. results in a compile-time error because Java already defines an `awt` package.

b. specifies that all of your classes in this file should go to into Java's `awt` package.

c. specifies that all of your classes in this file should go into your own `awt` package.

d. imports all of the classes in your own `awt` package.

12.2 Cognitive Level and Cognitive Load

Such multiple-choice questions are training on a high cognitive level. The trainee has to apply his knowledge to a new situation. The extrinsic cognitive load is low, because attention is not splitted as much as it would be in an online training situation. Plus, in online training, several steps are needed to get feedback. Often, the trainee has to ask for support, to get an accurate feedback. For instance because the code above did not what he has expected it to do.

The effectiveness of this approach gives some support to cognitive load theory. But, especially to this approach, there are other theories that apply as well. This kind of multiple-choice questions is derived from behavioural psychology. It is known as «Discrimination Frame Sequence» (Espich, 1967, p. 38).

12.3 Similar Check Up Questions

In a live educational setting, you can implement this kind of multiple-choice trainings in various ways. Dörig and Waibel explain the group activities below in Dörig (1995). There are group activities that use discrimination. Multiple-choice questions can also be used for flash cards.

12.4 Group Activity Imperator

You can give short code examples to the trainees and ask them to decide whether it compiles or not, whether it creates the output you want or not. In a group of youngsters it is highly appreciated to decide yes or no by thumb up and down. The trainer or moderator asks the questions and after some seconds or minutes of reflection he asks for the decision. If trainees do not agree about right or wrong, the group discusses the solution.

12.5 Group Activity Black Sheep

Black sheep is similar to imperator. The stimulus is not one code example but different solutions. One of the solutions, the black sheep, is wrong. The trainees have to decide which one. As in imperator, in case of disagreement, the solution is discussed.

12.6 Individual Activity Flash Cards

Flashcards with multiple-choice questions are possible. The problem is that the trainee might remember the right answer by hard instead of recreating the answer by reflection. That means he remains on the low cognitive level of recall instead of applying basic knowledge to a new situation. The creation of flash cards by peers and exchanging this flashcard can overcome this problem.

12.7 Creating own Examples

The trainer can ask the trainees to invent their own examples for imperator, black sheep or write their own flashcards. Finding examples is on the same cognitive level than applying basic knowledge to new situations. See

the evaluation of **Metzger** e. a. (1993, p. 72) based on the taxonomy of Bloom (1956).

12.8 Conclusion

Cognitive load can be reduced by several offline activities. Some of them may apply to distant learning; some are designed for group activities. Building schemata with such activities is often more effective than online problem solving. At least there is less risk that new items of information supersede working memory.

13 Chapter Summary

May we relay on experience and estimations. Don't we need proof?

13.1 Estimations and Experience from the Analysis of Existing Material

A lot of experience on teaching Java supports cognitive load theory. There is no proof of the correctness of cognitive load theory. But, the cognitive load theory seems to be an appropriate model for the evaluation and design of highly interrelated topics like teaching object oriented programming with Java.

Books and classroom activities could be designed to be more effective. Planned books, scripts and classroom activities should be checked for cognitive load. Cognitive load theory may give us hints to improve teaching and to design improved educational material.

Among the most efficient approaches to reduce cognitive load there are:

- Microworlds
- Books that use a spiral approach
- Reinforcing knowledge acquisition by flash cards, mind maps and short review questions
- Learning to apply knowledge to code by checking code examples, either in a textbook on a flash card or in a group activity.
- Fixing and improving code while focusing on specific problems and small tasks.

13.2 Do We Not Need Proof?

I mentioned in the beginning the metaphor of a racing car where behaviourism, cognitivism and constructivism are like the engine that powers our car, but the cognitive load theory is like weight and aerodynamics that slow down our racing car. Often, we do not have the time and the money to prove that replacing some material by lighter material or making a smoother shape really does make our car faster. Nevertheless, it is important to think about such improvements permanently. We might get surprised sometimes. The whole system might not react as predicted. For instance, we might loose traction when we remove some weight at the wrong place.

So it is with cognitive load. For instance, Sweller (1990, p. 178) reports of worked examples without positive worked examples effect in geometry. The inefficiency of the worked example effect in this examples, as he

could prove in this later study (Sweller, 1990), was according to the split-attention effect.

13.3 Counting or Rating

Sweller (1988, p. 272) is counting the cognitive load for a mathematic task based on the number of (1) statements, (2) productions, (3) cycles to solution, (4) conditions. Because of the heavy cognitive load, I could not quite follow his calculations. Anyway, most of the time, in practice I would suggest to rate the amount of cognitive load. As teacher, we can make some estimation what our students know when they start a course, we can also figure out whether the amount of new or interrelated information is high. With some empathy we can also rate how much cognitive load our instructional design adds to the learning situation.

As Langer, e. a. (1974) point it out, often, rating is much more accurate than counting. The number of words in a sentence does not imply that this sentence is difficult to read. For instance, you can replace «to get around» by «circumvent» and save two words, but the sentence gets rather more complicated than simpler.

Therefore, rating the cognitive load, is an accurate mean in practice. If there are time and money, however, it would always be nice to check by testing whether our expectation are met. Most often, this is not possible to do, and most often, we will not fail with our rating neither.

Chapter 5: Existing microworlds for teaching Java

This chapter mentions the initial ideas behind microworlds, gives a short history about learning to program with microworlds, and evaluates the existing microworlds.

1 Motivation for Microworlds in the Field of Programming

1.1 A Look at the Preface of the Existing Worlds

The initial idea of microworlds is to provide students a familiar environmental. They should not be forced to think in bits and bytes, numbers and characters. There is already a large amount of new information, new concepts and new procedures to learn. The danger to loose sight of the essential in programming, that is problem solving, is high. Students get lost in syntactical and technical detail.

Boles (1999, page 3) adds to his initial thoughts very figuratively:

> «Der Kampf mit dem Compiler bekommt somit höhere Priorität als der Programmentwurf an sich und kann frühzeitig zur Frustration führen.»

> Translation: «The fight against the compiler gets higher priority than program design and, soon, may lead to frustration.»

Bergin (1997, p vii) writes in his preface similarly:

> However, the concepts of Karel are still as vibrant and valid an introduction to the programming and problem solving processes as they were when first introduced. … Here, the objects are robots that exist in a simple world. There can be one or several robots assigned to a task. The programming task is divided into two parts. The first part is defining the capabilities of the robots that are needed. The second is providing a description of the task for the robots to perform. The programmer uses his or her problem solving skills on both parts of this task.

1.2 Short History of Microworlds

Karel the robot was created in 1981 to teach Pascal (Bergin, 1997, p vii) it was adapted to C++ by Bergin in 1997. Bergin adapted it to Java too but did not officially publish it, but a version can be found on the web (Bergin, 2001a).

Boles refers to two roots for the Hamster Model: The microworlds based on LOGO (Ross, 1983) and Karel the robot. The later was first adapted to ELAN (Oppor, 1983, Klingen, 1983 and 1985), a procedural programming language, and later to Java. Boles uses Java to introduce procedural-programming concepts only. The planed extension to show object-oriented programming was postponed to 2003.

The idea behind LOGO is stated in Papert (1980). The enthusiastic expectation of Papert, to have LOGO as a touchable and loveable math-world, which would allow all children to enter his fascinating world of mathematics, were not met (Mendelsohn, 1990). Or, may be, do we not know about their long term effect? — But at least, Papert showed us that programming can be a children's play. However, LOGO is not just a toy, it is an even more powerful language as the procedural programming languages such as BASIC of those days. The idea of the loveable and touchable has been maintained in Karel++ and especially in the Hamster World. But it is used not for children, it is used for students.

1.3 WYSIWHa instead of WYSIWiG

Rajan (1990) refers to Jones (1984, p. 777), and reports that special problems are related to the flow of control. Missing knowledge about flow of control hinders students from writing programs and understanding programs.

Rajan (1990, p. 391), therefore, wants for programmers WYSIWHa: What you see is what happens, in analogy to the WYSIWiG (What you see is what you get) of end user applications. A microworld offers exactly this WYSIWHa. All what happens, happens in the microworld and can be seen.

1.4 Conclusion

The authors' main motivation is to introduce problem solving and program design early in the teaching process. A long tradition of LOGO programming with children and students education with worlds for Pascal and other procedural languages was successful to do so. Reducing cognitive load is part of this approach. Especially Boles is aware of the importance to reduce "the number of concepts that need attention at the same time."

2 Microworld, Problem Solving and Object Oriented Design

As problem solving and design is the main field of interest. We take a look at what microworlds may provide.

2.1 Software Development is a Problem Solving Process

Dietrich Boles (1999, chapter 3, «Programmentwicklung», p. 35) and Joseph Bergin (1997, chapter 3.8, «Tools for designing and writing robot programs») both are driven by the idea that the trainee should learn from the very beginning that software development is a problem solving process. He or she should learn not only bricks of Java but permanently experience programming as a process of problem solving. The trainee should learn the techniques to do it in the beginning and apply them throughout the entire course.

2.2 Scenarios Used to Teach Software Development

Joseph Bergin uses two basic problems to teach problem solving with object-oriented programming. One problem is to design a house by placing beepers. Variations are placing beepers to form digital digits, or placing beepers around walls.

The other problem is to harvest a field of beepers. To show the advantages of a good design, the problem is alternated. Robots should draw different type of houses by placing beepers, or the size of the beeper field is changed.

