Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Object-Oriented Programming through the Lens of Computer Science Education

Marc-Pascal Berges

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:

Univ.-Prof. Dr. Johann Schlichter

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Peter Hubwieser
2. Univ.-Prof. Dr. Torsten Brinda, Universität Duisburg-Essen

Die Dissertation wurde am 07. April 2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02. Juli 2015 angenommen.

Abstract

In recent years, the importance of the object-oriented paradigm has changed significantly. Initially it was mainly used in software engineering, but it is now being used more and more in education. This thesis applies methods of educational assessment and statistics to object-oriented programming and in doing so provides a broad overview of concepts, teaching methods, and the current state of research in this field of computer science.

Recently, there have been various trends in introductory courses for object-oriented programming including *objects-first* and *objects-later*. Using current pedagogical concepts such as *cognitive load theory*, *conceptual change*, and *self-directed learning* in the context of this work, a teaching approach that dispenses almost entirely of instruction by teachers was developed. These *minimally invasive programming courses* (MIPC) were carried out in several passes in preliminary courses in the Department of Computer Science at the TU München. The students were confronted with a small programming task just before the first lecture. Using worksheets that were based on the objects-first approach, the essential object-oriented programming concepts were presented with brief explanations. A student tutor was set aside to support each group.

The results of the courses have been investigated in various ways. First, changes in conceptual knowledge with concept maps were collected. Second, the program code that was produced was examined. Among other things, we evaluated which concepts were implemented by the novice programmers. The combination of these two methods of investigation provides a link between knowledge and ability. It turns out that there are concepts in programming that can be applied before a classification or representation in existing knowledge structures takes place. The correlation of these results with the personal data collected such as gender, school experiences or previous programming knowledge leads to a detailed picture of the skills and abilities of participating students.

Additionally, the psychometric approach of item response theory was applied to the program code. This methodology provides an opportunity to evaluate programming abilities on the basis of programming tasks embedded in small or large projects. The basic idea is that items are created representing observable structural elements in the program code. Based then on the code items applied on the data of the MIPC course, a valid dataset was examined with reference to the Rasch model.

A graphical illustration is introduced to visualize the definitions of the concepts covered in the course. The so-called concept specification map (CSM) connects concepts by their definitions. Since there are seldom "complete" definitions in a single place in textbooks, CSM considers specifications of concepts in a larger area. By linking the individual specifications, an overview is achieved of how a specific concept is defined in a given textbook. Furthermore, these important dependencies for teaching are discussed and an example of a qualitative analysis of textbooks is presented.

Zusammenfassung

In den letzten Jahren hat sich die Bedeutung des objektorientierten Paradigmas ganz erheblich gewandelt. Anfangs wurde es vorwiegend in der Softwaretechnik angewandt, inzwischen geschieht dies aber auch mehr und mehr in der Lehre. Die vorliegende Arbeit wendet geisteswissenschaftliche und statistische Methoden auf die objektorientierte Programmierung an und liefert dadurch einen breiten Überblick über Konzepte, Lehrmethoden und die aktuelle Forschung in diesem Bereich der Informatik.

Speziell zur Einführung in die objektorientierte Programmierung gibt es in den letzten Jahren verschiedene Strömungen wie *objects-first* und *objects-later*. Unter Anwendung aktueller didaktischer Konzepte wie der *Cognitive Load Theory*, *Conceptual Change* und *Self-Directed Learning* entstand im Rahmen dieser Arbeit ein Lehrkonzept, das nahezu komplett auf Instruktion durch Lehrpersonen verzichtet. Diese sog. *Minimally Invasive Programming Courses* (MIPC) wurden in mehreren Durchläufen im Rahmen der Vorkurse der Fakultät für Informatik an der TU München durchgeführt. Dabei wurden die Studierenden bereits vor der ersten Vorlesung mit einer kleinen Programmieraufgabe konfrontiert. Mithilfe von Arbeitsblättern, die sich am Ansatz *objects-first* orientieren, wurden die zur Lösung notwenigen Konzepte der objektorientierten Programmierung mit kurzen Erklärungen vorgestellt. Zur Unterstützung wurde jeder Gruppe ein studentischer Tutor zur Seite gestellt.

Die Ergebnisse der Kurse wurden auf verschiedenste Weise untersucht. Zum einen wurden die Veränderungen von konzeptuellem Wissen mit *concept maps* erhoben. Außerdem wurde der produzierte Programmcode untersucht. Dabei wurde u.a. ausgewertet, welche Konzepte die Programmieranfänger umgesetzt haben. Die Verbindung dieser beiden Untersuchungsmethoden liefert einen Zusammenhang zwischen Wissen und Können. Dabei wird erkennbar, dass es Konzepte in der Programmierung gibt, die angewendet werden können, bevor eine Einordnung beziehungsweise Repräsentation in vorhandene Wissensstrukturen stattfindet. Die Korrelation dieser Ergebnisse mit den erhobenen Personendaten wie zum Beispiel Alter, Geschlecht, Schulerfahrung oder Vorkenntnisse in der Programmierung zeigt ein ausführliches Bild der Fähigkeiten und Fertigkeiten der teilnehmenden Studierenden. Zudem lässt sich die Frage beantworten, ob es verschiedene Arten von Programmieranfängern gibt.

Des Weiteren wird der psychometrische Ansatz der Item-Response Theorie auf Quellcode angewendet. Dieses Vorgehen ermöglicht es Programmier- bzw. Codierfähigkeiten auf der Basis von in Projekten eingebundenen Programmieraufgaben zu evaluieren. Die grundlegende Idee hinter dem Verfahren ist die Generierung von Items, die strukturelle Elemente des Programmcodes repräsentieren. Auf Basis der Quellcodes aus den MIPC-Kursen wurde ein gültiges Item-Set im Sinne eines Rasch-Modells ermittelt.

Um die in einem Kurs behandelten Konzepte umfassend zu definieren und in einen Zusammenhang zueinander zu bringen, wird eine spezielle graphische Veranschaulichung eingeführt. Die sog. *Concept Specification Map* (CSM) verbindet Konzepte durch ihre Definitionen. Da es in Lehrtexten selten “vollständige‘” Definitionen an einer einzigen Stelle gibt, berücksichtigt die CSM Spezifikationen von Konzepten in einem größeren Textabschnitt. Durch die Verknüpfung der einzelnen Spezifikationen ergibt sich ein Überblick, wie bestimmte Konzepte im jeweiligen Lehrbuch definiert werden. Außerdem werden dadurch auch wichtige Abhängigkeiten für die Lehre aufgezeigt. Am Beispiel einer qualitativen Textbuchanalyse werden die Einsatzmöglichkeiten vorgeführt.

Acknowledgments

The work on a thesis is an interesting and challenging task. First and foremost, it is a big trial for oneself. None of this would be possible without the support of many. So, I want to thank those people at the very beginning.

First, I want to thank Prof. Dr. Johann Schlichter for heading the examination committee. Furthermore, I thank Prof. Dr. Torsten Brinda, my second supervisor, for his patience, his advice, and the fast evaluation.

A thesis cannot be written without the assistance of very pleasant colleagues. I want to thank you all for the stimulating discussions. Namely, Alexander Ruf and Dino Capovilla, who accompanied me for a long time, should be mentioned here. A very special thanks to Andreas Mühling who was always there to listen to my ideas and to criticize them. The work with him on the gap between knowledge and abilities was very interesting. Additionally, the theoretical clarification of the methods around item response theory and cluster analysis made a huge contribution to the success of this work.

To write a thesis without supervision is not possible at all. But, supervision is not the only necessity during the writing process. In Germany we have the word "Doktorvater" which perfectly describes my supervisor Prof. Dr. Peter Hubwieser. During the last seven years, he has supported me in all my endeavors. It was and is a great pleasure to work with him. Without his encouragement, especially during the end of the writing phase, this thesis would have never been finished.

Last but not least I have to thank my wife. As I have written in the beginning of these acknowledgments, a thesis is a great adventure and a big trial for oneself. Without her support and the patience, I would not have written these lines. Thank you for everything.

Contents

1 Introduction

1.1 Problem Setting and Motivation

Technology that is related to computer science is changing very fast. While it was difficult to get access to a computer several years ago, it is now common to have access to computers and the internet wherever you are and whenever you want. Furthermore, children of today are accustomed to using computers, as well as the concepts related to computer science, in their daily lives.

> “Today, I have a computer in my pocket that is more than 100,000 times faster and has 10,000,000 times more memory than a ZX81. It is connected to every other computer on the planet and can access virtually every piece of human knowledge ever created, nearly instantaneously. The pace of change in computing is extraordinary.”(Crow 2014)

In this time of change the demands on the educational system are higher than ever. For this reason there have been a huge variety of attempts in recent years to change computer science education. In the Bavarian grammar school a compulsory subject based on the notions of object orientation has been introduced (Hubwieser 2012). Great Britain introduced a computer science subject in schools (The Royal Society 2012). The biggest associations for computing and computer science education have revised their curricula to face the new challenges in computer science subjects (ACM/IEEE-CS Joint Task Force on Computing Curricula 2013). Furthermore, associations that focus on computer science education – such as teacher associations (Computer Science Teachers Association (CSTA)) or working groups of national computing associations (Gesellschaft für Informatik (GI)) – have introduced new standards. Additionally, there are some first competency models that try to define a measurement for education in computer science.

In the 1990s there was a paradigm shift in programming notions in industry. The time before was characterized by procedural programming notions. Since this shift, the object-oriented paradigm has become state of the art, although there were many discussions of whether or not object orientation was only coming up for a moment (see for example, Broy and Siedersleben 2002; Jähnichen and Herrmann 2002). Transfer of the paradigm shift in education can be dated about ten years later after the millennium.

> “[T]he shift was reality and programming instructors could no longer vacillate or attempt to ignore it. Action was demanded to begin the work of fleshing out the new paradigm and learning how to work within it. The only questions raised concerned what does this mean for the introductory

course, how can this be done in my course? Most faculty are [...] willing to change and desiring be up-to-date. They know, however, that the paradigm shift means more than changing programming languages. [...] We should all be encouraged by the fact that the problems of teaching introductory programming from the object perspective are recognized, that a variety of strategies are being explored, and by the promise that the revolution will quickly produce a flood of new and better pedagogical aids." (Mitchell 2000, p. 105)

According to Mead et al. (2006), "[t]eaching programming has three basic components: curriculum, pedagogy, and assessment" (p. 182). As mentioned above, the curriculum and pedagogical aspects are based on a broad theory published in literature. The assessment aspects are also investigated in many studies. However, most of them have their origins in software engineering or questionnaires on concepts underlying the programming skills. In particular, the programming tasks are important for educational purposes. The knowledge and abilities of novice programmers are especially of great interest.

"Programming is a very useful skill and can be a rewarding career. In recent years the demand for programmers and student interest in programming have grown rapidly, and introductory programming courses have become increasingly popular. Learning to program is hard however. Novice programmers suffer from a wide range of difficulties and deficits." (Robins et al. 2003, p. 137)

The observations of Robins et al. (2003) are the motivation for this thesis. According to them, the variety of novice programmers' previous knowledge is very diverse. The investigations that are provided in the upcoming chapters present methodologies to analyze the heterogeneity among novice programmers' knowledge and abilities. Furthermore, the notions of object orientation are systematically investigated and a new methodology of displaying the interdependencies of the underlying notions is introduced. For this purpose an experimental introductory course based on several notions that employ the social cognitive learning theory of Bandura (1977, 1986) are introduced and investigated. The experimental course was a preliminary course before the first semester started.

1.2 Research Questions

After presenting the general motivation for this thesis, the research questions are addressed. They follow the need to investigate novice programmers' knowledge and abilities to address the previously mentioned problems. So, this thesis can be divided into three parts. After presenting a theoretical framework for my studies in the first part, an overview is given on computer science education and the representation of object orientation and object-oriented programming within this field. This leads to the first two research questions which are answered in Chapter 5:

RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competency models?

Usually, educational planning processes are guided by certain documents, for example standards and curricula. As computer science education is quite a new subject in relation to the established subjects such as mathematics or natural science, there is still a lot of change in those documents. In addition, the formal definition of competencies in computer science education is in the early stages. Since this thesis concentrates on computer science education and especially object-oriented programming, examples of the documents mentioned before are analyzed on their coverage of object-oriented concepts.

RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?

Another important aspect in computer science education is the strategy of introducing specific topics, particularly programming aspects. As the relevance of object orientation for introductory programming courses has grown during the last decade with the paradigm shift, different approaches have been investigated and discussed in a broad manner. But, which strategies could prevail in educational practice?

There are a lot of concepts related to object orientation. This provides a serious challenge for both the learners and the instructors. Textbooks for introductory courses face this challenge in a very special way. Each course and each instructor has their own way of introducing object-oriented concepts. To determine if the structure of the course and the structure of the corresponding literature or course materials fit together, the following two research questions have to be answered in Chapter 6:

RQ3: How to represent logical interdependencies between the concepts of object orientation?

Only a minority of textbooks contain a summary of clear definitions of the concepts used within the texts, or even give an overview how the concepts are related to each other. To compare course materials and other literature, a methodology for representing definitions in a clear and comparable way has to be developed. Furthermore, the term of definition is mostly not suitable for literature. In most cases, there are only specifications of concepts spread over the complete text.

RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?

As novice programmers may use textbooks intensively, the textbooks of introductory courses have to fit the conceptual structure of the course. Concerning the literature that is recommended for introductory courses, differences and structures are shown to demonstrate the method of graphically representing the conceptual structure.

Besides structuring concepts around object-oriented programming, investigating the knowledge and the abilities of novice programmers are central concerns of this thesis. New approaches for categorizing and evaluating novice programmers' knowledge and abilities are the main purpose of the investigations. Marginally, the investigation of an extreme course design agrees with this research. All these aspects meet at the last few research questions that will be answered in Chapter 7:

RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?

By minimizing instruction, an attempt is made to determine which concepts are used first in relation to the participants' previous knowledge. In contrast to investigations on the impact of a course design, which follows a specific order of the topics, in the courses underlying this investigation the participants chose their own order according to the theory of self-directed learning.

RQ6: How to evaluate knowledge and abilities of novice programmers?

By using the statistical method of cluster analysis, groups are identified in the novice programmers' knowledge and abilities and the differences between these groups concerning their programming knowledge and abilities are investigated.

RQ7: Is there a difference between understanding the concepts of programming and applying them?

There is evidence that in programming it is possible to apply certain concepts without understanding them. By conducting a test on the knowledge of novice programmers and assessing their programming abilities at the same time, this suspicion is investigated.

RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

In classical test theory (CTT) program code can be analyzed to find evidence for the abilities of programmers. But, the questions have to be posed directly. Instead, in item response theory (IRT) only the probability of a right answer is of interest. To examine the programming ability in its complexity, programming cannot be investigated by small programming tasks. In fact, programming is a latent procedure with many involved psychometric constructs. A programming task contains a lot of implicit items that have to be solved in order to successfully complete the task. The results of the items that have been solved can be found in the resulting programs.

1.3 Methodology and Structure

This thesis is based on several research methodologies. These are introduced in the following chapters. Here, the structure and a small description of the content is provided.

Chapters 2-4: Theories underlying the investigations
In the chapters on theoretical background, the notions and theories underlying the studies conducted in this thesis are presented. This implies the notions and concepts of object orientation and the development of the corresponding paradigm. Furthermore, some educational theories that were important for designing the experimental course setting for my investigations are shown. These are the cognitive load theory and the theories concerning self-directed learning. Additionally, the conceptual change theory is presented as it is important for understanding how programming notions are learned and can explain the difficulties of the paradigm change in programming. In addition to the underlying theories, there are several methodologies applied during the data analysis. Concept mapping is presented as a methodology to externalize knowledge structures. Furthermore, the statistical methodology of cluster analysis is presented. Finally, a theoretical framework for the psychometric approach of item response theory (sometimes called latent trait theory) is provided.

Chapter 5: Representation of object orientation in an educational context
After introducing the object-oriented notions and their development, the representation of object-oriented programming in standards, curricula, and competency models is the main focus. Exemplary literature is reviewed and presented. For the educational methodologies, a broad variety of literature throughout the last decade is examined and the main results are presented. Here, the first two research questions (**RQ1** and **RQ2**) are answered.

Chapter 6: A graphical representation of concept specifications
A method for representing the main concepts and their definition of specification in a text is presented. After defining a graphical representation, this is applied on two different kinds of literature. The first example is a single paper that defines in a very clear way the basic concepts of object orientation. This demonstrates how to present the provided concepts in a compact way (answer to **RQ3**). The second example is based on a semi-automatic text analysis. In a first step, text passages containing a specific keyword are examined from a given text. The rating (coding) process is done with regard to the qualitative text analysis of Mayring (2010). Here, the research question addressing the representation of object-oriented concepts in textbooks (**RQ4**) is answered.

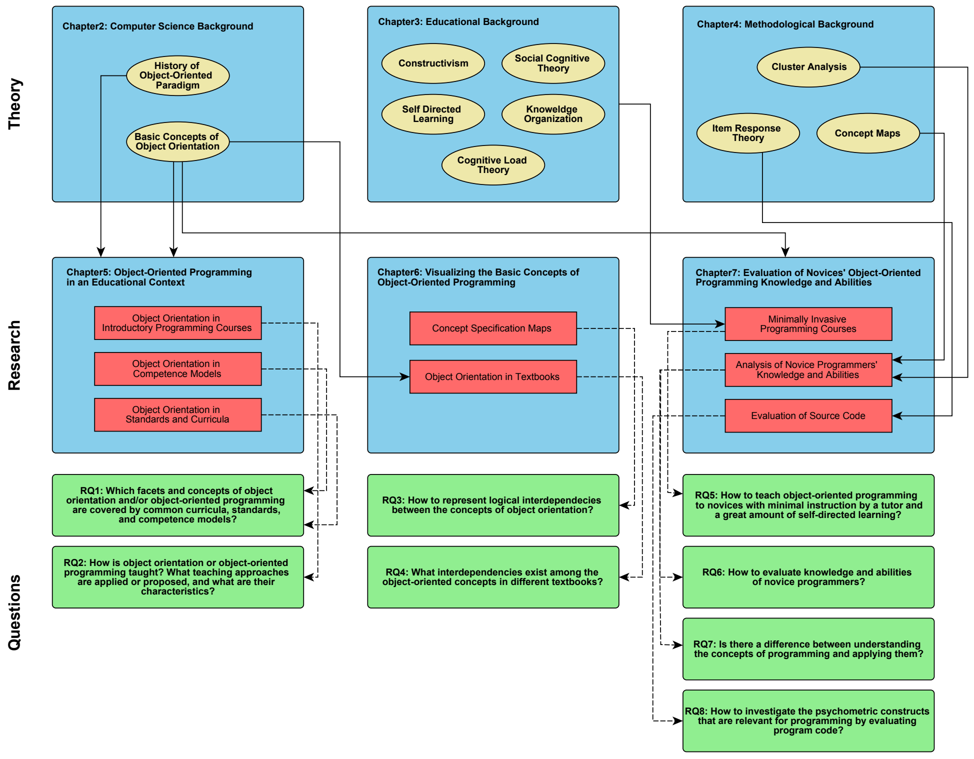
Chapter 7: Evaluation of novice programmers' object-oriented programming abilities and their corresponding knowledge
Here, an experimental introductory course about object-oriented programming is evaluated. The design of the course is based on the theory of self-directed learning and tries to avoid instruction during the programming task. The given materials are structured and investigated with help from the graphical representation method introduced in this

thesis (answer to **RQ5**). In addition, knowledge of the participants is assessed by evaluating concept maps (answer to **RQ7**). Furthermore, the programming abilities are measured with a new approach based on the item response theory conducted on the programs (answer to **RQ8**). Additionally, a cluster analysis on the programs produced provides differences within the novice programmers' knowledge and abilities. This answers research question **RQ6**.

Chapter 8: Conclusion
The final chapter provides ideas for future investigations. Questions that arose during the investigations of this thesis are presented. Furthermore, the presented investigations are summarized. More precisely, the research questions presented in the previous section are checked to determine whether the provided studies could answer them. Conclusions on all results are drawn.

Appendix:
Excerpts from the course materials and the resulting products are given in the appendix. Additionally, the graphic representation of the textbook analysis is shown in the appendix due to space reasons.

The following graphical scheme represents the content and structure of this thesis. Each chapter is displayed with its major content. For the chapters related to theoretical background, the theories are displayed as ellipses. The research presented in this thesis is shown as squares in the related chapters. Additionally, for each chapter the corresponding research questions are shown. The scheme is presented after each chapter. The content of the preceding chapter is in color, while the rest of the thesis is in gray color.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

2 Computer Science Background

As mentioned in the introductory chapter, the research that was conducted for this thesis is based on theories from different fields. This first chapter introduces theories and concepts from the field of computer science. In particular, the concepts underlying object-orientation and the theory of the paradigm shift in programming are presented in this chapter. In Chapter 6, methodology for a systematic display of the object-oriented concepts is introduced. Additionally, the theory in this chapter is the underlying basis for the experimental courses of Chapter 7.

2.1 Small History of the Object-Oriented Paradigm and the Corresponding Programming Languages

> “[The] term object-oriented programming is derived from the object concept in the Simula 67 programming language.” (Nygaard 1986, p. 128)

The early definitions of object orientation in literature are mostly based on object-oriented programming. In this section the historical development of the paradigm and the concepts are shown. Furthermore, the history of object-oriented programming languages is related to the development of the paradigm.

Although there are many different paradigms in literature, there is, according to Mitchell (2000), only one paradigm shift according to the definition of Kuhn (1996). Kuhn introduced the notion of a paradigm as an achievement or theory that fulfills two characteristics. First, the theory is “unprecedented to attract an enduring group of adherents away from competing modes of scientific activity”. Second, it is “sufficiently open-ended to leave all sorts of problems for the redefined group of practitioners to resolve” (p. 10).

Additionally, Kuhn defined the shift between two paradigms. Triggered by a crisis when a problem cannot be solved with the predominant paradigm, the scientific group tries to find solutions or work-arounds within the paradigm. If the effort for the work-arounds rise, the old paradigm is replaced by a new one that fulfills the definition of a paradigm for the new set of problems (cf. Kuhn 1996, pp. 77).

As a result of the difficulties with programming low-level machine languages, higher-level languages and the imperative paradigm were introduced in the late 1950s. The **imperative paradigm** is based on the idea of a step-by-step execution model of commands and statements. The order of the execution is controlled by control statements.

Additionally, declarative statements provide a name to a value, thereby creating variables. A synonym for the imperative paradigm is the **procedural paradigm**. Although they are used as synonyms, there is the slight difference of added functions or procedures, respectively (cf. Jana 2005; Minarova 2013). A model of the procedural programming is displayed in Figure 2.1

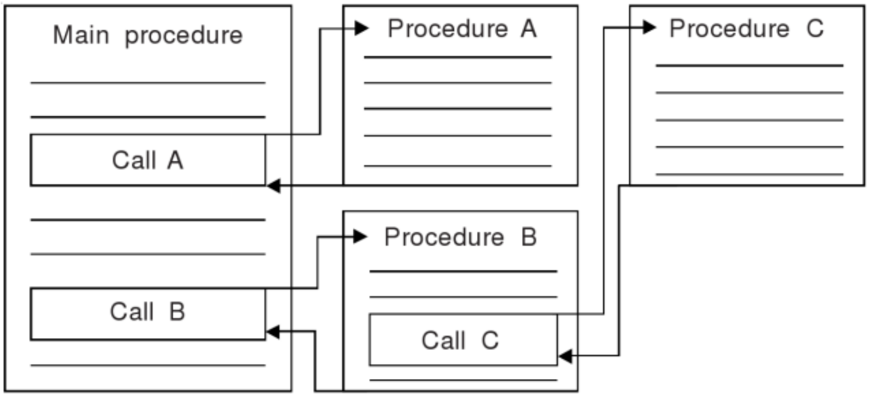


Figure 2.1: Procedural programming model (Jana 2005, p. 7)

In addition to the imperative procedural paradigm several other paradigms were introduced during the 1960s, for example the **functional paradigm** or the **logical paradigm**. The functional paradigm is based on functions. More precisely, all computations are done by calling or applying functions. In difference to the imperative paradigm, here, the result of one computation is the input for the next one. The logical paradigm is based on axioms, inference rules, and queries. New facts are derived from known facts by applying the given rules. As mentioned at the beginning of this section the imperative procedural paradigm is the first programming paradigm. It attracts a reasonable number of programmers (cf. Jana 2005).

In the 1980s, the invention of computers that were affordable for many people caused an increased effort in the development of graphical user interfaces. Additionally, the software crisis that was induced by the necessity for more security in software led to conflicts with the procedural paradigm that was common in that time (cf. Luker 1989). The result was the structured paradigm which is a subset of the **procedural paradigm**.

According to the definition of a paradigm shift by Kuhn (1996), the shift took place in the early 1990s. In addition to the problems mentioned above, the need of re-usability of program code and the increasing number of embedded systems made a work-around within the procedural paradigm more difficult and therefore caused the shift to the **object-oriented paradigm** (cf. Luker 1989, 1994). This paradigm is based on the notion of communication between objects as unit of data and behavior (cf. Jana 2005). A model of the object-oriented paradigm is displayed in Figure 2.2

Interestingly, it took about ten years until the shift found its way from industry into education. Another ten years later the shift in education is complete and nearly every

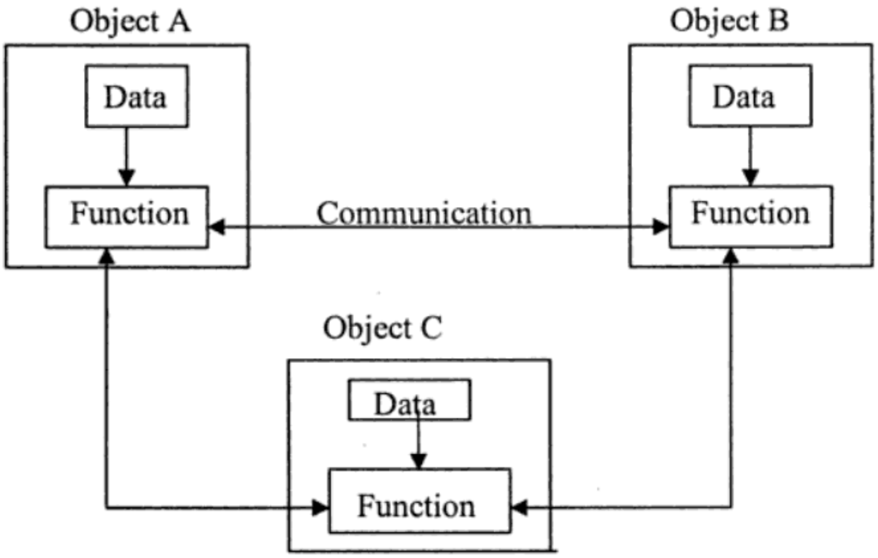


Figure 2.2: Organization of data and behavior (function) in object-oriented programming (Kedar 2007, p. 171)

university is teaching the object-oriented paradigm as the introduction into computer science (cf. Luker 1989, 1994).

The change in ideas is illustrated by Henderson-Sellers (1992). As an example, a model of a counter is presented in both paradigms. The focus is on the shift in the view on functionality. While the procedural counter needs a definition and initialization of the counter variable followed up by a loop that modifies the counter and checks the probable range violation, the object-oriented counter is simply modeled with the needed functionality and is used as it is. The implementation itself is hidden in the counter object and is irrelevant to the user (cf. Henderson-Sellers 1992, p. 44).

The object-oriented paradigm was developed by multiple people over a long period of time (cf. Capretz 2003, p. 1). In fact, the development of object orientation and especially the notion of objects started in the 1960s, together with the development of the discipline itself.

The development of languages can also be seen as an indicator for the paradigm shift. As with other notions, object orientation is also bound to the development of specific programming languages. But, which languages can be called object-oriented and why? Stroustrup (1988) gives quite a simple answer: “A language is said to *support* a style of programming if it provides facilities that makes it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or exceptional skill to write such programs; it merely *enables* the technique to be used” (p. 2).

The very first object-oriented programming language known is **Simula**; developed by Ole-Johan Dahl and Kristen Nygaard (cf. Capretz 2003, p. 3). Its development started in 1962 and ended with **Simula67** in 1967. Nevertheless, the development has continued and Simula is still active in some special environments (cf. Black 2013, p. 4). In Simula67 the central role of objects and classes was introduced. Resulting from the success of the language, the notions of object, class, and inheritance became widely accepted.