Fig. 12, drawing with beepers

Both problems can be done in several ways: 1. One Robot, one class, several methods, 2. One Robot, several classes 3. robot teams.

2.3 One Robot One Class Strategy

One robot in one operation could do the entire job. Bergin lets the students first do the whole harvest task in one method.

Afterwards, the job is analysed. Repetitive sections are separated and defined in methods. With this predefined methods, one robot does the job by executing a main method that calls sub methods. For alternations a set of different basic method should be set up. The smaller and easier to read the basic methods are, the less different methods are needed, the better the solution is. This is the procedural programming approach.

2.4 One Robot Different Classes Strategy

Variations in robots behaviour can be made by subclasses. In the harvest example different types of harvester could have all their own method to harvest a row. Some robot classes harvest a long row, some a short row. Only this method is different from one harvester to the other. The main task method is the same for all robot classes.

In the house-painting example, a basic roofer class that paints a standard roof may be sub classed by roofers that paint a flat roof or a steep roof.

2.5 Robot Teams, Collaboration Strategy

In the harvest example, we can have a team of robots harvesting the field, the more rows a field has, the more robots we need to do the job.

In the house-painting example: Instead of having one robot that makes walls, doors, windows and roofs, we use a team of different robots to do the job. This example is very figurative, because, as Bergin shows, we can have a carpenter, a roofer, a mason, and a contractor who coordinates the team.

64

2.6 Link to Object Oriented Design

Robot names and method names are crucial for writing readable code. The trainee sees the advantage of using talking names for classes and methods. For instance a RooferForFlatRoofs gets the instruction to «make3UnitsRoof()». Variations in behaviour of single robots are usually done by sub typing (polymorphism), not by providing more methods for one single type of robot. By setting up different robot teams for different houses one can see the power of aggregation. With robot teams (aggregation) one can combine different types of roofs, windows, walls and doors to construct a vide variation of houses.

2.7 Advantage of Microworlds

In the microworld you get all the power to solve a given simple problem in a variety of ways. There is no way outside microworlds to show different ways of problem solving from the very beginning. Without microworlds the trainees learn in the beginning the theory about how ingenious object oriented programming is. With microworlds he can use effectively object oriented programming techniques with very little training. King's (2000) excellent book needs 443 pages before getting to the topic of sub typing. And without microworlds this is the way one should go to effectively teach it well.

2.8 More Variations, More Training and Transfer

The teacher cans set-up new problems easily. In the harvest problem, for instance, using variations of beeper fields can show the need for sub typing. In the basic field, there is a beeper on every field. You can define fields of beepers with a beeper on every second cell (vertical and/or horizontally) or with two beepers on every second cell.

Fig. 13, variations of harviest problem

2.9 Conclusion

There is very little cognitive load, this way the student can study and use effectively key concepts of object-oriented programming like inheritance and collaboration after a very short time of training. He does think about object oriented design from the very beginning without experiencing that this should be something difficult or cumbersome.

3 Other Elements of Teaching

Teaching control structures is not a big problem in object-oriented programming. It can be taught also by traditional exercises easily. It is still good to know that Bergin (1997, p. 96 to 146) and Boles (105 to 150) show a lot of examples that are fun.

3.1 The If Statement Gives Robots a More Flexible Behaviour

The if statement makes the harvest easier: If there is a beeper harvest it! This way, robots can harvest fields where some beepers are missing.

Bergin (1997, p. 80-82) uses hurdles to introduce the If-Else statement.

If there is a hurdle, jump over it, else go ahead.

3.2 Save Operating

You can define methods that avoid robots crashing, because they run out of beepers to put, try to harvest a beeper that is not there, or run into a wall or another robot. This is a very good example of the if statement, because avoiding exceptions by checking whether the necessary conditions for an operation are met, is an important strategy to write nicely running applications.

3.3 While Statement

Good examples for the usage of while statements are fields of variable size (while field has not ended) and with an unknown number of beepers on one cell (while there are beepers). You can set a wall surrounding a range that has no beepers on some cells.

Fig. 14, mountains of variable height, or sinks to test while loop

Walking over a mountain of undefined height (while mountain still goes up) is also a good example for usage of the while statement.

3.4 For Statement

Instead of the For statement with its complicated syntax Bergin (1997, p. 101-103) uses the Loop statement. Boles (1999) avoids looping at all.

3.5 Recursion

Bergin and Boles both discuss recursion with good examples. As recursion does not occur in object-oriented programming, I regard this examples as obsolete. I do not recommend using them in a course about object-oriented programming.

3.6 Conclusion

As Bergin (1997) shows us, important concepts of object-oriented programming and object-oriented design can be teached efficiently with microworlds. With some improvements in the choice of exercises, we could do even better.

4 Disadvantages of Existing Worlds

There are some disadvantages of the existing worlds.

4.1 Disadvantage of The Hamster Model

The Hamster World is not object-oriented. As problem solving is rather different in procedural programming and the main idea of microworlds is teaching problem solving methods, the Hamster World is not suited for a short course in object-oriented programming.

The advantage of the Hamster World is its nice interface and the usage of pure Java code.

4.2 Karel++

This is probably the best tool for a first contact with object-oriented programming. The code is more C++-like as Java-like. This adds a little redundancy. This fact adds some extra training at the time, when the course leaves the microworld and real Java coding starts.

What I miss most in Karel++ are two things: attributes and arguments. Both elements are crucial for the collaboration of robots. As I regard collaboration as the most important aspect in object-oriented programming I miss this feature badly.

4.3 JKarel

The textbook to JKarel is completely based on Karel++. But, JKarel can much more. JKarel is written in pure Java and completely object oriented. The robots of JKarel are defined in regular Java-classes that implement the interface Direction. JKarel even allows multithreading.

The disadvantage of JKarel is exactly its strength. The user works in a Java-class and sees there some of this confusing code elements that I would rather like to hide from him. He sees for instance the static method «main()», or the import of the package «kareltherobot». The class must be

saved as «Main.java», execution starts in the method «main()» of this class. The class has to be compiled, etc.

4.4 Conclusion

All three worlds lack some important features: The Hamster Model is not object oriented, Karel++ has no attributes and no parameter passing, JKarel does not hide well enough the ugly object oriented overhead.

Starting with Karel++ and switching to JKarel seems to be an option. The draw back of this idea is that you are switching the learning environment twice: From Karel++ to JKarel, from JKarel to regular Java.

I decided to start a new world, because I had some additional ideas for improvements.

Chapter 6: Reducing cognitive load in Roboworld

As I decided to design a new microworld (Roboworld) to overcome disadvantages of the existing once, I considered also reducing cognitive load even more than the existing worlds do. Everything that distracts students from learning should be omitted while maintain compatibility to the programming language Java.

What this world should provide to lower cognitive load.

1   Hassle Free File Handling

Existing worlds use open and save actions for scenarios and for classes. The advantage of this is flexibility, the disadvantage is, that there is no list of scenarios. A list of scenarios lets users quickly switch scenarios and gives them an overview.

1.1   Advantages

A trainee might like to look up a scenario to examine classes he has written there. A list of scenarios helps him to find the scenario. Retrieving a file from the file system is more cumbersome.

Fig. 15, List of scenarios, will be listed alphabetically in the final version

Retrieving files might also add to cognitive load if the trainee is not absolutely familiar with file handling procedures.

Quickly browsing scenarios is not possible, this adds to the cognitive load. This is especially true when classes and scenarios are saved separately.

1.2 Overcoming the Disadvantage of this Approach

The disadvantage of my solution is loosing flexibility. You cannot import a single scenario. I plan to add this flexibility later by providing import and export functions for single classes and scenarios. Yet, for a beginner's course, this feature is not of high priority.

1.3 Expectation, Conclusion

A list of scenarios should not make a big difference. But, as the working memory is very limited, we should profit also from slight advantages. With only about 7 items remembered at the same time in working memory, any little improvement counts.

2 A Simple Parser/Compiler

Karel++ uses its own code and compiler. I like this feature, because, this lets you work with a simpler code. Also, using the Java compiler may add problems. For instance, the Hamster World is not compileable offline on a Macintosh computer.

2.1 Giving Remedial Information

The parser / compiler can give other information than the Java compiler. Information can address beginners' problems. It may allow to forgive more mistakes or to use simpler keywords, such as «loop» instead of «for». For trainees from foreign languages, the teacher might translate error massages and warnings.

Good remedial information cannot be completely implemented in the beginning. I plan to watch trainees closely during classroom training and check for the most common failures. These failures, I plan to catch at compile time and give trainees the most accurate remedial information.

In a later version, all remedial information should be kept in a text file that the program imports at start-up. This way, the remedial information can be translated into different languages.

Probably the best solution would be to have the remedial information spoken to the student. So he or she could listen to the information while fixing the code. This way we could profit form the modality effect.

3 A Panel to set up a Scenario (World With Beepers and Walls)

As in the existing worlds, the scenario with beepers (nuts) and walls is set up by clicking. Other than the Hamster Model, the scenario shows rows and columns. Other than in Karel++, rows and columns are not referred to by streets and avenues. It uses the spreadsheet metaphor.