> "The main impact of Simula 67 has turned out to be the very wide acceptance of many of its basic concepts: objects, but usually without own actions, classes, inheritance, and virtuals, often the default or only way of binding 'methods' (as well as pointers and dynamic object generation)." (Dahl 2002, p. 86)

A sample code for class definition, object instantiation, and inheritance is presented in Listing 2.1.

Listing 2.1: Sample code in **Simula67**

```
Class Rect (Width, Height); Real Width, Height;
                           ! Class with two parameters;
 Begin
    Real Area, Perimeter;  ! Attributes;

    Procedure Update;      ! Methods;
    Begin
      Area := Width * Height;
      Perimeter := 2*(Width + Height)
    End of Update;

    Boolean Procedure IsSquare;
      IsSquare := Width=Height;

    Update;                ! Life of rectangle started at creation;
 End of Rect;
```

Simula has been followed up by several languages that support the notion of object orientation on different levels. The first language that made the term "object-oriented" prominent was **Smalltalk** (cf. Capretz 2003, p. 2; Rentsch 1982, p. 51). It was invented in the 1970s under the leadership of Alan Kay and contains some major ideas that emphasize the object-oriented characteristic. In Smalltalk all elements are objects; more precisely, there are no primitive data types. The communication between objects is realized according to the concept of data encapsulation and information hiding. The only possibility for communication is sending and receiving messages. Additionally, the objects have their own memory space that is only accessible within the object itself. Finally, every object is an instance of a class that holds the shared behavior for its instances (cf. Kay 1993, p. 19). A sample code with all these concepts is presented in Listing 2.2.

Listing 2.2: Sample code in **Smalltalk**

```
Class Smalltalk.Rect

Superclass: Core.Object
Type: none
Instance variables: Area Perimeter Width Height

Rect class methods for 'instance␣creation'
width: aWidth height: aHeight
|temp|
temp := self new width: aWidth height: aHeight.
temp Update.
^temp

Rect methods for 'accessing'
height: aHeight
Height := aHeight

width: aWidth
Width := aWidth

Rect methods for 'updating'
IsSquare
^Width=Height

Update
Area := Width * Height.
Perimeter := 2 * (Width + Height).
```

Following the first object-oriented languages, in the 1980s several improvements to the languages have been implemented and new ideas have been included. The language **Eiffel** was developed in the 1980s by Bertrand Meyer.

> "Eiffel is a language and environment intended for the design and implementation of quality software in production environments. The language is based on the principles of object-oriented design, augmented by features enhancing correctness, extendibility and efficiency; the environment includes a basic class library and tools for such tasks as automatic configuration management, documentation and debugging." (Meyer 1987*a*, p. 85)

The handling of multiple inheritance was a new contribution to programming languages. The correct treatment through renaming and the introduction of assertion and invariant mechanisms supported this basic object-oriented feature. Furthermore, it was the first programming language that introduced static typing in an object-oriented language (cf. Meyer 1987*a*, p. 94). Listing 2.3 presents a small class definition in the language Eiffel.

In the 1980s **C++** was invented by Bjarne Stroustrup (Stroustrup 1994). It is based on the procedural programming language **C** and "was designed to support data abstraction, object-oriented programming, and generic programming in addition to traditional C programming techniques" (Stroustrup 2003). The major change to C is the introduction of the class concept. Based on this, the concepts inheritance and generalization are added as well. Again, like in Eiffel, multiple inheritance is possible. Additionally, there

Listing 2.3: Sample code in **Eiffel**

```
class RECT
create
   Make

feature{NONE}  --constructor
   Make(aWidth, aHeight: REAL)
      do
         Width := aWidth
         Height := aHeight
         Update()
      end

feature
   Width : REAL
   Height : REAL
   Area : REAL
   Perimeter : REAL

feature
   Update
      do
         Area := Width * Height
         Perimeter := 2 * (Width + Height)
      end
   IsSquare : BOOLEAN
      do
         Result := Width=Height
      end
end
```

are functions and methods supporting the two paradigms (cf. Sebesta et al. 2013, pp. 108). An example of a class definition is presented in Listing 2.4.

Based on the procedural language **Pascal** (Wirth 2002*a*,*b*), another group of languages supporting procedural elements, as well as object-oriented notions, was developed in the time before the paradigm shift. Again, the object-oriented notions of object, attribute, and method were added to an existing language. In the 1980s Borland developed **Turbo Pascal** with object-oriented additions and later **Delphi** for Windows platforms. The basic language underlying Delphi is called **Object Pascal**. Listing 2.5 presents a small class definition in Object Pascal.

Finally, **Java** was developed in the mid-1990s. The language was derived from C++. Most concepts of C++ were implemented in Java. Yet, some were modified to make the language more reliable (e.g., references). Other concepts were not implemented (e.g., multiple inheritance). In Java all functionality is bound to either an object or a class. Methods can only be accessed through the corresponding class or object. Although Java does not support multiple inheritance in a direct way, interfaces provide restricted substitute (cf. Sebesta et al. 2013, pp. 111). While in its first years Java was mainly used on the internet, now “Java is widely used in [the] IT industry and academic environments” (Vujošević-Janičić and Tošić 2008, p. 73). Listing 2.6 presents a small sample class.

Listing 2.4: Sample code in **C++**

```
class Rect {
    public:
        void Update();
        Rect(double aWidth, double aHeight);
        bool IsSquare();
    private:
        double Height,Width,Area,Perimeter;
};

Rect::Rect(double aWidth,double aHeight) {
    Height = aHeight;
    Width = aWidth;
}

void Rect::Update() {
    Area = Width*Height;
    Perimeter = 2 * (Width + Height);
}

bool Rect::IsSquare() {
    return Width==Height;
}
```

In addition to the languages described in this section, there are many more programming languages for all existing programming paradigms and programming styles. Several overviews try to provide an almost complete genealogy of programming languages. For example, there are lists[1] and graphical overviews[2][3] of the development of programming languages on the internet. Figure 2.3 presents an overview of all previous described languages. It is adopted from (Zuse 1999). The languages described above are in cyan color.

Some important features of the languages introduced above are displayed in Table 2.1 (adopted from a similar overview of Hristakeva and Vuppala (2009)).

Characteristic	Language				
	Smalltalk	Eiffel	C++	Object Pascal	Java
Object orientation	Pure	Pure	Hybrid	Hybrid	Hybrid
Typing	Dynamic	Static	Static	Static	Static
Inheritance	Single	Multiple	Multiple	Singe	Single (Extended)
Method overloading	No	No	Yes	Yes	Yes

Table 2.1: Important feature of the presented object-oriented programming languages

[1] http://people.ku.edu/˜nkinners/LangList/Extras/langlist.htm - last access: 09.12.2014
[2] http://www.levenez.com/lang/ - last access: 09.12.2014
[3] http://oreilly.com/pub/a/oreilly/news/languageposter_0504.html - last access: 09.12.2014

Listing 2.5: Sample code in **Object Pascal**

```
type
 Rect = object
            Width: double;
            Height : double;
            Area : double;
            Perimeter : double;
            constructor init(aWidth, aHeight : double);
            procedure Update;
            function IsSquare : boolean;
           end;

constructor Rect.init(aWidth, aHeight : double);
 begin
  Width := aWidth;
  Height := aHeight;
  Update();
 end;

procedure Rect.Update;
 begin
  Area := Width*Height;
  Perimeter := 2 * (Width + Height);
 end;

function Rect.IsSquare : boolean;
 begin
  result := Width=Height;
 end;

begin

end.
```

Listing 2.6: Sample code in **Java**

```
public class Rect{

  private double Width;
  private double Height;
  private double Area;
  private double Perimeter;

  public Rect(double aWidth, double aHeight){
    this.Width = aWidth;
    this.Height = aHeight;
    Update();
  }

  public void Update(){
   this.Area = this.Width * this.Height;
   this.Perimeter = 2 * (this.Width + this.Height);
  }

  public boolean IsSquare() {
   return this.Width==this.Height;
  }
}
```

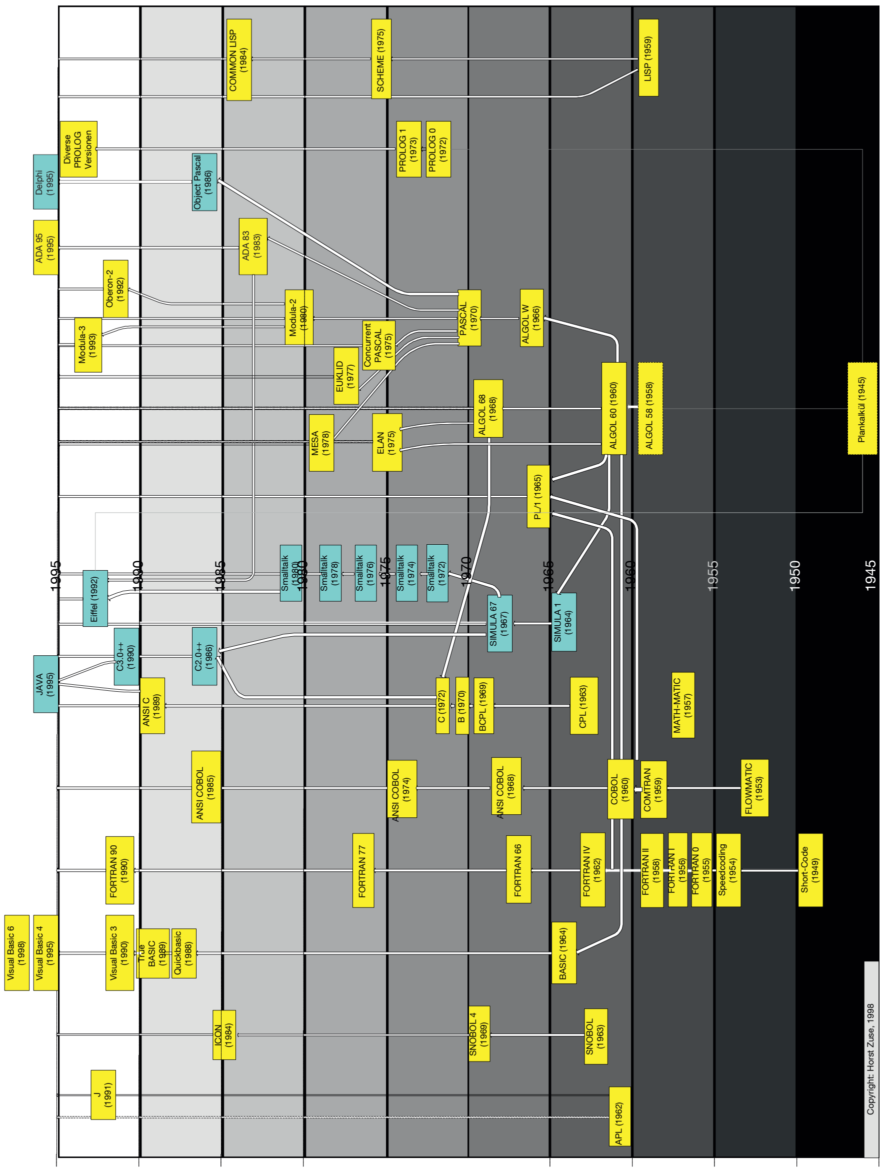


Figure 2.3: Overview of the history of programming languages (adopted from (Zuse 1999, p. 6) – languages described in this thesis are in cyan color

2.2 Different Views of Object Orientation

Obviously, object-orientation plays an important role for this thesis. Therefore, this term has to be thoroughly clarified. Many different definitions or specifications of object orientation can be found in the literature. Unfortunately, it turned out that there are several substantially different views of object orientation that cannot be integrated into one “mainstream” perception. In his overview of the history of the object-oriented approach, Capretz (2003) described several of these views:

> “To some, the concept of object was merely a new name for abstract data types; each object had its own private variables and local procedures, resulting in modularity and encapsulation. To others, classes and objects were a concrete form of type theory; in this view, each object is considered to be an element of a type which itself can be related through sub-type and supertype relationships. [...] [O]bject-oriented software systems were a way of organizing and sharing code in large software systems. Individual procedures and the data they manipulate are organized into a tree structure. Objects at any level of this tree structure inherit behavior of higher level objects; inheritance turned out to be the main structuring mechanism which made it possible for similar objects to share program code.” (Capretz 2003, p. 2)

In the following the most prominent views of object orientation are summarized. Starting with the early definitions by Dahl and Nygaard (1966), several examples from different sources are presented here to demonstrate the variety.

Dahl and Nygaard (1967) introduced the notions of classes and objects in the context of the development of Simula. They stated that the “class concept introduced is a remodeling of the record class concept proposed by Hoare. [...] A prefix notation is introduced to define subclasses organized in a hierarchical tree structure. The members of a class are called objects. Objects belonging to the same class have similar data structures” (p. 159). Additionally, they introduced the notion of inheritance on a class level and the combination of data and objects that belong to a class.

Meyer (1987*b*) pointed out an important difference between the object-oriented paradigms and other programming styles. While the latter regard data as passive, the object-oriented paradigm focuses on active objects that operate on their own data (cf. Meyer 1987*b*, p. 53).

Later, Meyer (2009) defined seven concepts that have to be implemented to fulfill the requirements of object-oriented paradigm: object-based modular structures, data abstraction, automatic memory management, classes, inheritance, polymorphism and dynamic binding, and multiple and repeated inheritance. This definition is comprehensive, although it is more a definition of object-oriented languages than of object orientation in general. Furthermore, it emphasizes the software development process as follows:

> “The object-oriented approach is ambitious: it encompasses the entire software lifecycle. When examining object-oriented solutions, you should

> check that the method and language, as well as the supporting tools, apply to analysis and design as well as implementation and maintenance." (p. 22)

Another approach for defining object orientation was conducted by Wegner (1990). His definition is built hierarchically from the concept of object, adding the concepts of class and inheritance and is based on the classification of programming languages. The first group of languages is object-based. By adding the concept class, the remaining languages are called class based. The last and most restrictive language group includes the object-oriented languages with the introduction of inheritance.

> "Object-based, class-based, and object-oriented languages are progressively smaller language classes with progressively more structured language requirements and more disciplined programming methodology. Object-based languages support the functionality of objects but not their management. Class-based languages support object management but not the management of classes. Object-oriented languages support object functionality, object management by classes, and class management by inheritance." (Wegner 1990, p. 26)

In the end it can be summarized as: *object-oriented = objects+classes+inheritance*.