3.1 The Spreadsheet Metaphor

The reason for the usage of the spreadsheet metaphor is that people outside The United States are not used to streets and avenues that build blocks. Therefore this metaphor might add cognitive load for trainees out-

side The United States. Spreadsheets are widely used and people are familiar with them. Spreadsheets refer to columns by letter. The number of columns a robot walks is difficult to count with letters. Therefore, I refer to columns by number not by letter. This way, students can add up or discount them.
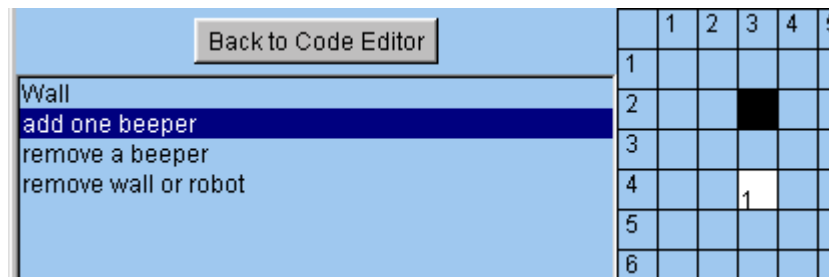

Fig. 16, Panel for changing the microworld

3.2 More Functionality

More functionality will be added later to this panel. Drawing Walls from point to point, removing all beepers, setting a predefined amount of beepers, setting an infinite number of beepers are all features that will be added later.

3.3 Better User Interface

The user interface should be improved as well. The solution, now, is quick and efficient. But I know, setting up a world with icons and by drag-and-drop would be state-of-the-art today. Also this solution, as it is, does not support the subject-predicate-logic of graphical user interfaces. I intend to make later a user interface where the options work like painting tools.

3.4 Conclusion

I did not invest much in the set-up functions of scenarios. As this is something simple, it should not do any harm.

4 A Panel to Manually Instantiate Robots and to Call Up their Methods

4.1 Motivation for this Feature

The existing microworlds use a main task that calls the other methods of the hamster or robot. Karel++ uses a static task, which instantiates one or several robots. In the Hamster Model, the robot has been instantiated already by clicking. An instance method «main()» is triggered when the user chooses to run his solution.

For me, both solutions seem not very appealing. The advantage of the solution of the Hamster Model is its simplicity. But, it needs no further discussion. It has nothing in common with object oriented programming. The solution of Karel++ gives the trainee the wrong impression that there must be a main task. But, the modular thinking of object-oriented programming should teach that there is no main task. Plus, it needs fur-

ther manipulations when the students like to test a single method they have written.

4.2 Solution Part One: an Instantiation Tool

Fig. 17, the instantiation tool

The figure above shows the instantiation tool. The user may choose from any of the robot types he has defined in a class or he may use the basic API-class UrRobot. With the four arguments he may specify column, row, direction and number of beepers that the robot is carrying.

The graphical display of the instantiation tool follows the syntax of Java. It does not follow the subject-predicate-logic. This is done on purpose.

4.3 Solution Part Two: Better Trigger Different Operations Manually than Running Just One Main Task to Reinforce the Paradigm of Object-Oriented Programming

According to the paradigm of object-oriented programming, robots should be built to do several tasks and these tasks are more or less equivalent, usually. So, the user should be able to trigger different tasks manually. To build a batch of all programmed operations to complete—a one and only— major task should be the exception not the rule. A one and only main task supports procedural thinking and this is just what we want to avoid.

4.4 The Possibility to Trigger any Operation Reduces Cognitive Load

Procedural thinking means to do your planning from the whole (the top) to the details (the bottom). This exactly produces a lot of cognitive load for the same reason as the goal free effect does. The novice should look for solutions bottom up: finding methods that do some of the job; experimenting with them; testing them, and improving them. It is crucial that he can call up all this methods manually.

Also, this bottom-up idea, reflects the idea of Papert (1980). Children (or students) should experiment. Build up their knowledge and experience driven from their own desire. Playing around affects students and children much more than solving a problem.

4.5 Solution Part Three: The Macro-Like Learn-Mode

Fig. 18, robot control centre with learn mode

To help students even more, I added a macro like learn mode. In this mode, every method that has been called up will be written into the code of the class that is currently open for editing.

4.6 Why Instantiating a Robot Manually?

Firstly, instantiating manually let you avoid the new statement in the beginning. The user can experiment with robots without knowing how to set up the code for instantiation. But, he can see what the command «new» does: it constructs a new robot. It places the robot at the place and in the direction specified in the constructor.

Later, when one robot works together with others, the student has not only to know how to construct a robot with a command line, he also has to know how robots remember with whom they are working and how to forward commands to their colleagues. Therefore, it is preferable that the trainee already knows the schema of instantiation when he gets to this highly interrelated topic.

4.7 Conclusion

Instantiating a robot manually teaches a fundamental schema early, easily, and figuratively and, what is the most important, separately from any other topic.

5 Visualise the Exchange of Messages Without Splitted Attention

Method calls—or the exchange of massages—are a basic concept of both structured programming and object oriented programming. There are two means to show the exchange of messages without split attention. This can be done in a textbook or directly on the microworld.

5.1 Visualization in a Textbook

Fig. 19, King (2000, p. 96)

The figure above shows a good example to visualise method call and return value in a textbook. According to split attention effect, the explanation would be better close to the arrows and not in a separate text.

5.2 Visualization in Existing Worlds

The Hamster Model gives a one-line feedback about execution at the bottom of the window.

The visualization of Karel++ is more elaborated. It uses the way debuggers show code execution.

Fig. 20, Hamster Model

Fig. 21, Feedback about execution in Karel++

Also feedback about execution is more sophisticated in Karel++, it works with splitted attention. The attention goes back and forth between Karel in the right window and the sequential execution shown in the window to the left. The window to the left with the code jumps up and down to display always the current line of code, which will be executed next.

5.3 Improvement

As the four graphics show, execution is displayed as messages between robots. Calling is visualised by red colour, returning by black colour. The calling robot turns green to show that it is still running. The inactive robot stays black. There is no split attention and no inadequate usage of colours.

Fig. 22, Sender and Listener exchanging messages

This way, the most information, about the exchange of messages, is where it ought to be, on the graphic. The trainee should be able to see every single step in the process. A method call is not only the passing of control, it is the passing of values, too. The return of the method is not only the passing of control back, it may be the passing of the return value. This return value may be used directly, or, it may be assigned to a variable. All this single steps should be visualised.

5.4 Giving More Comprehensive Feedback about the Sequential Execution

There is even more need for information about the sequential execution process.

Fig. 23, Sequential feedback about execution

This feedback is given in the panel at the bottom right. The user can compare execution of the method and the definition of the method in its class. Well, to have three panels showing three sights of the same, one sight as class definition; one as sequential execution and one as robots in actions are not accurate according to redundancy effect. This might be a field for further improvements. For instance, I consider displaying the sequential execution as a pop-up text when clicking on a method or on a robot.

5.5 Acknowledgement

This idea is based on Joseph Bergins, «Objects in Action» (Bergin, 2000 b). Bergin writes:

> «It is useful to think of an executing object oriented program as a graph or web of objects (nodes) and communication paths (arcs). An object communicates with another by sending it a message. This message is a request for service. The sender is the client and the receiver is the server. The receiver will fulfil the service request and perhaps return information to the sender. Often the server must rely on other objects to help it fulfils the service request. It then itself acts as a client to these other objects which act as servers for it. Control returns to the message sender even if no information is returned.»

Bergin displays (2001 b) an animated GIF that gave me the initial idea for the visualization I used in Roboworld.

5.6 Conclusion

Despite the possibility of further improvements, I believe that the graphical and textual feedback helps to acquire schemata of collaborating objects. Manual instantiation of robots, learn mode and the possibility to play and experiment goal free around with robots might reduce cognitive load. Most important, as I believe, is the graphing of messages between robots. There, the most important happens, therein, I feel, lies the heaviest cognitive load.

6 Define Attributes that Reference Robots and the Instantiation of Robots within Program Code

The association between objects is done by attributes. This basic concept should be implemented in the microworld, too.

6.1 Existing Worlds

Existing worlds are not object oriented (Hamster World), use only local variables or four predefined attributes (Karel++), or are written in Java classes (JKarel).

6.2 Motivation

As analysed in previous chapter, attributes (also called fields or instance variables) are so important to organize collaborations, plus, are extremely interrelated to other topics, they should be available in microworlds.

To reduce cognitive load, I use only attributes. At the moment, it is only possible to reference robots. But, the idea is, that it should become possible to pass robot references by messages and store them in attributes. This way a lot of design concepts and execution schemata could be learned in the microworld. By displaying the message with the value sent from robot to robot, the trainee does not need to imagine this value, as it is the case outside the microworld. Object references are visualised as serial numbers of the object. Therefore, it is easy to see for the trainee, that the receiver of the message now knows the reference number sent to it. He only needs to assign it to an attribute.

The example, with the sender and the two receivers, might give a first glance on the powerful tool Roboworld will become when it not only can assign robot references to an attribute to refer to other robots, but also can pass object references from one robot to the other.

6.3 Conclusion

Working with instance variables (attributes) and only them causes a minimum of cognitive load and enables a maximum of schema to be learnt.

7 Defining Classes

To lower cognitive load, there is a template class to define new types of robots (objects). The trainee can use this template, use the learn (macro) feature to add operation calls. So, with almost no training, he writes his own classes that do interesting things. This way he or she can directly experience what a class does and what it is for. The keywords are limited to «class», «extends», and «void» at the beginning. Besides this key words the user learns how a simple class head and body looks like, how a simple method head and body looks like and how a call to a basic operation looks like. He or she can soon call from one of his operations a self defined operation. Thanks to the template method, there is now need to learn anything by hard. Everything is obvious, concrete and understandable. What a difference to the hello world application that shows odd things, such as static methods, access modifiers, array definitions and with System.out.println() a reference to the standard output stream via a static variable of the predefined class System.