A definition that is neither focused on languages nor on object in particular was the one by Blair (1991). He defined four dimensions of object-oriented systems, not only languages. They were all based on object-oriented computing as a special kind of abstraction:

> "**Encapsulation** is defined as the grouping together of various properties associated with an identifiable entity in the system in a lexical and logical unit, i.e. the object. Furthermore, access to the object should be restricted to a well-defined interface." (p. 111)
>
> "**Classification** is the ability to group associated objects according to common properties. Various classifications can be formed representing different groupings in the system. All objects within a particular grouping will share all the common properties for that grouping but may have other differences." (p. 111)
>
> "**Polymorphism** implies that objects can belong to more than one classification. Classifications can therefore overlap and intersect. Thus it is possible for two different classifications to share common behaviour." (p. 111)
>
> "**Interpretation** is defined as the resolution of polymorphism. In polymorphic environments, it is possible for a particular item of behaviour to have several different meanings depending on the context. It is therefore the task of interpretation to resolve this ambiguity and to determine the precise interpretation of an item of behaviour." (p. 112)

Another point of view was presented by Hares and Smart (1994). Object orientation was shown in the perspective of its benefits for database development and software design. The authors focused on the improvement that the new paradigm provided.

> “Object orientation is the technology that is replacing today’s database and programming technology for the design and development of computerized application systems, the technology that some regard as the ultimate paradigm for the modeling of information, be that information or logic.” (Hares and Smart 1994, p. 1)

According to a description of all major concepts related to object orientation, Hares and Smart (1994) presented a graphical overview of the concept of objects in general and information hiding or encapsulation in particular (see Figure 2.4).

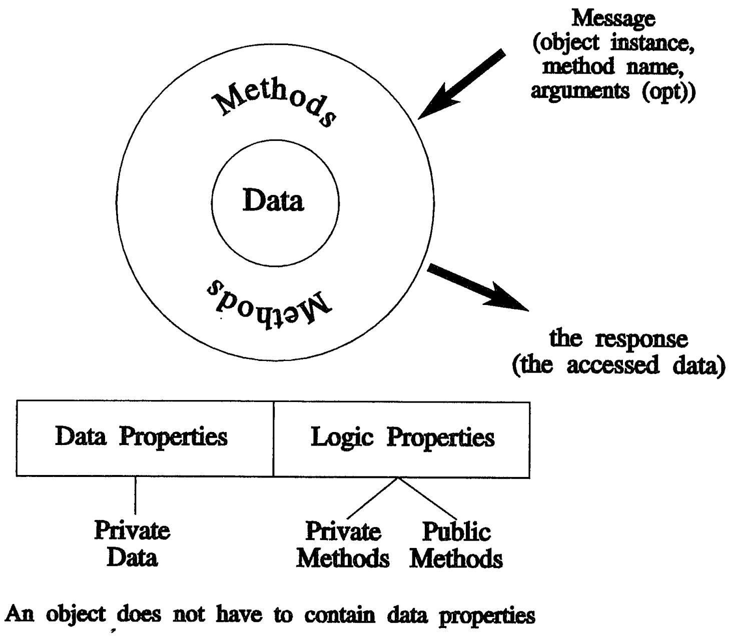


Figure 2.4: Graphical representation of the concepts object and encapsulation (Hares and Smart 1994, p. 43)

In the early 1990s, Rumbaugh (1991), Jacobson (1992), and Booch (1994) introduced object-oriented modeling techniques. Later on, these different approaches were integrated into the unified modeling language (UML). The first version[4] was published in 1997. In the context of UML the following view of object orientation was formulated:

> “In [the object-oriented] approach, the main building block of all software systems is the object or class. Simply put, an object is a thing, generally

[4]http://www.omg.org/cgi-bin/doc?ad/97-08-11 - last access: 19.02.2015

drawn from the vocabulary of the problem space or the solution space; a class is a description of a set of common objects. Every object has identity (you can name it or otherwise distinguish it from other objects), state (there's generally some data associated with it), and behavior (you can do things to the object, and it can do things to other objects, as well)." (Booch et al. 1999)

2.2.1 A Definition of Object Orientation by its Fundamental Concepts

Apparently, there is a broad variety of views of object orientation. But how should object orientation be understood in the context of this thesis? It seems natural to solve this problem by properly defining the most important concepts of object orientation. But how find these most important concepts? For this purpose, Armstrong (2006) investigated many sources of literature. She searched for the keyword "object-oriented development" and found 239 sources. 88 of those asserted that a specific set of concepts characterized the object-oriented approach. These 88 specific sets were evaluated, counting the relative frequencies of the addressed concepts. The result is displayed in Table 2.2.

It turned out that eight of these concepts were mentioned in more than 50% of the sources: *Inheritance, Object, Class, Encapsulation, Method, Message Passing, Polymorphism*, and *Abstraction*. Honoring this particular importance, she called these eight concepts the "quarks of object-oriented development" (Armstrong 2006). Concerning the particular importance of the "quarks", it seems reasonable to select these concepts as foundation of object orientation in this thesis.

The next subsections describe and define these eight "quarks". Please note that this is done in logical order, not according to the order of Armstrong (2006). The interdependencies of the concepts are discussed in Chapter 6.

Concept	Count	Pct.	Concept	Count	Pct.
Inheritance	71	81%	Object Model	4	5%
Object	69	78%	Reuse	3	3%
Class	62	71%	Cohesion	2	2%
Encapsulation	55	63%	Coupling	2	2%
Method	50	57%	Graphical	2	2%
Message Passing	49	56%	Persistence	2	2%
Polymorphism	47	53%	Composition	1	1%
Abstraction	45	51%	Concurrency	1	1%
Instantiation	31	35%	Dynamic Model	1	1%
Attribute	29	33%	Extensibility	1	1%
Information Hiding	28	32%	Framework	1	1%
Dynamic Binding	13	15%	Genericity	1	1%
Relationship	12	14%	Identifying Objects	1	1%
Interaction	10	12%	Modularization	1	1%
Class Hierarchy	9	10%	Naturalness	1	1%
Abstract Data Type	7	8%	Safe Referencing	1	1%
Object-Identity Independence	6	7%	Typing	1	1%
Collaboration	5	6%	Virtual Procedures	1	1%
Aggregation	4	5%	Visibility	1	1%
Association	4	5%			

Table 2.2: Results of the text analysis presented by Armstrong (2006)

2.2.1.1 Object

The name-giving concept of object orientation was introduced in Simula67. According to this, the idea of the concept **object** is related to a representation of the real world.

> "In effect, an object-oriented program is a description or simulation of an application. The objects in the program are the entities in the simulation." (Sethi 2003, p. 253)

The dualism of **object** as a data carrier and something that executes actions (cf. Armstrong 2006, pp. 124) is also pointed out by Blair (1991): "An object is an encapsulation of a set of operations or methods which can be invoked externally and of a state which remembers the effect of the methods" (p. 26).

More precisely, the state of an **object** is defined by the values of its attributes (cf. Christensen 2005; Rentsch 1982). Other concepts related to **objects** that can be found in literature are instance and class. Instance is a synonym and class is the higher-level structure.

2.2.1.2 Class

According to Booch et al. (1999), “a class is a description of a set of common objects”. All instances of a **class** share the same kind of data, even if they differ in their state (data values). In turn, according to Blair (1991, p. 29) “a class is a template from which objects may be created. It contains a definition of the state descriptors and methods for the object”.

Additionally, in many definitions the concept of **class** is also used as a data type for the corresponding object (cf. Christensen 2005).

> “Similar objects, objects with common properties, are grouped into a class. A class can be thought of as the type of an object.” (Sethi 2003, p. 257)

Some definitions even emphasize **class** as the most important concept in the object-oriented approach. Precisely, abstraction – introduced in Section 2.2.1.8 – is the central concept and class is the realization of it.

> “In fact, ‘object-oriented’ is really a misnomer because what we really should be talking about is ‘class-oriented,’ since the essence of the object-oriented technique is actually the class. [...] So classes and abstract data types are actually central to the object-oriented approach, the abstract data type essentially being the formal specification of the object class.” (Henderson-Sellers 1992, p. 34)

2.2.1.3 Method

The implementation of the behavior of an object is called a **method**. Nevertheless, a **method** is nothing but a procedure strictly bound to an object. According to Armstrong (2006) a **method** is “a way to access, set or manipulate an object’s information”, while according to the definition of Blair (1991), **methods** are the only possibility to change the state of an object.

2.2.1.4 Message Passing

As mentioned above, the activity and communication of and between objects is one of the fundamental idea of object orientation. This is enabled by the mechanisms of **message passing**. According to Armstrong (2006, p. 126) **message passing** is “the process by which an object sends data to another object or asks the other object to invoke a method”.

Rosson and Alpert (1990) emphasize the communication aspect of **message passing**. “Central to [object-oriented programming] OOP is the metaphor of communicating objects.[...] All computation in an OOP system takes place as a result of message sending; all objects share the ability to send and respond to message” (Rosson and Alpert 1990, p. 355).

Sethi (2003) summarizes important aspects of **message passing**. A "message to an object corresponds to a procedure call; messages can carry parameters. In response to a message, the object executes a method, which corresponds to a procedure body; it returns an optional result. The result, if any, will itself be an object of some class" (Sethi 2003, p. 258).

2.2.1.5 Encapsulation

In the literature, **encapsulation** is defined as a collection of data that is only accessible through well defined processes. Additionally, there are definitions only focusing on the grouping of data and the corresponding functionality (cf. Deitel and Deitel 2012, p. 49) and those that focus on the well defined accessibility of that data (cf. Henderson-Sellers 1992, p. 19). The balance between readability of code and the **encapsulation** of data in single classes is a central topic of this concept (cf. Eckerdal et al. 2006).

> "In any class in any language there is what is usually known as a private part and a public part - the private part is where information is hidden (that's usually the implementation of the functionality and the attributes (the variables)); and the public part is the statement, the names, of the functions that this object will respond to messages about; in other words, that can be used by a client object." (Henderson-Sellers 1992, p. 60)

Besides the **encapsulation** of data and functionality, Henderson-Sellers (1992) states that the well defined access is defined by an interface concept. So, only the necessary functionality for changing the data is provided through an interface. The data itself cannot be changed from the outside of an object (cf. Henderson-Sellers 1992, p. 60). This fact is often called information hiding and is also used as a synonym for **encapsulation**.

2.2.1.6 Polymorphism

Blair (1991) postulates **polymorphism** to be "one of the most characteristic features of object-oriented systems" (Blair 1991, p. 35).

According to Armstrong (2006), "polymorphism is defined as: the ability of different classes to respond to the same message and each implement the method appropriately" (Armstrong 2006, p. 126).

Cardelli and Wegner (1985) listed different aspects of **polymorphism**. They distinguish between universal and ad-hoc polymorphism. Universal polymorphism, as well as ad-hoc polymorphism, is again differentiated into two types. The first type is parametric and inclusion polymorphism; the second type is overloading and coercion. The definitions are summarized by Blair (1991).

Parametric polymorphism “In parametric polymorphism, a single function (coded once) will work uniformly on a range of types. It is possible that the function will operate on all types but more likely the types will be required to exhibit some common structure.” (p. 82).

Inclusion polymorphism “Inclusion polymorphism also allows a function to operate on a range of types” (p. 83). In fact, this means that any method of the superclass is applicable in the subclass. For this reason, this type of polymorphism is strongly related to inheritance (see Section 2.2.1.7).

Overloading “Overloading allows a function name to be used more than once with different types of parameter. [...] The typing information of the parameters will then be used to select the appropriate function” (p. 81).

Coercions “Languages supporting coercion have certain in-built mappings (coercions) between types. If a particular context demands one type and a different type is provided, then the language will look to see if there is an appropriate coercion” (p. 81).

2.2.1.7 Inheritance

The concept of **inheritance** plays a central role in most definitions of object orientation. It was introduced with the development of Simula67 (cf. Dahl and Nygaard 1967). According to Armstrong (2006) **inheritance** is “a mechanism that allows the data and behavior of one class [(superclass)] to be included in or used as the basis for another class [(subclass)]” (Armstrong 2006). In the subclass the original attributes and methods can be changed by overriding (cf. Sebesta et al. 2013, pp. 545). By this way, **inheritance** supports code reuse on the one hand, and on the other hand, provides the feature to build class hierarchies.

> “Inheritance classifies classes in much the [same] way [that] classes classify values. The ability to classify classes distinguishes object-oriented programming from traditional programming languages by providing greater classification power and conceptual modeling power.” (Wegner 1989, p. 26)

The question of which application is really useful is discussed by Hu (2011). A study on the difficulties concerning the application of **inheritance** was conducted by Daly et al. (1996) and showed that most problems are related to understanding program codes.

A special problem in this context is multiple inheritance. If a class is derived from more than one class and at least two superclasses provide the same data or functionality, there must be a mechanism to uniquely define which data or which functionality can be used by the subclass. There are languages that provide multiple inheritance such as C++, while others, such as Java, do not. Whether multiple inheritance is used in a language is a choice between the advantages and disadvantages and depends on the field in which the language is mainly used (cf. Sebesta et al. 2013, pp. 551).

2.2.1.8 Abstraction

The fundamental concept of **abstraction** underlying object orientation is the last “quark” of Armstrong (2006). According to her, **abstraction** is “the act of creating classes to simplify aspects of reality using distinctions inherent to the problem” (Armstrong 2006). By this way, object orientation enables us to model parts of our world in a natural way. Consequently, this real-world modeling approach was one of the central aspects that led to the paradigm shift from imperative procedural programming to object-oriented programming (cf. Quibeldey-Cirkel 1994).

Further, Armstrong (2006) summarizes:

> “Data abstraction is possible in classical development, but it is enforced in the [object-oriented] OO approach. Many authors define abstraction in a generic sense as a mechanism that allows us to represent a complex reality in terms of a simplified model so that irrelevant details can be suppressed in order to enhance understanding. Others have conceptualized abstraction as the act of removing certain distinctions between objects so that we can see commonalities.” (Armstrong 2006)

2.2.2 Taxonomies of Object Orientation

Besides her literature analysis Armstrong (2006) introduced, a new taxonomy of object orientation. It is based on the “quarks” examined from literature (see Section 2.2.1). For the taxonomy (see Table 2.3), the concepts mentioned above are grouped in two constructs: *structure* and *behavior*.

The concepts relating to structure “are focused on the relationship between classes and objects and the mechanisms that support the class/object structure. [...] In essence a class is an abstraction of an object. The class/object encapsulates data and behavior and inheritance allows the encapsulated data and behavior of one class to be based on an existing class” (p. 127).

The second construct “behavior” combines the concepts related to the communication of objects. So, “[m]essage passing is the process in which an object sends information to another object or asks the other object to invoke a method. Polymorphism enacts behavior by allowing different objects to respond to the same message and implement the appropriate method for that object” (p. 127).

Construct	Concept	Definition
Structure	Abstraction	Creating classes to simplify aspects of reality using distinctions inherent to the problem.
	Class	A description of the organization and actions shared by one or more similar objects.
	Encapsulation	Designing classes and objects to restrict access to the data and behavior by defining a limited set of messages that an object can receive.
	Inheritance	The data and behavior of one class is included in or used as the basis for another class.
	Object	An individual, identifiable item, either real or abstract, which contains data about itself and the descriptions of its manipulations of the data.
Behavior	Message Passing	An object sends data to another object or asks another object to invoke a method.
	Method	A way to access, set, or manipulate an object's information.
	Polymorphism	Different classes may respond to the same message and each implements it appropriately.

Table 2.3: Object-oriented taxonomy presented by Armstrong (2006)

In addition to the definition of a taxonomy, Armstrong (2006) cited two older object-oriented taxonomies. The taxonomy of Rosson and Alpert (1990) contains four elements that characterize the object-oriented design process. Basically, these elements are:

Communicating objects: includes the concepts of object, message passing, and method.

Abstraction: includes the concepts of information hiding, encapsulation, data abstraction, and polymorphism.

Shared behavior: includes the concepts of inheritance, class, and instance.

Designing with objects: includes the concept of object modeling.

Henderson-Sellers (1992) summarize the basis of object-oriented systems in a triangular relationship. The concepts of encapsulation and information hiding, classification and abstract data types, and polymorphism through inheritance build the nodes, and are based on the consensus of a literature review on object-orientation (cf. Henderson-Sellers 1992, p. 19). A graphical overview is presented in Figure 2.5

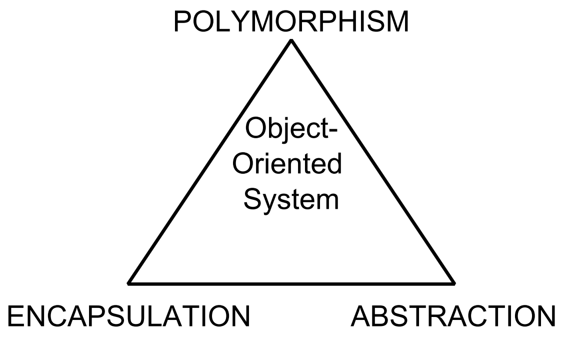
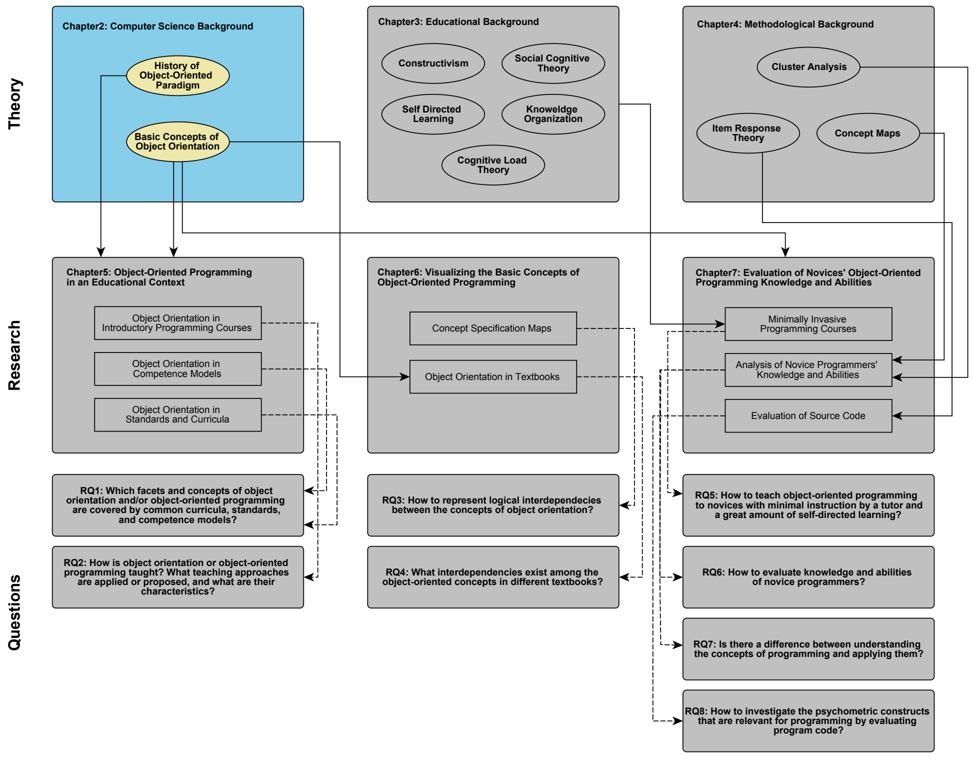


Figure 2.5: Object-oriented triangle by Henderson-Sellers (1992)

Relevance for this thesis

The experiments described in the following chapters are all related to object orientation. Chapter 5 gives an insight into the educational base (e.g., curricula and standards) of computer science. Here, the focus is on object-orientation. Although the text analysis in Chapter 6 can be applied on any topic, in this thesis it has been conducted on textbooks related to the introduction into object orientation or object-oriented programming, respectively. The experimental courses and studies on programmers in Chapter 7, are also associated with object orientation. For this reason, a summary view has been given on the development of the object-oriented programming paradigm in the last view paragraphs. In addition, the following chapters refer to the definition of object orientation, which was presented in the previous sections.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

3 Educational Background

The second chapter of the theoretical background for this thesis summarizes the educational theories that are taken into account in the experimental settings described in Chapter 7.

3.1 Constructivism

The elementary learning theory underlying the experimental courses introduced in Chapter 7 is the theory of constructivism. “Constructivism is a broad term with philosophical, learning and teaching dimensions, but it generally emphasizes the learner’s contribution to meaning and learning through both individual and social activity”(Bruning et al. 2011, p. 215). The theory is mainly based on the work of Piaget (1929) and Vygotsky (1962). Basically, there are two principles that characterize the theory, especially in contrast to the learning theories of behaviorism and cognitivism.

> “Knowledge is not passively received but actively built up by the cognizing subject;
>
> The function of cognition is adaptive and serves the organization of the experiential world, not the discovery of ontological reality” (Glasersfeld 1989*b*, p. 162).

The main assumption is that skills and knowledge of an individual are gained through interactions with people and situations (cf. Maturana and Varela 1980, p. 49). This implies that the learner is regarded as an active creator in the learning process. The role of the teacher changes to a kind of mentor who supports the learning process by providing learning materials and creating different learning situations.

According to Schunk (2011), constructivism is not a single viewpoint, but rather provides different perspectives on the acquisition of knowledge (see Table 3.1).

Similar to the conceptual change theory described in Section 3.5, the provided inputs should be in conflict to the present cognitive structures of the students to enforce accommodation and assimilation. These cognitive restructuring processes depend on equilibration, the “central factor and the motivating force behind cognitive development” (Schunk 2011, p. 236). According to Piaget (1929), every individual tries to find a state of mind where the cognitive structures are equivalent to the environment. Therefore, we can fit the environmental (external) reality to our cognitive structures (assimilation) or vice versa (accommodation) (cf. Glasersfeld 1989*a*, pp. 126).

Perspective	Premises
Exogenous	The acquisition of knowledge represents a reconstruction of the external world. The world influences beliefs through experiences, exposure to models, and teaching. Knowledge is accurate to the extent it reflects external reality.
Endogenous	Knowledge derives from previously acquired knowledge and not directly from environmental interactions. Knowledge is not a mirror of the external world; rather, it develops through cognitive abstraction.
Dialectical	Knowledge derives from interactions between persons and their environment. Constructions are not invariably tied to the external world nor wholly the workings of the mind. Rather, knowledge reflects the outcomes of mental contradictions that result from one's interactions with the environment.

Table 3.1: Overview of the different perspectives on constructivism (Schunk 2011)

As a consequence of these restructuring processes, the role of the teaching person has to change. This is taken into account in the experimental setting described in Chapter 7.

> "If, then, we come to see knowledge and competence as products of the individual's conceptual organization of the individual's experience, the teacher's role will no longer be to dispense 'truth', but rather to help and guide the student in the conceptual organization of certain areas of experience. Two things are required for the teacher to do this; on the one hand, an adequate idea of where the student is and, on the other, an adequate idea of the destination." (Glaserfeld 1983)

Another theory associated to the constructivism theory that had an influence on the experimental setting is the theory of cognitive apprenticeship. Here, students work together with experts in a authentic working situation.

> "So the term apprenticeship helps to emphasize the centrality of activity in learning and knowledge and highlights the inherently context-dependent, situated, and enculturating nature of learning." (Brown et al. 1989, p. 39)

Therefore, teachers promote the learning process first by "making explicit their tacit knowledge or by modeling their strategies for students in authentic activity. Then, teachers and colleagues support [the] students' attempts at doing the task. And finally they empower the students to continue independently" (Brown et al. 1989, p. 39). More precisely, the cognitive apprenticeship process can be divided into four phases:

> "**Modelling** involves showing an expert carrying out a task so that students can observe and build a conceptual model of the processes that are required to accomplish the task.

Coaching consists of observing students while they carry out a task and offering hints, scaffolding, feedback, modelling, reminders, and new tasks aimed at bringing their performance closer to expert performance.

Scaffolding refers to the supports the teacher provides to help the student carry out a task.

Fading consists of the gradual removal of supports until students are on their own". (Collins et al. 1989, pp. 481)

These steps were explained to the peer tutors who guided the participants through the experimental courses described in Chapter 7.

3.2 Social Cognitive Theory

Bandura (1986) investigated several social influences on the learning process. According to the "Social Learning and Imitation Theory" of Miller and Dollard (1941) he first introduces the "Social Learning Theory" (Bandura 1977). Later, Bandura (1986) revised his theory. Basically, in "the social cognitive view people are neither driven by inner forces nor automatically shaped and controlled by external stimuli. Rather, human functioning is explained in terms of a model of triadic reciprocality in which behavior, cognitive and other personal factors, and environmental events all operate as interacting determinants of each other" (Bandura 1986, p. 18). In detail, he described five capabilites that affect the learning of persons.

The **symbolizing capability** enables people to understand an manage their environment. "People process and transform passing experiences by means of verbal, imaginal and other symbols into cognitive models of reality that serve as guides‘for judgment and action. It is through symbols that people give meaning, form, and continuity to the experiences they have had. Symbols serve as the vehicle of thought" (Bandura 1989, p. 9). Additionally, symbolizing capability enables people test possible solutions in thought, rather than solve problems solely by performing actions and suffering the consequences of missteps (cf. Bandura 1989).

The **forethought capability** is the "ability to bring anticipated outcomes to bear on current activities [...]. It enables people to transcend the dictates of their immediate environment and to shape and regulate the present to fit a desired future. In regulating their behavior by outcome expectations, people adopt courses of action that are likely to produce positive outcomes and generally discard those that bring unrewarding or punishing outcomes" (Bandura 2001, p. 7).

The **vicarious capability** enables people to learn from models instead through the effects of one's actions. In fact, humans "have evolved an advanced capacity for observational learning that enables them to expand their knowledge and skills on the basis of information conveyed by modeling influences. Indeed, virtually all learning phenomena resulting from direct experience can occur vicariously by observing people's behavior and its consequences for them" (Bandura 1989, p. 21).

The **self-regulatory capability** enables people to make causal contribution to their own motivation and actions "by arranging facilitative environmental conditions, recruiting cognitive guides, and creating incentives for their own efforts" (Bandura 1986, p. 20).

Finally, the **self-reflective capability** enables "people judge the correctness of their predictive and operative thinking against the outcomes of their actions, the effects that other people's actions produce, what others believe, deductions from established knowledge and what necessarily follows from it" (Bandura 2001, p. 10).

All these capabilities enable human beings to learn on the basis of the social cognitive theory and its derivations, such as the self-directed learning. The following sections describe the self-regulatory capability in detail and the influence of the theory on self-efficacy.

3.2.1 Self-Regulation

According to Bandura (1986) self-regulation is an interaction of personal, behavioral, and environmental processes. It "refers to self-generated thoughts, feelings, and actions that are planned and cyclically adapted to the attainment of personal goals"(Zimmerman 2000*a*, p. 14). Basically, self-regulatory processes are, according to Zimmerman (2000*a*), divided into three cyclic phases. The fourthought phase involves task analysis processes and self-motivation sources. Self-efficacy beliefs (see Section 3.2.2 are central sources of motivation (cf. Bandura 1997).

As the influence of actions and behavior on the learning process is a central issue in the self-related learning theory, it is important for the learner to observe their own actions and behavior. When complex facets need to be learned, there are many different factors that compete for attention. To be successful in self-observation, it is important for individuals to be aware of what they are doing (cf. Bandura 1986, pp. 336).

The aspect of self-observation is related to the facet of self-motivation in a very strong way. Changes in actions and behavior are influenced by the amount of self-set goals that are involved. Furthermore, self-observation depends on several factors. So, there is a temporal proximity, which means that changes in behavior are more effective if they are related to the present and not to the past (cf. Zimmerman 2000*a*, pp. 26).

Additionally, feedback has to be performed informatively. If the ideas of how one is doing something are quite vague, or when there is no clear evidence of progress, self-observation is unclear.

> "Immediate self-observation provides continuing information and thus the best opportunity for self-evaluation to influence the behavior while it is still in progress." (Bandura 1986, p. 338)

In addition to the facets mentioned above, motivation for and valence of the behavior is important for the self-regulative process. Furthermore, success or positive alteration of actions and behavior is more effective than punishment or negative feedback.

Self-observation is the basis for self-directed reactions and learning processes, but without a comparative function there would be no change in actions or behavior. These comparisons are very individual and depend on personal standards, which are built by several social influences.

The comparison of other standards against an internal standard is quite difficult, as in most cases there is no accurate criterion. According to Bandura (1986) there are four types of comparisons.