I implemented an all-in-one-window solution. The usage of multiple windows as in the Hamster Model or in Karel++ splits attention unnecessarily.

Of course, further improvements are possible, for instance, the Hamster Model uses colour coding for keywords, methods and classes, a pop-up menu for keywords would be an idea that I could implement easily.

To conclude: Already the existing microworlds omitted a lot of cognitive load. Implementing a learn mode and a all-in-one-window solution, I could diminish the cognitive load even more.

8   Call a Overridden Methods in the Super Class

This is crucial in the microworld. Otherwise it is not possible to show that different types of objects may move differently for instance. Inheritance, according to my experience, does not provoke a heavy cognitive load any way.

As overriding methods is already absolutely convincing in Karel++, almost no improvements must be made.

I only made a minor improvement, because other than Karel++, Roboworld does not display the executed code segment. Therefore, I set information about the type of robot a method belongs to into the sequential feedback. For instance, the feedback line could look like this:

-> Listener@ #15: c 3, r 4, dir 2 carries 0.turnLeft() as UrRobot

«Listener@ #15» is the class and the serial number of the robot.

« c 3, r 4, dir 2 carries 0» is the state of the robot.

« turnLeft() as UrRobot» not only tells the method, but also in which class the method is defined. «turnLeft()» in this case is inherited from class UrRobot.

To conclude, generally, there is no evident need to reduce cognitive load while teaching this topic. But, displaying the class that a method belongs to, at runtime, might reduce the cognitive load in complex situations.

9   Control Statements (If, While, Loop)

Here two questions arise: firstly, what features shall we support, and secondly how shall we deal with a fixed number of repetitions (loops)?

9.1   Existing Worlds

There are two solutions: Karel++ that only supports if, while and uses loop instead of the for statement and JKarel and the Hamster Model that are written in pure Java.

9.2   Solution in the Spirit of Cognitive Load Awareness

The solution of Karel++, using only three conditional branching control structures is the choice. Avoiding the Switch-statement is evident. Once basic are learned, there will be an example, where the teacher can introduce the Switch-statement as an elegant way to use a clear and easy code. There is no need to introduce Switch early. It even does not introduce any schema. One can easily substitute the Switch-statement by nested If-statements.

Also the For-statement of Java is always intelligibly replaceable by a While-statement. The For-statement in its full power is redundant. But, looping a fixed number of times needs a quite complicated coding. Therefore, introducing a Loop-statement with a very simple code sounds sensi-

ble. It introduces a schema (executing a fixed number of loops) with a very simple code. So, even as there is no Loop-statement outside of microworlds, there is a loop-schema that is used rather frequently.

10 Feedback in Case of Never Ending Recursion

10.1 Motivation

In the chapter were I analysed differences between object-oriented programming and procedural programming, I stated that recursive procedure call is not a paradigm of object-oriented programming. I also explained that extensive usage of control structures is a clear sign of the absence of object-oriented programming. But, never ending loops may happen in object-oriented programming, as they occur in procedural programming.

Adding information about how many loops already have been executed can give the trainee a feedback about such errors.

10.2 Solution

In Roboworld, therefore, I give a feedback about the number of loops that looks as follows:

>   Loop 366 enter true

In the case of the sender that waits for the user to add beepers the never ending loop was made on purpose. But trainees could run into such problems by mistake. Gently giving feedback about errors does lower the cognitive load, too, because the trainee does not have to keep this special error in mind when searching programming errors. He perceives the error easily.

11 Lower Cognitive Load by Omitting Detail

Much detail need not be teached in the beginning.

11.1 Variables and Operations That Are Not Essential for Object Oriented Programming

Local variables, class variables and class operations are not needed to understand object oriented programming. Also, other primitive types than Boolean and Integer, are not essential at all.

11.2 Access Modifiers

Access modifiers are not needed because there is no access to attributes anyway and there is no evident demand for private methods. We can ban these features from microworlds. Other worlds provide more features. I suggest switching to JKarel as soon as there is need for access modifiers.

11.3 Operators Are Not Essential

Operators as *, + , -, / are not implemented. Even without such operators one can do mathematical tasks like converting a binary number set by beepers, for instance beeper, beeper, no beeper, beeper in the decimal number 13 $(8 + 4 + 0 + 1)$ if there is some need to do mathematical exer-

cises. Another interesting but very demanding mathematical exercise would be to operate an abacus with beepers. You can substitute Boolean operations by nested If statements.

11.4 Type Safety

Type safety demands quite an overhead of knowledge: How to define an interface, declare implementation, do the implementation, use the Instanceof operator, or, understanding the error messages in case of type incompatibility. Type safety is not needed, because the trainee can realise types rather than defining them explicitly as interfaces and implement them. In case of calling an operation that a robot does not realise, he sees exactly when and why his or her application crashes. So the goal is rather to experience the need for type safety than to work with the tools (interface, implementation) of type safety.

To conclude, the teacher can introduce the schemata related to types. For instance, in the example of one sender and two listeners, we could have two different classes of listener that both implement different methods of «beeperPicked()». We can teach, that in this case, both robots have the same type Listener, even if they are not of class listener.

11.5 Arrays

Arrays are not implemented. As an array is nothing but a robot with a fixed number of links, you can define your own arrays. Just define a new robot class and add the number of attributes you like. Provide a method to preset with which of the robots in the array you work with currently. Override all operations of the robot type you like to realise in this array. All overridden operations just call the operation with the same name of the current robot. This way you have an array just like the once used by Java, only that you do not use the special array tools but you use your own methods to provide array functionality.

11.6 Garbage Collection

The removal of robots that one does not need any more is done by call of an operation. Of course it is more work to call an operation to remove a robot than to just setting the reference to it to another robot or to null. But the process of finalization and garbage collection in Java is quite difficult to explain and understand. Therefore, I preferred manual clean up in my microworld. The user gets to know the problem of clean up and he experiences an understandable way to solve it. Based on this experience, the trainee can learn the more sophisticated method that Java uses later easily.

11.7 Conclusion

A big strength of microworlds is their minimalism. And this should not be changed to avoid bringing back in cognitive load.

12 Does Roboworld Solve Novices' Problems?

In Chapter 3: Key problems of teaching object-oriented programming, I pin pointed to some major problems of novices' with Java. Does Roboworld help us to overcome these problems?

12.1 Understanding Variables in Java

Variables are very different in object-oriented programming. Most important are instance variables. They may hold values, such as row, column, direction. This values are updated and queried by methods. For these update and query operations I inherited the convincing solution of Karel++. What Karel++ lacks are instance attributes (called fields in Java) that hold references to other robots. This very important feature to really work with collaborating robots has been added.

Conclusion: Students can study the two major types of variables (values and references to objects) as instance variables (fields). This way, I hope having set a secure base for understanding variables in Java.

12.2 Understanding Pointers or Object References

In Roboworld students choose robots by selecting them from the list of instantiated robots. In this list, robots are not referenced by name but by type and number. The syntactical representation of type and number is about the same as by Java's default «toString()»-method. This way references should become all natural for students. The will be taught that a field (instance variable) of a robot holds that reference number. This number may be used for UML-diagrams as well, instead of variable names to support this concept even more.

Conclusion: Object reference should become easier to learn by making them visible.

12.3 Understanding Argument Passing with Messages

At the moment, only passing a Boolean variable as return value is implemented. The implementation of passing one argument forth a back of any type will be added.

Parameter passing happens visibly with drawn messages between robots.

Already the visible exchange of messages will help to understand argument passing better. Student get an impression of request and answer between client and server.

Conclusion: For me, this is a important feature of Roboworld. It has first priority when extending the features of the existing version.

12.4 Grasping the Idea of Object-Orientated Programming

I stated and motivated that three things are important to grasp the idea of object-oriented programming:

1. Not falling back into the paradigm of procedural programming's overall control and therefore abusing static attributes and static methods.
2. Collaboration and encapsulation.

3. Inheritance and Polymorphism

(1) Overcoming overall control: The instantiation tool, the possibility to trigger any operation manually and the absence of a main-task-method help to overcome this problem. I suggest to let the students play around with Roboworld the way Papert describes how children use LOGO's turtle. This way students might get away from the idea of overall control even more.

(2) Collaboration and encapsulation. Encapsulation was already demonstrated in Karel++, because there is, after instantiation, no direct access to the robot's attributes. Collaboration is improved by two means: Firstly, robots visibly exchange messages. Secondly, the associations between robots are maintained between to manual calls to the robots operation by attributes.

(3) Inheritance and polymorphism are implemented. But, there is no type-safety, missing operations will provoke a runtime error. Students shall get the punishment of type-unsafety, so they will appreciate Java's type-safety when they will learn about it.

Conclusion: There are solutions for this three central problems in Roboworld. Most of them are not solved as well in Karel++, and other worlds are to complicated for beginners.

Chapter 7: Explaining basic concepts with Roboworld

The power of the microworld is its graphical user interface. It lets users interact with abstract concepts in a familiar context. Everything is there in the microworld so you can name the things and explain them figuratively.

1   Primitive Data Types, Data Input and Output and Storage

1.1   Data

Beepers are the basic kind of data in the microworld. Beepers are of type positive-integer-without-zero. This data type is the traditional data type and was in Europe the only data type used until the 15th century. Therefore, for instance, accounting uses only this data type. In mathematics, usually, students study this type at first. There, numbers of this type are called the natural numbers. We can conclude that this data type is the most evident and familiar data type.

We can also represent binary data with beepers. If we use only one beeper for each field in an area, we can represent binary data.

1.2   Data Storage

Beepers can be handled. Beepers are the basic data taken from external storage and put into internal storage of the robot. With walls we may set limitations in size for external data storage. The visual representation with blocks of row and arrows represents the nature of computer data in general. External data storage consists of multiple series of sequential data. We can even make a representation of only sequentially accessible data by limiting access to data using walls.

Fig. 24, sequentially accessible data

1.3   Data Output

Joseph Bergin (1997, p. 58) used the robots to do paintings with beepers. I think this is a excellent idea, because it represents data output. To enhance the possibilities to visualise aspects of data output, I added the feature of setting colours with beepers. I use two colour schemes:

The first uses 1 to 8 beepers. The first colour means white, the second black, the next six beepers paint the colour circle (red—yellow—green—

blue—magenta). With this colour scheme students may be asked to do colour paintings or even write primitive drawing applications. In such an application, two robots may work together: the first is responsible for a figure, the second for the colour.

The second colour coding starts at the number of nine beepers. Setting 9 beepers means black (no light at all). Adding one beeper means adding blue, you return from complete blue to no blue, adding 6 beepers means adding green, you return from complete green to now green; adding 36 beepers means adding red. May be you recognize this as the 216 web colours, coded with beepers.

This way you may visualise that different values can mean different output depending on the system environment. So we have different values stored and different values triggering output to screen.

1.4   Data Input, Event Driven Applications

The user can add and remove beepers manually while the application is running. So he can even give input while the robot is moving. This way, he can communicate with the robot by inputting raw data (as beepers). You can set up scenarios where the robots work event driven. Just let the robots loop for ever, waiting for events, such as adding and removing beepers manually. Setting a beeper is like clicking. Setting a wall is like pressing a modifier key.

In the example of sender-listener (Fig. 22, Sender and Listener exchanging messages, p. 76) the user can add a beeper to the field where the sender is. The sender checks for the beeper, removes it and sends a message to the listener. This example gives students not only a simple example of the sender-listener concept used in Java (Swing or AWT), but also a concrete idea how this sender-listener concept might be implemented in the Java runtime environment (JRE).

2   Objects (Robots), Operations, Methods, Classes

Robots stand for objects. Is this a accurate metaphor?

2.1   Robots Are a Perfect Metaphor for Objects

Robots are programmable in the real world and in the microworld. Therefore, students are familiar with the idea of writing programs for a type of robot. This means a class, which is the written program for a type of robots, seems to be an all natural idea.

There are different types of objects as there are different types of robots. Also the behaviour of objects are defined in classes as the behaviour of robots is defined with them. Robots and objects both store data internally. Data that is lost when the robot or object is discarded.

Objects and robots both exchange messages, i. e. call operations of other objects (robots). When a robot (object) receives a message (a call to one of its operations), it performs the method it has been trained (programmed).

2.2 The Idea of Robots May Help Understand Objects Even Better

In real Java applications, you can think of robot teams working on data, views and controls. The robot's name (identifier) implies for which object or tasks the robot takes responsibility. When we say that a object is doing something, then we can think of a robot to be in action that is responsible for this object. For instance, a robot of class TextField is responsible for the TextField. When he gets the instruction setText("Hello World!"), he changes the text displayed in the TextField for which he is responsible. The idea of robots being in action instead of objects being in action might overcome the natural blockade of students when they have to do with unanimated objects. I mean objects that students do not associate at all with any activity. I remember in a course where we were modelling a washing machine that students did not like to award any responsibility to the laundry. Objects are a data model of the reality that we are modelling. As a result, we can imagine that there is for any object a robot in action. This robot manages the data of its object. Later we may merge the idea of the robot and its object to one idea, the idea of the object in object-oriented programming.

2.3 Attributes, Association, Aggregation, Collaboration

Information is stored in robot's attributes. These attributes row, column, direction, number of beepers are visible from the very beginning. With almost no prerequisites, after some basic instructions how to operate in the microworld, the trainee can see how operation calls change the information in the attributes. He also sees, that the call to update operation is the only way to change the objects attributes. There is now way to change objects attributes directly.

One important information that might be saved in an attribute is the link to other robots if there are robot teams in action. Other than in the existing world (Karel++, Hamster Model) attributes link to other robots. So one can show the important fact how associations are handled after a short time of instruction.

2.4 Instantiation of Robots

You can get a new copy (instance) of a robot, by setting its initial values and call for a new robot. This instantiation can be done manually. So the trainees sees first what «new» means and that an instance is nothing else than a new copy of an object. He or she sees the serial number and the class of the robot when he or she chooses the robot from the list of robots. This information is also visible in the sequential output that he or she sees in the pane at the lower right of the window. All this helps the student to get a representation of robots (and objects) as instance of a class.

2.5 Inheritance, Overriding, Polymorphism

Even the first robot inherits from «UrRobot» the basic attributes and methods. With little information (key word «super») the trainee can alter (override) the inherited methods, for instance to define a robot, «DoubleStepper» or «MileWalker», that makes two or more steps when it gets

the instruction to move. The microworld shows in the feedback area to the lower right of the screen from what class of robot the applied method is taken. Inheritance, this way, becomes an all familiar idea.

2.6 Conclusion

Robots are the most accurate visible representation of objects in object-oriented programming. They are the most powerful metaphor for them that we can think of.

3 Application Programming Interface (API)

It is always difficult to get an idea what an application programming interface (API) is for. In a microworld, this is easy to see. If there would not be the class UrRobot, the basic methods for manipulating the robots would be missing. We need this basic operations to do anything in the world. This is the smallest and the most figurative example of an application programming interface that I can think of.

4 Conclusion

A microworld of robots is a most accurate representation not only of objects in object-oriented programming but also of other basic concepts such as data storage, data input and output, sequentially and randomly accessible data, or the application programming interface. As accurate metaphors support learning (Iding, 1997), this might be an interesting side effect of Roboworld.

Chapter 8: Supporting the Unified Modelling Language

Do we need the diagrams of the Unified Modelling Language (UML), or do these diagrams add unnecessary redundancy? If we need them, how does Roboworld supports learning the diagrams of the Unified Modelling Language.

1 Is the Unified Modelling Language (UML) a Subject for a Java Short Course?

1.1 Unified Modelling Language (UML) in the Software Industry

The UML-diagrams are widely used in the software industry. They are used during the process of analysis. The developer creates a model of the problem domain described by the problem domain expert. This model is written with UML-diagrams.

There are design patterns that are used in any large project. The developer will consider them already, while he is modelling the problem domain. Ignoring them results in software that is almost not manageable. Grady Booch writes in the foreword to Gamma (1994, p. xiii):

«All well-structured object-oriented architectures are full of patterns. Indeed, one of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects. Focusing on such a mechanism during a system's development can yield an architecture that is smaller, simpler, and far more understandable than if these patterns are ignored.»

The missing knowledge of design pattern is reported by Fowler (2000, chapter 2.6, see text in box) as one of the main problems in project teams.

Hunt (1998, p. 16) explains some of the draw backs of poor object oriented design; he explains especially the problems of inheritance. Inheritance can reduce comprehensibly of code extremely. To revise the sequential execution of code, the developer has to look up methods in different classes. The cognitive load of this job is extremely high. If the design is poor the only way to solve problems of this kind is refactoring.

1.2 Unified Modelling Language (UML) as a Mean to Understand the Application Programming Interface (API)

Without understanding design patterns, the student will have difficulties to understand large parts of the application programming interface. The Java API uses often aggregation. Students who know the underlying concept of aggregation will have less problem to use these parts of the API. They

recognize the pattern and can make use of its schema. For instance, knowing the schema «chain of responsibility» that is used for input and output streams with Java reduces dramatically the cognitive load when learning about streams. If I can tell a student that input and output uses chain of responsibility, he or she will have much less problem to use the API classes to set up input and output procedures. There are only four items of knowledge needed besides this schema to do the job: Difference between «Stream» and «Reader/Writer», difference between source- and object stream, difference between input and output, and that you usually set up a chain of responsibility of one source stream and one ore more object streams. Not knowing the design pattern might be what overloads the working memory and hinder learning.

1.3 Is the Visual Representation Needed?

Theoretically, design patterns can be learned without visual representation. The visual representation of concepts and schema is not some aid that is self explaining. It is something we must learn if we will use it. Duval (1995) made extensive studies about this fact.

The Unified Modelling Language (UML) does not use a lot of semantic elements, but other system to visualise complex facts use them neither. What UML makes difficult is the syntactical variation of its signs in rather large number of diagrams. Diagrams, that are similar in what they show, but largely different in the point of view they are representing it. For instance, an objects diagram, a sequence diagram and a collaboration diagram show all objects in collaboration: The objects diagram shows that objects are collaborating, but not when and how; the sequence diagram emphasises the sequential execution of the collaboration, where the vertical axis is the time axis; the collaboration diagram displays the same, but uses sequential numbers for the messages sent.