> **Normative comparison** is the comparison of the own ability to regular activities or standard norms based on representative groups. They are used to determine one's relative standing. Therefore, it is important that the compared normative group is typical for the individual that is comparing themself. Atypical groups lead to misjudgment in their own abilities.
>
> **Peer group comparison** is the comparison with those who seem to have similar abilities. Here, the comparison is difficult, as the individual is choosing the level of comparison with regard to the group. For example, surpassing "those known to be of lesser ability has dubious merit. For this reason, people ordinarily choose to compare themselves with those whom they regard as similar or slightly higher in ability" (Bandura 1986, p. 347).
>
> **Self-comparison** describes the continual comparison of the current actions or behaviors to former ones. Based on this comparison, people raise their internal standards after a success and decrease them after a failure. In general, referring to the self-comparison of personal challenges rather than to social comparison with others is regarded as more positive.
>
> **Collective comparison** means that the group performance is evaluated and not the individual's performance. Their own achievement is put into relation with the group's accomplishment.

3.2.2 Self-Efficacy

Perceived self-efficacy is defined by Bandura (1986) as the judgment of one's capability to accomplish a certain level of performance. Additionally, as most outcomes have their origin in actions, there is a dependency between behavior and the expected outcome; or, with regard to thought, the outcomes that people anticipate depend on the person's judgment of how well they can perform in a given situation (cf. Bandura 1986, p. 392).

As people tend to avoid situations that exceed their abilities and rather face situations they suppose they can handle, self-efficacy has a strong influence on the development of abilities. "The efficacy judgments that are the most functional are probably those that slightly exceed what one can do at any given time" (Bandura 1986, p. 394).

According to Zimmerman (2000*b*, p. 83), the *level* of self-efficacy refers to its dependence on the difficulty of the particular task. Furthermore, *generality* pertains to the transferability of self-efficacy beliefs across activities. The *strength* of perceived efficacy is measured by the amount of one's certainty about performing a given task.

Self-efficacy also has a strong influence on the amount of time an individual will spend on a problem. If self-doubts are a problem, it is more likely that an individual will give up facing difficulties that cannot be resolved in a short time. Another aspect is the influence of self-efficacy on the judgment of failures. A person who perceives low self-efficacy, judges a failure as a deficit in their own ability, while someone with a high self-efficacy will judge it as an insufficient effort (cf. Zimmerman 2000*b*, pp. 83).

Problems related to the introduction into computer science are strongly related to the different facets of self-efficacy. Several studies have investigated the influence of self-efficacy and have addressed developing measures for it (Giannakos et al. 2012; Ramalingam et al. 2004; Compeau and Higgins 1995).

Information on self-efficacy is acquired from different sources, which are described in the following list provided by Bandura (1986, pp. 399). However, the list does not provide any assessment of a self-efficacy level.

Performance attainments: When failure is unrelated to the effort since the failure occurs quite early in the problem solving process, the degree of influence depends on the former level of self-efficacy of the individual. Enhanced self-efficacy can be generalized to other fields than where it was acquired.

Vicarious experience: If people from a peer group perform successfully, self-efficacy is increased. Again, the former level of perceived efficacy has an influence on the sensitivity of the change. Hence, the kind of external information that is used to evaluate the actions or behavior has an effect on vicarious experience.

Verbal persuasion: Telling an individual that they are able to solve a given task increases self-efficacy. Here, the discrepancy between positive effects of encouraging and the negative effects of discouraging are bigger than it is at performance attainments or vicarious experience. The effect of verbal persuasion is strongly related to motivational aspects, as “those who have been persuaded of their inefficacy tend to avoid challenging activities and give up quickly in the face of difficulties” (Bandura 1986, p. 400).

Psychological state: Here, the focus lies on the general feeling of an individual. For example, fear has a strong influence on self-efficacy. People who are afraid of a given task perform worse than those who are delight in solving the problem. Furthermore, “fear reactions generate further fear through anticipatory self-arousal” (Bandura 1986, p. 401).

Besides the self-efficacy of individuals, there is self-efficacy of groups. This collective efficacy has its roots in self-efficacy of the members of the enclosing group (cf. Bandura 1986, p. 449).

As mentioned by Zimmerman (2000*b*), self-efficacy is multidimensional. It is dependent on the topic that the efficacy belief is related to. Because of this, self-efficacy can only be expressed in relation to a specific topic or context. Furthermore, self-efficacy is related to the future and is assessed before performance.

3.3 Self-Directed Learning

According to Knowles (1975, p. 14), the theory of self-directed learning can be regarded as an agglomeration of different aspects of learning by self-instruction. Self-directed learning is based on social cognitive learning (see Section 3.2). The facets of self-observation, self-efficacy, and others mentioned above are used in this theory. This is because it follows, in general, the human psychological development. We are, at first, dependent on parents and then through growing up become independent of adult influences.

Knowles (1975) defines self-directed learning as a description of “a process in which individuals take the initiative, with or without the help of others, in diagnosing their learning needs, formulating learning goals, identifying human and material resources for learning, choosing and implementing appropriate learning strategies, and evaluating learning outcomes” (Knowles 1975, p. 18).

Despite the implication of most labels, self-direct learning is not learning in isolation. On the contrary, in most cases self-directed learning takes place in groups with the help of teachers, tutors, or other mentors (cf. Knowles 1975, p. 18).

The main difference between self-directed learning and a teacher-centered strategy is that the learner themself decides what and how topics should be learned, whereas in a teacher-centered strategy the teacher is responsible for this. More precisely, the orientation to learning a specific topic is the result of previous conditioning and is task- or problem centered. Therefore, learning experiences should be organized as projects with tasks and problems to be solved (cf. Knowles 1975, pp. 20).

Basically, as the learner is responsible for the topics and methodology of learning, the role of the teacher or tutor in self-directed learning has to be redefined. The teacher mostly assumes the function of a facilitator in the learning process. According to Knowles (1975), several areas have to be taken into account when preparing and organizing the topics. First, in a basic step the students have to understand that they are applying self-directed learning and because of that have to be aware of the basic principles of this theory. Another very basic element is the role of the teacher-student relation. As the teacher is regarded as a facilitator, the students have to be enabled to accept the situation. Furthermore, they have to be encouraged to work together in a collaborative manner.

Additionally, setting learning goals and defining a learning plan, which is a central tool in self-directed learning, have to be specified. Also contained in these areas are the processes for making unknown resources available to the students. Setting goals is an important step in self-directed learning, as the learning goals should be achieved in a constructive process.

A central purpose of the self-directed learning theory is that the learning activities are elected by the students with the availability of a teacher to help them as needed. At the end there can be an evaluation of the learning outcomes from the self-directed learning, as it is known by the teacher-centered approach (cf. Knowles 1975, pp. 34).

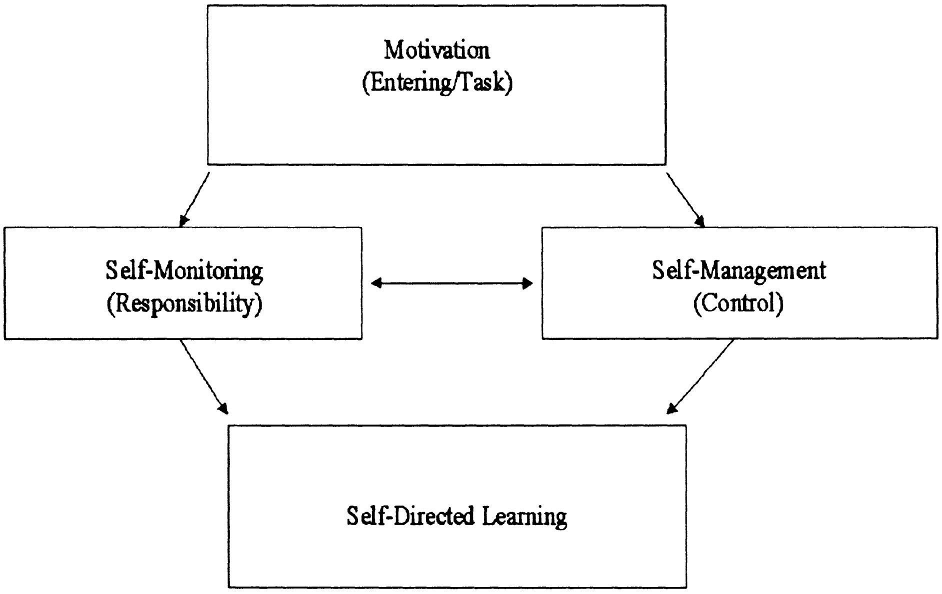


Figure 3.1: Dimensions of self-directed learning (Garrison 1997, p. 22)

In addition to the setting of the learning environment described above, Garrison (1997) defines three dimensions in self-directed learning (see Figure 3.1). Self-management includes the definition of learning outcomes, management of the learning resources, and the availability of support. It undertakes the control function in the self-directed learning process. Self-monitoring comprises the cognitive and metacognitive abilities of the learners. For successful self-directed learning the learners must be able to observe and review their own learning process. Finally, motivation is the key dimension for successful self-directed learning.

3.4 Cognitive Load Theory

The general assumptions of the cognitive load theory are based on the widely accepted ideas of human cognitive architecture (cf. Sousa 2006). Additionally, the mental representation of information as schemes builds the background for the theory (cf. Gerjets et al. 2009, p. 44; Sweller 1989, p. 458).

> "Although schemas are stored in long-term memory, in order to construct them, information must be processed in working memory. Relevant sections of the information must be extracted and manipulated in working memory before being stored in schematic form in long-term memory. The

> ease with which information may be processed in working memory is a prime concern of cognitive load theory.” (Sweller et al. 1998, p. 259)

Cognitive load is either a result of the intrinsic nature of the learning materials or its presentation. Furthermore, cognitive load theory is strongly related to instructional design.

> “Cognitive load theory is concerned with the manner in which cognitive resources are focused and used during learning and problem solving. Many learning and problem-solving procedures encouraged by instructional formats result in students engaging in cognitive activities far removed from the ostensible goals of the task. The cognitive load generated by these irrelevant activities can impede skill acquisition.” (Chandler and Sweller 1991, p. 294)

Cognitive load theory differentiates between three sorts of cognitive load: intrinsic load, germane load, and extraneous load (cf. Sweller et al. 1998, p. 259; Clark et al. 2005, p. 9) (see Table 3.2).

According to the cognitive load theory, the three sorts of load are additive. Clark et al. (2005) recommend a balancing of the different loads through instructional design. When adopting instructional designs the balance of extraneous and germane loads are important as an increase in the extraneous load leads, in most cases, to a decrease in the germane load (cf. Sweller et al. 1998, p. 259).

The basic idea underlying the cognitive load theory is the strong influence of the instructional material on the learning process. The physical barriers that limit the learners’ cognitive processes are faced by avoiding extraneous cognitive load (cf. Gerjets et al. 2009, p. 43). The introduction of worked examples is proposed as a solution to address this problem (cf. Sweller 1989, p. 463). Furthermore, the importance of avoiding mutually referring information in instructional materials is emphasized, because it splits the attention of the learner (cf. Chandler and Sweller 1991, p. 296). Combining information in different representations in the learning material is the main source of problems as it produces cognitive load that is irrelevant for schema building. The more the relevant information is presented in one source, the easier it is learned (cf. Sweller 1989, pp. 464).

The measurement of cognitive load is very difficult as cognitive processes cannot be measured in a direct way. Nevertheless, Sweller (1988, p. 272) proposes measuring cognitive load through computational models of problem solving strategies. These models imply different types of measure. On the one hand, there are the number of statements in the working memory. On the other hand, Sweller (1988) defines a production system in a computational model as “a set of inference rules that have conditions for applications and actions to be taken if the conditions are satisfied” (p. 264). The number of productions, the number of cycles needed to solve the problem, and the total number of conditions are introduced as a measure for cognitive load.

Load type	Source	Cognitive processes	Effect on learning
Intrinsic CL	Domain complexity (element interactivity) x prior knowledge	Necessary to hold interacting elements active in working memory in parallel	Harmful in that a too high intrinsic CL may cause cognitive overload
Extraneous CL	Poor instructional design	Irrelevant to schema construction and automation	Harmful, ineffective
Germane CL	Supportive instructional design	Relevant to schema construction and automation	Helpful, effective

Table 3.2: Overview of the different levels of cognitive load (CL) theory (Gerjets et al. 2009, p. 43)

3.5 Knowledge Organisation

Anderson (2005) described conceptual knowledge in the following way.

> “When we store an experience in memory, we do not record every detail, as a physical recording would. We keep some of the information and drop other details. [...] Other abstractions are possible. For instance, we can abstract from specific experiences to general categorizations of the properties of that class of experiences. This sort of abstraction creates conceptual knowledge involving categories.” (p. 154)

Another definition of conceptual knowledge was given by Anderson and Krathwohl (2009). They distinguished two parts of declarative knowledge: factual and conceptual knowledge. They have “reserved the term *Factual Knowledge* for the knowledge of discrete, isolated ‘bits of information’ and the term *Conceptual Knowledge* for more complex, organized knowledge forms” (p. 42). Complementary, they introduced two additional kind of knowledge: procedural and metacognitive knowledge. According to them, procedural knowledge “is the ‘knowledge of how’ to do something” (p. 52) and metacognitive knowledge “is knowledge about cognition in general as well as awareness of and knowledge about one’s own cognition” (p. 55).

A comparison of the two different definitions by Anderson (2005) and Anderson and Krathwohl (2009) shows that factual knowledge can be represented by propositions, conceptual knowledge by propositional networks, semantic networks or schemata. Here, “[p]ropositions represent the atomic units of meaning and can be used to encode the meaning of sentences and pictures” (Anderson 2005, p. 169).

The theory of conceptual change is based on two different views of knowledge representation. Özdemir and Clark (2007) give an overview of the different theories, focusing on the two representation perspectives *“knowledge as theory”* and *“knowledge as element”*.