A small but nasty problem is that for some details (type, identifier) UML deviates from the syntax of Java. For instance, «ballance : int» in UML is written «int ballance» in Java syntax. And it uses other names for instance attributes are called fields in Java.

1.4 Semiotic Research on Mathematical Sign Systems

From the semiotic point of view, the Unified Modelling Language is a sign system. Following the argumentation of Duval (1995) the results from the extensive research for sign systems in algebra and geometry must be transferable to the Unified Modelling Language.

Duval (1995, p. 75) explains the crucial points based on studies of mathematics: (1) Even now the teacher uses different sign systems (registre de représentation), students will not be able to coordinate them. (2) Hence, most of the time, learning will take place only in one sign system. (3) Most important: This understanding in only one sign system (compréhension monoregistre) is a major handicap, as soon as students leave the context in which learning took place, most of them will be not able to use their knowledge, however, they actually know it.

Duval concludes (1995, p. 76) :

«Plus généralement une compréhension monoregistre est une compréhension qui ne permet aucun transfert. Seul une compréhension intégrative, c'est-à-dire une compréhension fondée sur une coordination de registres donne ces possibilités de transfert.»

Translation: An understanding based only on one sign system is an understanding that allows no transfer at all. Only an integrative understanding, this means an understanding based on the coordination of sign systems makes transfer possible.

This conclusions is also supported by the review of Iding (1997, p. 249) for the usage of analogies to understand technical writings. See About Effectiveness of Metaphors, p. 33)

To conclude: the Unified Modelling Language, as I believe, is inevitable for two reasons. Firstly, it is used for communication: in books, among developers, in classroom, for documentation (see for instance Beck, 2000). Secondly, it is needed by the student himself as soon as he or she starts to design and implement more complex structures. There will be almost no chance to transfer the knowledge to this new situations if students do not have an integrated understanding of the paradigm of object-oriented programming in different sign systems and are able to switch from the one to the other.

1.5 Conclusion

Students can use the Unified Modelling Language for communication, for personal notes, and to build an integrated understanding of object-oriented programming in different sign systems. As design patterns help to reduce cognitive load dramatically when studying the API, for instance the parts to set up the graphical user interface with Swing or AWT, or input and output streams, I strongly recommend to start learning the most common design pattern and their visual representation with Unified Modelling Language before learning these parts of the API.

2 Learning Design Patterns and Unified Modelling Language with Roboworld

2.1 Motivation

As things are obvious and simple the teacher can introduce the diagrams of the Unified Modelling Language (UML) easily. This way one can build in the trainee's mind a visual representation of the patterns used in the solutions set-up in the microworld. The trainee will relay on this system later, when designing complex and abstract applications or while reading design or analyse patterns. It is quite difficult to build such a representation of patterns for classes and objects, when there are no simple and understandable examples. Examples, that are close to programming problems.

2.2 Design Patterns

You can use and discuss design patterns from the very beginning. For instance for the roofing we may use the pattern «template methods». I would strongly recommend to use UML diagrams and name the design patterns used in the process of problem solving. I would also suggest to think about scenarios where different types of design patterns could actually be used.

This way the visual representation with UML can be learned step-by-step. Starting with a simple class box, to show the robots attributes and basic operations. Afterwards, we can add the symbol for inheritance and for associations as soon as we need them. When we do this to summarise what we have done to solve a programming problem, we reinforce the awareness for strategies we used, and introduce the UML-diagrams needed at the same time.

2.3 Interaction Diagrams

Use the visual output in the scenario to draw collaboration diagrams. As in a collaboration diagram, we can place the symbols for the objects where we like, we can draw a collaboration diagram that represents actually a one-to-one static picture of the dynamic interaction.

Fig. 25, sender-listener as collaboration diagram

Later you may motivate the trainees to design collaborations or sequences in advance and see whether the robots meet their expectations.

This way, the students learn collaboration diagrams easily. Drawing the diagram will enhance their understanding of the interaction. They can

learn and train—based on easy examples—dynamic UML diagrams. The big advantage of collaboration diagrams over sequence diagrams is that they do not visualise a program flow. So collaboration diagrams emphasise exchange of messages and collaboration of objects rather than programming sequences. Therefore, from the didactical point of view, I prefer collaboration diagrams over sequence diagrams.

2.4   Sequence Diagrams

Sequence diagrams are widely used. For instance Gamma, e. a. (1994) used them. Therefore, the knowledge of sequence diagrams is important, however, they seem to be redundant, because they show nothing else than the collaboration diagrams do.

When students have learned to draw collaboration diagrams, we may start teaching the sequence diagram, too, as an alternative to collaboration diagrams.

To start with sequence diagrams, students may print out the output of the interaction and use this batch to draw a sequence diagram.

```
Sender@ #14: c 10, r 10, dir 0 carries 10 ready to run: waitForUser()
-> Sender@ #14: c 10, r 10, dir 0 carries 10.waitForUser() as Sender
  -> Sender@ #14: c 10, r 10, dir 0 carries 10.isFrontClear() as UrRobot
  true < -Sender@ #14: c 10, r 10, dir 0 carries 10.isFrontClear() as UrRobot
Loop 1 enter true
  -> Sender@ #14: c 10, r 10, dir 0 carries 10.areThereBeepers() as UrRobot
  false< -Sender@ #14: c 10, r 10, dir 0 carries 10.areThereBeepers() as UrRobot
  -> Sender@ #14: c 10, r 10, dir 0 carries 10.isFrontClear() as UrRobot
  true < -Sender@ #14: c 10, r 10, dir 0 carries 10.isFrontClear() as UrRobot
Loop 2 enter true
  -> Sender@ #14: c 10, r 10, dir 0 carries 10.areThereBeepers() as UrRobot
  true < -Sender@ #14: c 10, r 10, dir 0 carries 10.areThereBeepers() as UrRobot
Loop 1 enter true
  -> Sender@ #14: c 10, r 10, dir 0 carries 10.getBeeper() as UrRobot
  done < -Sender@ #14: c 10, r 10, dir 0 carries 11.getBeeper() as UrRobot
  ->Listener@ #15: c 3, r 4, dir 0 carries 0.beeperGetted() as Listener
    -> Listener@ #15: c 3, r 4, dir 0 carries 0.turnLeft() as UrRobot
    done < -Listener@ #15: c 3, r 4, dir 1 carries 0.turnLeft() as UrRobot
  done < -Listener@ #15: c 3, r 4, dir 1 carries 0.beeperGetted() as Listener
  ->Listener@ #16: c 7, r 4, dir 0 carries 0.beeperGetted() as Listener
    -> Listener@ #16: c 7, r 4, dir 0 carries 0.turnLeft() as UrRobot
    done < -Listener@ #16: c 7, r 4, dir 1 carries 0.turnLeft() as UrRobot
  done < -Listener@ #16: c 7, r 4, dir 1 carries 0.beeperGetted() as Listener
  -> Sender@ #14: c 10, r 10, dir 0 carries 11.areThereBeepers() as UrRobot
  false < -Sender@ #14: c 10, r 10, dir 0 carries 11.areThereBeepers() as UrRobot
```

Fig. 26, listing of the sequential execution of collaborating robots

This output is a good starting point. Students can translate this sequence of calls to robots into a sequence diagram. All what the students have to do is to design in top the three objects, and start drawing from top to bottom the messages sent between the robots. See Fig. 27 below.

Fig. 27, sender 2 listeners as sequence diagram

Also, in the case of sequence diagrams, after some training, students may write down the interaction they expect with a sequence diagram, and verify whether their expectations are met.

2.5   Statechart Diagrams

Even as worlds are simple, they still are rich. There is inheritance and aggregation. Talking about statechart diagrams, there are very interesting states. A full statechart diagram could look like the figure below.

Fig. 28, statechart diagram of a robot

I would strongly recommend to analyse this concurrent states individually in the beginning and to compile them to one diagram later in the course when introducing concurrent states in a statechart.

2.6 Class Diagrams

Drawing class diagrams is also something important. We can show the students that we mention in a class diagram all we think is important, and that we do not need to write down every detail.

Fig. 29, Class Diagram for Sender 2 Listeners example

A suggested representation for the example with sender and listeners shows the inherited operations and attributes. Attributes and operations that are not relevant for our example are omitted. The attributes beeperListener1 and beeperListener2 are represented through the association and the arrow that indicates in what direction the message is sent.

3   Conclusion

Different sign system are crucial for the transfer of knowledge to new situations. Analogies often brake and are therefore not always as adequate as UML diagrams.

Microworlds have not many details, but they contain all essential concepts. Therefore, they are excellent examples for all different kinds of UML diagrams. UML diagrams can be used for reflection and repetitions. We can hope so that students learn the visualised concepts better. The usage of attributes, visualised messages and a sequential top-to-bottom output in Roboworld supports the learning of UML-diagrams better than existing worlds do.

Chapter 9: Overall conclusion

Is there evidence for the effectiveness of Roboworld, and what can we learn from the analysis in this thesis and from the design of Roboworld about cognitive load theory?

1   Missing tests, but some evidence for usefulness of Roboworld

There is a major flaw of this Master thesis: I did not have the time to test Roboworld in classroom, yet. But I am quite confident for two reasons.

1.1   There Is Only Good Feedback about Karel++ Used in Classroom

Karel++ the robot is used at several institutions for higher education. I found nowhere any negative feedback about using Karel++ in classroom.

Larry Burks a teacher from Harvard University Graduate School of Design writes about Karel++:

> «I was looking for a way to introduce my students to object-oriented programming without scaring them out of the class in the first 10 minutes. Karel the Robot is a powerful aid to teaching because it has a very limited vocabulary, which is easily extensible and encourages students to build their own components quickly. It is a great way to introduce concepts that will be directly analogous in C++ and Java.»

Cathy LeBlanc, Assistant Professor of Computer Science Plymouth State College, Plymouth, NH uses microworlds also as project idea in her courses. She wrote about Karel++:

> «We've used Karel the robot and have had great success with beginning students trying to learn to program.»

People in this business have a narrow schedule. Therefore, I could not get any feedback in this short time elapsed, since I have set up a first pre-release with some scenarios.

1.2   Four Improvements of Roboworld Compared to Karel++

The second reason is: in comparison to Karel++ I added three enhancement that I think are crucial for the effectiveness of a microworld as steppingstone into object-oriented programming.

Firstly, elements to emphasise the idea of collaborating objects. These elements are (1) the instantiation tool, (2) the tool that lets user operate the robots manually to test single operations, (3) attributes that lets robots remember each other and build associations, and (4) the visual representation of messages between robots. See Chapter 6: Reducing cognitive load in Roboworld, starting at page 69.

Secondly, more compatibility with Java. I use a simplified Java code, not a mixture of Java and C++. There are also details related to Java. The

robots identification (class and serial number) is similar to Java. Positions and directions use the same logic as Java's class «Graphic».

Thirdly, elements to support the Unified Modelling Language (UML). The UML is important to get a transfer effect from the microworld to other design tasks. Without the students ability to coordinate different sign systems such as Robots interactions, Java-code, role-play with CRC-Cards (design by walking around), collaboration diagrams, sequence diagrams, class diagrams, there is less transfer of knowledge. As this is true for mathematics it must be true also for computer science, because the underlying problem of knowledge and comprehension is the same. See Chapter 8: Supporting the Unified Modelling Language, starting at page 88.

2 Conclusion from this Thesis

The question I asked in the beginning was, whether cognitive load theory can be this «rote rules to apply» as Wilson (1995) mentioned. Something used like the four factors for readable German text in Langer (1981). In this case, I should have been able to analyse and to improve existing instructional material.

2.1 Analysing Existing Instructional Material

I analysed existing methods to get around problems encountered while teaching Java. I rated the cognitive load, not distinguishing between intrinsic and extraneous cognitive load. I used my intuition to do so. But, my intuition of cognitive load could be represented in a scale, similar to the scales used by Langer (1974). It might look like this:

Heavy cognitive load			Low cognitive load	
- unknown items - heavily related items - several sources of information, needs splitted attention - redundant items without emphasis on the important information - items are out of order - important items are not marked up - several items must be considered simultaneously, because there are several steps to the goal			- known items - items can be considered separately - one source of information - no redundant items or emphasis on important information - items are in a logical order - straight forward method to solve the problem can be applied. - oral and visual information that go together well	
+ +	+	0	-	--

The thesis is: The heavier the cognitive load rated this way is, the more students will fail to learn.

I found quite often that I could relate problems experienced during teaching to cognitive load. And, that instructional material that was experienced as being efficient, often, had elements of lower cognitive load. Sometimes, I could relay not only on my experience, but also on experience of other teachers or students. For instance, on readers' ratings of books on amazon.com.

Based on this evaluation I would say that the cognitive load theory is a valuable tool to analyse instructional material. It might be used to choose the material to consider and what to put away. See Chapter 1: Cognitive load in a Java short course, starting at page 3, and Chapter 4: Existing solutions to overcome the problem of cognitive load, starting at page 30.

2.2 Improving Instructional Material

I checked Roboworld, which basic idea was to have a more Java oriented simpler version of Karel++, for cognitive load. I found different things that needs to be fixed or reconsidered. Thus, I would say cognitive load theory might be used to quality check instructional material under construction. Also, I could think about bringing information together, eliminating redundancy, or arranging goal free situations.

To conclude: I would state that cognitive load theory can really be used as rote rule to apply to instructional material and teaching. See Chapter 6: Reducing cognitive load in Roboworld, starting at page 69.

2.3 Cognitive Load Theory and Other Learning Theories

In a racing car, there is an engine that pushes the car forward. But, there are weight and aerodynamics that have to be considered too, because they might slow down the car. In analogy to such a racing car, we may see behaviourism, cognitivism and constructivism as the engine and cognitive load theory as weight and aerodynamics.

A typical example is the famous «constructed response frame sequences» Behaviourists loved to use it for their drill-and-practice. Because of their redundancy, these sequences are inefficient. But they can easily be replaced by flashcards. These flashcards (cards with question in front and answer in the back) use the same silly drill-and-practice as the «constructed response frame sequence». Only, they have less cognitive load.

Another example: Interesting material for explorative learning can be checked whether it is not too heavy for the students. A goal free task might reduce in such an explorative learning environment the cognitive load.

Semiotics emphasises that students must use different sign systems for real understanding. But the sign systems must be introduced the way they can be integrated without overloading the working memory.

Reducing cognitive load will therefore not get in the way of other learning theories. It will only refute some of their practices.

Appendix A: User Manual

1.1   Installation

A Java runtime environment must be installed at first. Then:

1. Copy the following two files to the same folder: scenarios.mws, roboworld.jar

2. With Windows operating system: double click roboworld.jar

The application starts up and shows the list of the existing scenarios:

3. Select one of the scenarios. Most evident examples are („Demo of two collaborating robots", „Sender-Listener, 2 Listeners", „Inheritance key-word super" and „Erroneous Robots with never ending execution") About the scenario **„Sender-Listener, 2 Listeners"** you find further information on page **3**. In all other examples robots are instantiated. In the lower left part of the window you select a robot in the list: „Choose Robot" and after that an operation in the list „Choose Operation". For instance:

Scenario Name	Robot	Operation
Erroneous Robots with never ending execution	ErroneousRobot@ # 3	endlessLoop() doSpecialTurn()
Demo of two collaborating robots	DemoRobot@ # 5	doTask()
Inheritance key-word super	BigHorse@ # 5	move()
Sender-Listener, 2 Listeners	Sender @ # 14	1. newListeners() 2. waitForUser() 3. see page

4. You may also open a new empty scenario.

When closing the window by clicking the close box, you get back from the scenario window to this window with the list of scenarios. You may:

- save changes to a scenario
- open a new scenario
- open a saved scenario
- dispose a scenario
- exit the application.

1.2   Inside a Scenario

Add and remove walls and beepers in the Scenario.

Compile an instantiable class

World with numbers, i. e. data (Beeper) and Objects (Robots) and walls

Choose class for editing code

Instantiate robot class

coloumn

row

direction

number of beep-

Feedback about code execution

active robot

operation

learning (macro) mode: the operation is entered in the code above.

Standard procedure:

1. Choose "new Class" in the list of class above the editing pane.
2. In the pane for code editing you define methods, by entering existing operations like move() or turnLeft(). See the doTask() template in the scenario „Demo of two collaborating robots".

100

3. Click Button Compile to make the class available for instantiation and check for elementary mistakes.
4. Click on **new** to instantiate a robot of the type of class left to the button new. Step for can also be executed as step 1 for existing robot classes like UrRoboter.

Result: The instance of the robot appears on screen to the right.

5. Choose the robot you wish to control in the list of robots.
6. Choose the operation, you like to send as message to the chosen robot.
7. With step and run you may either stepwise or non-stop run this operation.

With click on **Setup Map** the pane to the left changes. You can now define a world.

You may add walls and beepers or remove walls and beepers. You may even remove robots.

You may add walls and beepers while code is running.

This way you can interact with your robots. See for example the scenario

„Sender-Listener, 2 Listeners" call the operation newListeners() first. This instantiates two listeners that listen to the sender. Call **waitForUser()**.

Now the sender is checking permanently for beepers on the field where it stands. As soon as you **add beepers**, the sender sends the message beeperGetted() to the two listeners.

In the scenarios above, there could be several tasks for a trainee to perform:

- Remove all beepers in the room.
- Remove the beepers in the corner of the room.
- Remove beepers along the walls of the room.