3.5.1 Knowledge as Theory

The perspective “knowledge as theory” is based on the notions of accommodation and assimilation (Piaget 1952) and Kuhn’s paradigm shift in science (Kuhn 1996). Assimilation “incorporates all the given data of experience within the [mental] framework” (Piaget 1952, p. 6). In contrast, accommodation means the transformation of the mental framework (cf. Piaget 1952, p. 7). As mentioned in Section 2.1, the paradigm shift theory of Kuhn (1996) assumes that a given mental framework is only replaced if there is a need for it, which means that new ideas cannot be integrated into the present framework. So, the perspective “knowledge as theory” assumes that if “a learner’s current conception is functional and if the learner can solve problems within the existing conceptual schema, then the learner does not feel a need to change the current conception. Even when the current conception does not successfully solve some problems, the learner may make only moderate changes to his or her conceptions. [...] In such cases, the assimilations go on without any need for accommodation. It is believed that the learner must be dissatisfied with an initial conception in order to abandon it and accept a scientific conception for successful conceptual change" (Özdemir and Clark 2007, p. 352).

The changing process can take place in two different modes. The first one is called *weak restructuring* (Carey 1985, pp. 186) or *conceptual capture* (Hewson 1981), depending on the literature. This mode is applied if the learner is able to solve some problems with his given theory, while some other problems cannot be solved with his present knowledge. Because of that, only some moderate changes are applied. The second mode is called *strong restructuring* (Carey 1985, pp. 186) or *conceptual exchange* (Hewson 1981), depending on the literature. The learner applies this mode if there is a dissatisfaction with his present conceptual knowledge on the given topic, especially if the learner is not able to solve the given problems they face.

Posner et al. (1982) provide four criteria that need to be fulfilled to make conceptual change possible.

1. "*There must be dissatisfaction with existing conceptions.* Scientists and students are unlikely to make major changes in their concepts until they believe that less radical changes will not work. Thus, before an accommodation will occur, it is reasonable to suppose that an individual must have collected a store of unsolved puzzles or anomalies and lost faith in the capacity of the current concepts to solve these problems" (p. 214).
2. "*A new conception must be intelligible.* The individual must be able to grasp how experience can be structured by a new concept sufficiently to explore the possibilities inherent in it" (p. 214).
3. "*A new conception must appear initially plausible.* Any new concept adopted must at least appear to have the capacity to solve the problems generated by its predecessors. Otherwise it will not appear a plausible choice. Plausibility is also a result of consistency of the concepts with other knowledge" (p. 214).
4. "*A new concept should suggest the possibility of a fruitful research program.* It should have the potential to be extended, to open up new areas of inquiry" (p. 214).

The above mentioned idea of conceptual change has a strong influence on misconceptions in learning.

> "Misconceptions are [...] only inaccurate beliefs; misconceptions organize and constrain learning in a manner similar to paradigms in science. In other words, prior conceptions are highly resistant to change because concepts are not independent from the cognitive artifacts within a learners' conceptual ecology." (Özdemir and Clark 2007, p. 352)

Furthermore, Posner et al. (1982) suggest that the conceptual ecology, the current concepts of an individual, has a strong influence on the accommodation process during conceptual change. They distinguish between different determinants of the accommodation's direction. First, failures in concepts, called anomalies, are determinants of accommodation. Analogies and metaphors can suggest new ideas, while epistemological commitments such as explanatory ideals or general views emphasize the accommodation. Last, metaphysical beliefs and concepts and connections to knowledge in other fields determine the accommodation (cf. Posner et al. 1982, pp. 214).

Another notion of cognitive processes related to knowledge change has been discussed by Özdemir and Clark (2007). The assimilation and accommodation process during conceptual change can appear in different forms. If the old and the new concept are fundamentally different, a replacement process takes place. Additionally, conceptual change takes place by splitting the old concept into several new concepts or integrate two old concepts into one new concept. These processes can be observed especially in investigations on children's knowledge structures (cf. Özdemir and Clark 2007).

According to the knowledge-as-theory perspective, even children form such theories of knowledge. As the main focus of this thesis is the investigation of novice programmers' knowledge and ability, the knowledge structure of novices in general is of special interest. Each novice owns their own "theory" that comes from daily experiences. These conceptions are the base for any learning process the novices are involved in (cf. Özdemir and Clark 2007, p. 354).

Nevertheless, the intention of the learner towards the learning content is important for enabling a change in the cognitive processes (cf. Duit and Treagust 2003, p. 672). As Duit and Treagust (2003) emphasize, conceptual change does not mean any exchange of given knowledge structures, but rather is a restructuring of knowledge during the learning process of a specific concept. In most cases the teacher tries to find a teaching approach that does not fit the learner's conceptual model. The resulting dissatisfaction leads to a refinement or assimilation of the new model. Some studies have shown that the old conceptual model does not disappear completely, but stays active in some context and is, therefore, a source of misconceptions (cf. Duit and Treagust 2003, p. 673).

3.5.2 Knowledge as Elements

The perspective "knowledge as elements" is also common in conceptual change theory, but far less influential. The students' understanding is regarded as a collection of independent elements. These elements are built from daily experiences and interactions with the real world. During the learning process the loose connections between the elements are revised and new elements are connected to former ones. From this perspective conceptual change is an evolutionary process rather than a replacement process as in the knowledge-as-theory perspective (cf. Özdemir and Clark 2007).

According to diSessa (1993), knowledge elements are called phenomenological primitives (p-prims). Each p-prim is self-explanatory. As the name implies, they are based on real world interpretations.

> “Roughly speaking, p-prims are about the ‘size’ and complexity of words, although in several senses they are clearly smaller and simpler than words. In the first sense, lexical items often have clusters of meanings; [...] P-prims are, by contrast, more comparable to a single sense of a word; they are the smallest, context-invariant mental activations.” (p. 191)

The evolutionary process mentioned above is explained by the change in functionality of a p-prim. While they first have to be self-explanatory, later on they have to “defer to much more complex knowledge structures” (p. 115). It is possible that a p-prim refers to various phenomena. During the process of gaining expertise, the priorities of the p-prims in a knowledge cluster change.

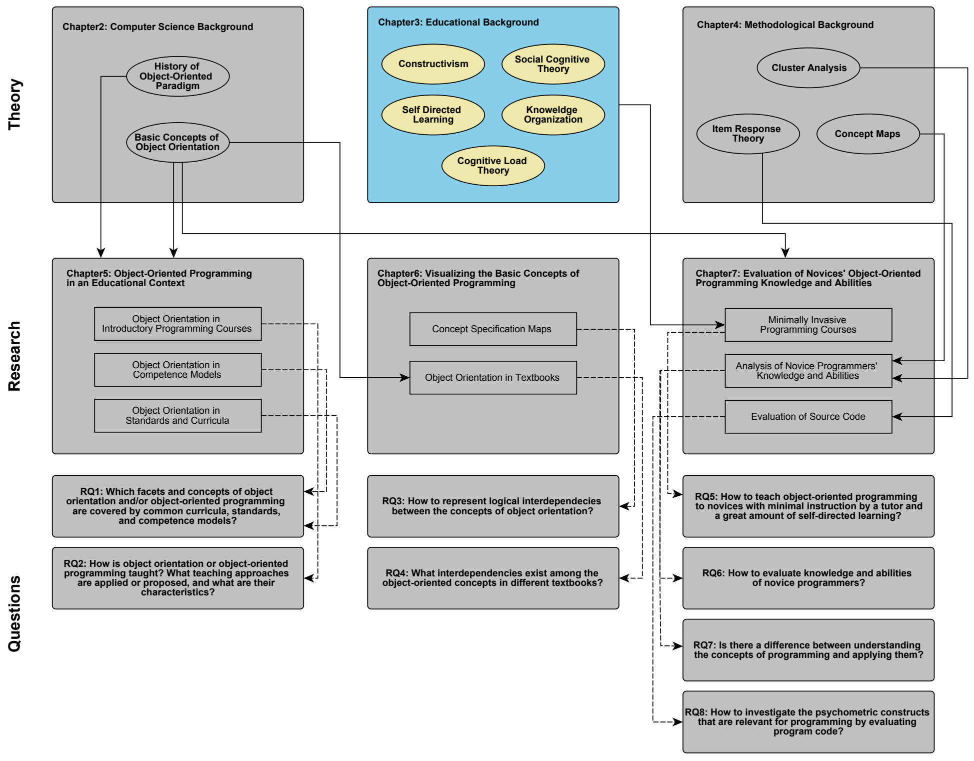
> “Following the theory sketch, I describe these changes as shifting priorities, which may gradually relocate a knowledge element within the knowledge system.” (p. 142)

Based on this theory, misconceptions are seen as individual components that have to be revised and reconnected in a more proper way. In accordance with the knowledge-as-theory perspective, naive knowledge is regarded as highly resistant to change, which is important in novices’ education (cf. Özdemir and Clark 2007, pp. 354).

Relevance for this thesis

The investigations presented in Chapter 7 are undertaken in the context of an introductory programming course. For that reason several preliminary assumptions were presented in the Sections 3.1-3.3. The educational background of the courses is based on the notions of constructivism (Section 3.1), social cognitive theory (Section 3.2), and self-directed learning (Section 3.3). Furthermore, the notions of cognitive load theory (Section 3.4) were considered during the development of the course materials (Section 7.3).

The investigations on knowledge (Section 7.5.2) and misconceptions (Section 7.5.3) require an understanding of knowledge organization (Section 3.5). For example, the resistance of novices' prior knowledge to change is a serious problem in education. Instruction often can only achieve a replacement of former knowledge in a special context. To perform conceptual change in a broad sense, students have to be enabled to reorganize their knowledge (cf. Özdemir and Clark 2007).


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knowledge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

4 Methodological Background

This chapter focuses on the methods needed for the data analysis in this thesis. Cluster analysis is applied to find similar and/or different groups (Sections 7.5.1-7.5.2). Item response theory is applied to evaluate the students' program code that was produced during the experimental courses. In this section only the theoretical background of the methodologies is presented. The application of the presented methodologies on the data is shown in the corresponding sections in Chapter 7. Before introducing the analysis methodologies, the theory underlying the concept maps is introduced. This technique allows representation of the students' knowledge structures and is applied in Chapter 7.

4.1 Concept Maps

In addition to the products of programming, the differences between knowledge of programming concepts and their application in the program code are investigated (see Section 7.5.4). For that reason, a technique that represents knowledge in an appropriate way was needed. One quite elaborate technique is the drawing of concept maps, which has been investigated in many different fields of research.

> "Concept mapping is a graphic technique for representing ideas, helping to think, solving a problem, planning a strategy or developing a process. Concept mapping means connecting different concepts of the subject and constructing relations by compiling the map." (Dahncke and Reiska 2008, p. 1)

4.1.1 Principles Underlying Concept Maps

In many subject domains concept maps have long been established as a teaching and assessment tool. Going back to Novak and Ausubel's theory of meaningful learning (cf. Novak 2002; Novak and Cañas 2008), there have been many alterations to the original concept mapping technique. For example, in contrast to the original description of a "good" concept map, a concept map does not usually need to be hierarchically organized anymore (cf. Ruiz-Primo et al. 1998).

Originally, concept maps were applied to express cognitive knowledge structures and especially conceptual change (see Section 3.5 and Novak and Cañas 2008, p. 3). In contrast, this technique is also valuable in the context of competencies.

"Concept interrelatedness is an essential property of knowledge, and one aspect of competence in a domain is well structured knowledge. A potential instrument to capture important aspects of this interrelatedness between concepts is concept maps." (Ruiz-Primo 2000, p. 31)

Essentially, concept maps are labeled, directed graphs with nodes representing concepts and edges symbolizing associations between these concepts. The labels of two incident nodes together with the label of the connecting edge form an association – the basic element of a concept map. For example, "Concept Maps - represent - Organized Knowledge". The comprehensive concept map in Figure 4.1 illustrates the structure of concept maps in general.

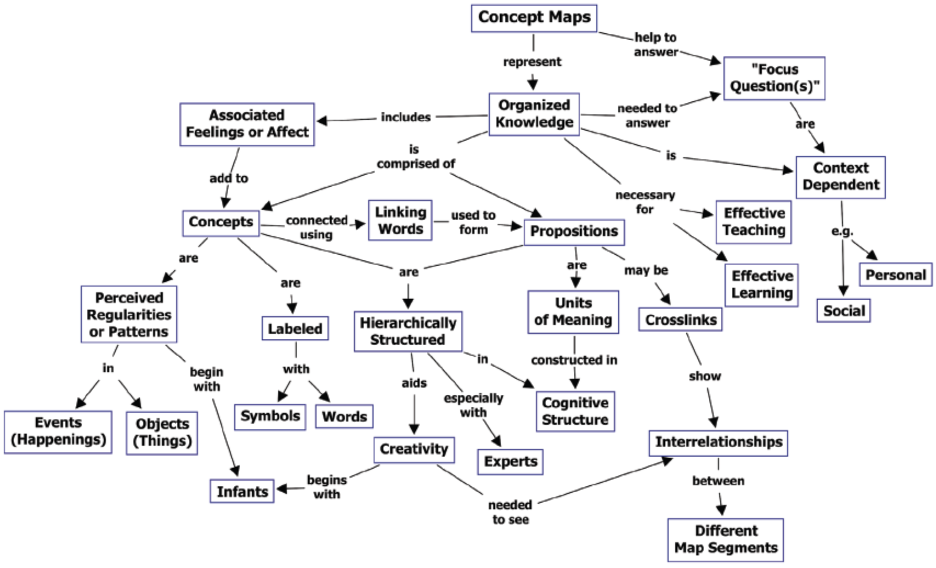


Figure 4.1: Comprehensive concept map that describes the structure of concept maps (Novak and Cañas 2008, p. 2)

As the central element of a concept map is a concept, this term has to be defined in a proper way. Novak and Cañas (2008) "define concept as a perceived regularity in events or objects, or records of events or objects, designated by a label. The label for most concepts is a word, although sometimes we use symbols such as + or %, and sometimes more than one word is used" (p. 1).

Additionally, the other main element in a concept map, the proposition of two or more concepts, is defined by Novak and Cañas (2008) as "statements about some object or event in the universe, either naturally occurring or constructed. Propositions contain two or more concepts connected using linking words or phrases to form a meaningful statement. Sometimes these are called semantic units, or units of meaning" (p. 1).

To stimulate the construction of a map on a specific topic, a focus question is usually posed. This question helps participants focus on the main purpose of the concept map that is being drawn. Additionally, concepts can be proposed with or without any initial connections. Alternatively, maps are created from scratch. Each method provides advantages and disadvantages and has to be chosen with regard to the research goal. Besides the focus question, it is important to define a context for the concept map because conceptual structure that should be represented by the map is strongly dependent on a given context. Otherwise, the resulting map can represent a concept with the same label but a different meaning (cf. Novak and Cañas 2008, p. 11).