Appendix B: Service Class for easy input, output in Java

My suggestion for a carefully commented Service class looks like this. You may add other functionality like inputting and outputting to file instead of input from keyboard or output to screen:

```java
import java.io.*;

/** this class provides basic input and output Services and
makes it possible to halt execution for some time.*/
abstract public class Services
{
        /** execution sleeps for a second
        @see sleep(double sec) for shorter or longer sleep times*/
        static public void sleep()
        {
                sleep(1.0); // call method sleep(double sec) here after
        }

        /** execution sleeps for number of seconds
        @param sec number of seconds, should be grater than 0.01 on most systems */
        static public void sleep(double sec)
        {
                try
                {
                        // Thread.sleep may thorw the InteruptedException
                        Thread.sleep((int) sec*1000);
                }
                catch (InterruptedException e)
                {
                        // The InterruptedException must be caught even if you do nothing in
                        // case of a Interruption
                }
        }

        /** write a line of code to standard output. This operation is called
        writeLine(), because this complies with java's naming convention.*/
        static public void writeLine(String s)
        {
                System.out.println(s); // System.out refers to the standard output (the screen)
        }

        /** write some text to standard output (the screen)*/
        public static void write(String s)
        {
                System.out.print(s);
        }

        /** promts for input and gets it. Input-Output-Exceptions that can
        occure with any access to your computer are caught (handeled) within
        this method.
        @param prompt the prompt to be presented to the user
        @return the input as String */
        public static String readLine(String pmt)
        {
                /* for decent input java uses a chain of responsibility
                        - object InputStream (System.in) is in contact with your computer and
                          is responsible for getting Keyboard input (a 8-Bit Ascii)
                        - the InputStreamReader transfers 8-bit ASCII-Code to 16-bit Unicode
                        - the BufferedReader provides methods like readLine so you do not
                          have to process any single character. *read() means that the method
                          read() is called repeatedly until the line is done.

                readLine()   _____   *read() _____  read() _____
                ---------->  | br : BufferdReader | ------->| isr : InputStreamReader | -------> | System.in : InputStream |
                <-Entire Line|_____|  <-Unicode|_____| <- ASCII  |_____|
                */
                InputStreamReader isr = new InputStreamReader(System.in);
                        // the constructor is ClassName( InputStream nextObjectInTheChainOfResponibility)
                        // so the chain is constructed from the end to the beginning
                BufferedReader br = new BufferedReader(isr);
                write(pmt); //calls write(String pmt) here after
                try
                {
                        // all access to your coumputers input-output devices may throw a IOException
                        String l;
                        l = br.readLine();
                        if (l.equalsIgnoreCase("exit"))
                        {
                                System.exit(0);
                        }
                        return l;
                }
                catch(IOException e)
```

```
                {
                        // the trown IOException is caught here
                        // return the name of the thrown exception instead of the line
                        return e.toString();
                }
        }

        /** promts for input of a number and gets it. Mal-formed-Number-Exceptions that
        might occure are caught (handeled) within this method.
        @param pmt the prompt to be presented to the user
        @return the input as a Number */
        public static double readNumber(String pmt)
        {
                String s = readLine(pmt);
                double out;
                try
                {
                        // the entry of the user might not be a number
                        // in this case a exception is thrown
                        out = (Double.valueOf(s)).doubleValue();
                        return out;
                }
                catch (Exception e)
                {
                        // the exception is caught here and written to screen
                        System.out.println(e);
                        // the return value is Not a Number because 0 would not be accurate
                        return Double.NaN;
                }
        }

}
```

Index

Bibliography

Beck, K., Cunningham, W. (**1989)** From the OOPSLA'89 Conference Proceedings, October 1-6, 1989, New Orleans, Louisiana , And the special issue of SIGPLAN Notices , Volume 24, Number 10, October 1989, http://c2.com/doc/oopsla89/paper.html

Beck, K., Gamma, E. (**2000**), JUnit A Cook'sTour, part of the JUnit distribution, doc\cookstour\cookstour.htm, available at ftp://www.armaties.com/D/home/armaties/ftp/TestingFramework/JUnit/

Bergin, J., Stehlik, M., Roberts, J., Pattis, R. (**1997)** Karel++ , A Gentle Introduction to the Art of Object-Oriented Programming, John Wiley and Sons, Inc, New York, online http://www.csis.pace.edu/~bergin/karel.html.

Bergin, J. (**1998)** Teaching Object-Oriented Analysis and Design in CS 1, online http://www.csis.pace.edu/~bergin/OOAD.html.

Bergin, J. (**2000)** An Object-Oriented Bedtime Story, published online http://www.csis.pace.edu/~bergin/Java/OOStory.html.

Bergin, J. (**2000a)** Why Procedural is the Wrong First Paradigm if OOP is the Goal, published online http://www.csis.pace.edu/~bergin/papers/Whynotproceduralfirst.html.

Bergin, J. (**2000b)** Objects in Operation, published online http://csis.pace.edu/~bergin/Java/oopanim.html.

Bergin, J. (**2001**) Fourteen Pedagogical Patterns, published online (under construction) http://www.csis.pace.edu/~bergin/PedPat1.3.html.

Bergin, J. (**2001a**) A Gentle Introduction to the Art of Object-Oriented Programming in Java, not published, but online (under construction) http://www.csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html.

Bergin, J. (**2001b**) Some Pedagogical Pattern, online http://www.csis.pace.edu/~bergin/pattern/fewpedpats.html.

Bloom, B., e. a. (**1956**), Taxonomy of Educational Objectives, The Classification of Educational Goals, Handbook 1, Harper, New York.

Boles, D. (**1999)** Programmieren spielend gelernt (mit dem Java-Hamster-Modell), B. G. Teubner, Stuttgart. Also online available at http://www-is.informatik.uni-oldenburg.de/~dibo/hamster.

Boone, B., Stanek, W. (**2000)** Java 2, Certification Exam Guide for Programmers and Developers, McGraw-Hill, New York.

Bishop, J. (**1997**) Java Gently, Programming Principles Explained, 1st edition, Addison-Wesley.

Campione, M., Walrath, K. (**1997)** The Java Tutorials, online 2001, documentation and language specifications, all available online (java.sun.com).

Cooper, G., **1998**, Research into Cognitive Load Theory and Instructional Design at UNSW, University of New South Wales, Australia, http://www.arts.unsw.edu.au/education/CLT_NET_Aug_97.HTML

Dörig, R., Waibel, R., **1995**, Schweizerische Zeitschrift für die kaufmännische Berufsbildung, I/1995.

Duval, R. (**1995**), Sémiosis et pensée humain, Registres sémiotiques et apprentissages intellectuels, Peter Lang SA, Bern.

Eckel, B. (**1998)**, Thinking in Java, Prentice Hall PTR, New Jersey, available free online: http://www.BruceEckel.com

Espich, E., Williams, B., **1967**, Developing Programmed Instructional Material, Fearon Publisher, Palo Alta, CA, USA.

Fowler, M., Kendall S. (**2000)** UML Distilled: A Brief Guide to the Standard Object Modeling Language, 2nd edition, Addison-Wesley.

Gamma, E., Helm, R., Johanson, Ralph, Vlissides, John (**1994)**, Design Patterns, Elements of Object-Oriented Software, Addison-Wesley.

Goguen, J. A., (**1996**), Semiotic Morphisms, http://www.cs.ucsd.edu/users/goguen/papers/sm/smm.html

Gibbons, J. (**1998)** Procedural Programming in Java, SIGPLAN Notices, V 33, N 4, April

Goll, J., Weiss, C., Rothländer, P. (**1999)** Java als erste Programmiersprache, B. G. Teubner, Stuttgart.

Hunt, J., McManus A. (**1998)** Key Java, Advanced Tips and Techniques, Springer, London.

Iding, M. K. (**1997**), How Analogies Foster Learning from Science Texts, Instructional Science (25): 233-253.

Jones, A. (**1984**), How Novices Learn to Program, Proceedings of the First IFIP Conference on Human Computer Interaction, INTERACT'84, London.

King, K. N. (**2000)** Java Programming: From the Beginning, Software online at http://knking.com/books/java/index.html

Krüger, G. (**2000)** Go To Java 2, Addison-Wesley, free online at http://www.javabuch.de.

Langer, I., Schulz von Thun, F., (**1974**), Messung komplexer Merkmale in Psychologie und Pädagogik. Ernst Reinhard Verlag, München.

Langer, I., Schulz von Thun, F., Tausch, R. (**1981**), Sich verständlich ausdrücken. Ernst Reinhard Verlag, München.

Lewis, M. D., (**1980**), Improving SOLO's User Interface: An Empirical Study of User Behaviour and a Proposal for Cost Effective Enhancement, Technical Report, No. 7, Computer Assisted Learning Research Group, The Open University, Milton Keynes, England.

Mendelsohn P., Green T.R.G. & Brna P. (**1990**), Programming Languages in Education: The Search for an Easy Start. In T. Green, T., Hoc, J.M., Samurcay, R., Gilmore, D., Psychology of programming (175-194). New York: Academic Press. Traduction française en ligne:

Metzger, Ch., Waibel, R., Henning, C., Hodel, M., Luzi, R., **1993**, Anspruchsniveau von Lernzielen und Prüfungen im kognitiven Bereich, Studien und Berichte des IWP, Heft 10, Institut für Wirtschaftspädagogik an der Hochschule St. Gallen.

Niemann, A. (**2001)** Objektorientierte Programmierung in Java, 2. A., verlag moderne industrie Buch, Kaarst.

Object Management Group, Inc. (**2000)** Unified Modeling Language, Specification, vers. 1.3, March, published online

Papert, S. (**1980**), Mindstorm: Children, Computers and Powerful Ideas, New York: Basic Books.

Rajan, T. (**1990**), Principles for the Design of Dynamic Tracing Environments for Novice Programmers. Instructional Science, 19(4/5):377-406.

Schröter, Brit, Plank, Johann (**2001)** Selfjava, version 1.08, http://www.selfjava.de/

Schildt, Herbert **(2001)** Java 2: A Beginner's Guide, Osborn/Mc Graw Hill, New York.

Sweller, J. (**1988**). Cognitive load during problem solving : Effects on learning. Cognitive Science, 12, 257-285.

Sweller, J., Chandler, P., Tierner, P., & Cooper, M. (**1990**). Cognitive load in the structuring of technical material. Journal of Experimental Psychology; General, 119, 176-192.

Wilson, Brent G., (**1995**), Maintaining the Ties between Learning Theory and Instructional Design, paper presented at the meeting of the American Educational Research Association, San Francisco, March1995., Available at: http://www.cudenver.edu/~bwilson