In this thesis concept maps are applied to compare the declarative knowledge and its application in a practical task (see Sections 7.5.3-7.5.4). Generally, the relationship between concept maps and application is investigated by Edmondson (2005).

> "The quality of students' propositional knowledge relates directly to their ability to apply that knowledge to problems presented in a classroom and beyond." (p. 30)

4.1.2 Application of Concept Maps

> "Concept mapping may be used to identify conceptual understanding of programming concepts. Also concept mapping can be used to represent understanding of an area of knowledge. It can be used as a planning tool and assessment tool for the teacher and as a means of collaborative sharing of knowledge." (Jakovljevic 2003, p. 311)

When considering concept maps as an assessment tool, the reliability and validity aspects have to be taken into account. Regarding the reliability of concept maps, according to Albert and Steiner (2005), all maps constructed by a person should represent the same model of knowledge every time. Additionally, there are reliability concerns about the scoring and evaluation of concept maps. A proper measure for this purpose is the coefficients of intercoder reliability such as those proposed by Mayring (2010) and Krippendorff (2004). Besides the reliability of the scoring, the validity of the assessment has to be proven for each task. Albert and Steiner (2005) distinguish two sorts of validity: content and application validity. Content validity can be checked by using empirically collected maps. For comparing of the collected maps, scoring methods such as those described by Ruiz-Primo et al. (1998) can be used. The application validity can be proven by comparing the concept maps of a given task with another observation that is independent from the concept mapping task (cf. Albert and Steiner 2005, pp. 3).

Furthermore, the explicit teaching of concept mapping techniques is emphasized in literature. The participants of any concept mapping task have to learn how to draw concept maps in advance (cf. Ozdemir 2005, p. 142; Ruiz-Primo et al. 1998, p. 14). A precise strategy for introducing concept mapping is presented by Novak and Gowin (1989, pp. 24).

However, the assessment itself can consist of various student activities: The student can be asked to draw a concept map or to complete a given basic map.

> "There is a variety of ways such maps may be produced. For instance, a map may be constructed by the evaluator based on student responses to an activity such as an interview or a word association task. Alternatively, students may be asked to construct a concept map themselves using pencil and paper." (McClure et al. 1999, p. 477)

The directedness of a concept mapping task expresses the degree of freedom the participant who creates the map has in choosing concepts and relations. When constructing a map, the list of concepts and/or the list of possible edge labels can be restricted or not. An overview of the various tasks related to the directedness (cf. Ruiz-Primo 2000, p. 34; Ruiz-Primo et al. 2001, pp. 261) can be seen in Figure 4.2. A comparison of different methods can be found in the work of Ruiz-Primo (Ruiz-Primo and Shavelson 1996; Ruiz-Primo et al. 2001; Yin et al. 2005).

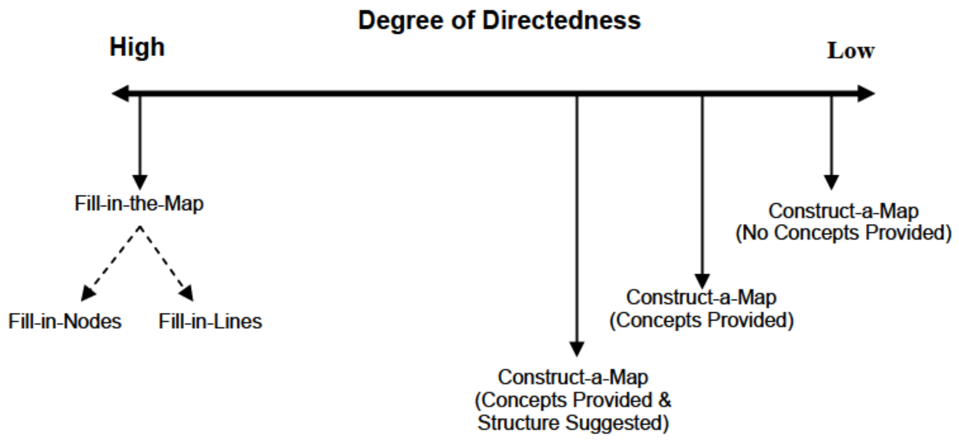

Degree of Directedness
High
Low
Fill-in-the-Map
Fill-in-Nodes
Fill-in-Lines
Construct-a-Map (Concepts Provided & Structure Suggested)
Construct-a-Map (Concepts Provided)
Construct-a-Map (No Concepts Provided)

Figure 4.2: Various concept mapping tasks and their degree of directedness (Ruiz-Primo 2000, p. 35)

Concerning those methods, there are also many other useful measures that can be found in literature; for example, Shavelson and Ruiz-Primo 2005; Albert and Steiner 2005; Goldsmith and Davenport 1990. For the assessment process, there are different suitable evaluation methods.

Intuitive: The concept maps are only evaluated on an intuitive level with regard to the number of concepts or the structure.

> "Intuitive evaluation is suitable for giving advice on learning with the aid of concept maps. The advisor or researcher is able to view the maps of the subjects and, on the basis of intuitive impressions (size, structure and correctness of individual propositions), evaluate the range of the subject's knowledge. An intuitive evaluation can only be performed with small, clear concept maps. This kind of evaluation is not very suitable for comparative studies because the results are only intuitive and depend to a large extent on the respective interviewer (advisor)." (Dahncke and Reiska 2008, p. 2)

Semi-quantitative: For evaluation the propositions can be rated and the valid ones are counted. Nevertheless, in this method all information is manually evaluated.

> "Semi-quantitative evaluation can be used for assessing small concept maps and for small numbers of maps. With this type of evaluation, several simple variables are calculated (number of all the propositions, number of correct propositions). Since everything has to be evaluated manually, this type of evaluation is very time-consuming and therefore not suitable for evaluating large numbers of concept maps." (ibid.)

Computer-aided quantitative: If concept maps are digitalized, a computer-aided quantitative evaluation is possible. The criteria stay the same, but the evaluation process can be automated.

> "Computer-aided quantitative evaluation can also be used for large maps and large numbers of maps. Before evaluation can be carried out on a computer, the information from the concept maps must be entered into the computer. Special computer programs can be used for this in order to reduce the time involved and input errors, yet it is still too time-consuming for using in everyday school life. Computer-aided quantitative evaluation is very suitable for using in research." (ibid.)

Quantitative by computer: For quantitative evaluation of the concept maps, the concept maps need to be constructed in the computer itself. It is well suited for large assessment. Nevertheless, computer assessment of concept maps needs suitable support by programs that enable the user to gather the assessment maps, as well as analyze the given maps. The editor CoMapEd[5] and CoMaTo[6], an analysis package for GNU R, provide the necessary functionality (cf. Mühling and Hubwieser 2012; Mühling 2014).

> "Quantitative evaluation by computer is the most efficient type of evaluation. This type of evaluation can, however, only be carried out if the concept maps are created on the computer itself. This type of evaluation rules out human input errors. It is suitable for using with a

[5] http://ddi.in.tum.de/comaped - last access 09.12.2014

[6] http://cran.r-project.org/web/packages/comato/index.html - last access 09.12.2014

large number of maps and comprehensive maps as well. It is the only type of evaluation which, apart from using in research, is also suitable for using in schools as far as the time factor is concerned." (Dahncke and Reiska 2008, p. 2)

In addition to these very general methods for concept map analysis, single concept maps can be scored straightforward. For the *holistic scoring*, an expert is asked to rate the complete map within a given range of values considering the overall understanding of the person creating the map. Again, the common problems such as reliability and validity that are associated with a scoring process have to be taken into account.

The *structural scoring* method rates the structure of the complete map, which should be hierarchical. But, in the meantime the constraint of a hierarchical structure has been proven to be ambiguous (cf. Ruiz-Primo et al. 1998). So, other structural purposes have to be scored. For example, pre-defined structural elements containing specific concepts can be scored higher than others. A study detecting those elements was conducted by Hubwieser and Mühling (2011*b*). The *concept relations method* simply scores the relations on given criteria. For example, correct relations are rated with 1, wrong with 0, and those that are neither correct nor wrong with 0.5. Hubwieser and Mühling (2011*c*) present a study that applies this scoring technique.

The similarity scoring is a quantitative scoring method that uses similarity measures for graphs in general calculates an index for the similarity of a given map to a master map that was created by an expert in advance. For example, similarity can be calculated with

$$\frac{1}{n}\sum_{i=1}^{n}|\frac{(S_i \cap T_i)}{(S_i \cup T_i)}|,$$

where S is the set of concepts related to a given concept i in the investigated map and T is the set of concepts that are related to the corresponding concept in the master map (cf. Keppens and Hay 2008).

Another quantitative methodology is proposed by Ifenthaler (2006). By using concept maps, he developed methodology to analyze mental models and their externalization. SMD technology (structure, matching, deep structure) is based on analyzing methods of the general graph theory. For the analysis, externalizations of the participants' mental models are evaluated concerning different criteria.

- The **surface structure** is defined for the qualitative and quantitative analysis of the individual mental models. The qualitative analysis is based on a simple visual comparison of the externalizations (e.g., concept maps). For the quantitative analysis of each map, the number of propositions is calculated. These numbers are compared among all individuals.

- For the **matching structure** structural indices of the graph theory are used for classifying the mental models (i.e., the diameter or the complexity of a graph). The matching structure is calculated as the shortest way between the most distant nodes on the graph.

- The **deep structure** describes semantic similarity. Therefore, the investigated representation of the mental model is compared to a reference model by the similarity measure of Tversky (cf. Ifenthaler 2006, pp. 45; Ifenthaler 2010).

Another qualitative method concerning the structure of maps classifies the structure into several categories such as spokes, chains, and nets. Again, the structure gives a hint about the grade of integration of the expressed knowledge in the mental model of the participant. Yin et al. (2005) extend the original types for their study.

A brief description and investigation on the validity, reliability, and efficiency of the described methods can be found in publications by Keppens and Hay (2008); McClure et al. (1999); Ruiz-Primo and Shavelson (1996); Shavelson and Ruiz-Primo (2005).

In addition to the discussed application of concept maps for assessment, Kern and Crippen (2008) used the mapping technique as an evaluation in the development of scientific understanding. They conducted a longitudinal study with their students to show the assimilation and accommodation processes during their lessons.

> "In a relatively short period of time, teachers can glean the following by viewing student concept maps: prior knowledge, misconceptions, and the acquisition and accommodation of new knowledge as maps are modified over time." (Kern and Crippen 2008, p. 33)

4.2 Cluster Analysis

A common method for finding homogeneous groups in data is cluster analysis. This requires a vector of the concept score results to be built for each individual. The cluster analysis is based on a measure of how different or similar these vectors are. More precisely, the differences between the objects of one cluster are as small as possible, while the differences between objects of different clusters are as big as possible. The distances are calculated according to a specific distance measure (cf. Bortz and Schuster 2010, p. 453).

There are several distance metrics for calculating the differences or the similarities. Only some of them are appropriate for all scale levels. For dichotomous data, Bortz and Schuster (2010) list different metrics. First they introduce a similarity coefficient S, which can be calculated directly from the contingency table. Another coefficient includes the similarity in the negative or 0-rated values. The values of the simple matching coefficient (SMC) again reach from 0 to 1.

In addition to the similarity coefficients for nominally scaled variables, a coefficient is introduced to handle interval scaled variables. For these variables the Euclidean metric can be calculated (cf. Bortz and Schuster 2010, pp. 454).

Ordinal scaled variables are difficult to handle. Some coefficients need the data to be re-scaled or treated by using rang correlation coefficients. Another option is the coefficient of Gower, which is described below.

Gower (1971) introduced a general coefficient for similarity. Therefore, a definition of similarity is given.

> "Two individuals i and j may be compared on a character k and assigned a score s_{ijk}, zero when i and j are considered different and a positive fraction, or unity, when they have some degree of agreement or similarity." (Gower 1971, p. 858)

The calculation of s_{ijk} is described for several data types that are interpreted as characters. First, Gower (1971) presents a quantity called δ_{ijk} that expresses whether a character k can be compared for two objects i and j. It is equal to 1 if a comparison is possible; 0 otherwise. Accordingly, treatment of non-existing values is possible. For all compared characters there is the following formula expressing the similarity coefficient:

$$S_{ij} = \frac{\sum_{k=1}^{\nu} s_{ijk}\delta_{ijk}}{\sum_{k=1}^{\nu} \delta_{ijk}}$$

The s_{ijk} is calculated depending on the type of character. For dichotomous data it equals 1 if the character is present in both objects i and j, otherwise 0. If the character is absent in both objects, the value of δ_{ijk} is set to 0 and by definition the value of s_{ijk} is also set to 0. For qualitative characters (multiple levels, not ordered), s_{ijk} is set to 1 if the two object i and j are equal in the kth character; otherwise 0. The last assignment of s_{ijk} concerns quantitative characters (multiple levels, ordered). Here, the characters k have values x_1 to x_n for a sample with n individuals. s_{ijk} is set to $\frac{1-|x_i-x_j|}{R_k}$ where R_k is the range of k either in the total population or only in the sample. If x_i and x_j are equal, s_{ijk} is set to 1. If they are at the opposite sides of the range, it is set to 0. For values in between, s_{ijk} is a fraction. For all three types, the coefficient values range between 0 and 1, where 1 means equality and 0 means maximal difference (cf. Gower 1971, pp. 860).

In Section 7.5.1, cluster analysis is applied on the program code that was produced by novice programmers in an experimental introductory course. The clusters are calculated with the GNU R software[7]. For the analysis methods described in the next two sections, there are two packages that are either included (*stats*) or can be integrated into the software (*cluster* described by Maechler (2013)).

4.2.1 Hierarchical Cluster Analysis

For hierarchical cluster analysis, a hierarchy of clusters is built either by starting from one cluster for each case and combining similar or different clusters into a bigger one (agglomerative), or by starting with one cluster for all cases and dividing based on criteria of similarity or difference (divisive). A disadvantage of this methodology is the fact that the assignment of an item to a cluster is fixed during the process. Once the item is assigned it can only be combined with other items to form a new cluster; but, the old assignment is still valid. For this reason it is recommended that the results of a

[7]GNU R project: http://www.r-project.org - last access 10.12.2014

hierarchical cluster analysis be proven or refined by using a non-hierarchical cluster analysis as described in the next subsection.

The fusion of the clusters can be conducted by several different strategies. **Single Linkage** selects two clusters (A and B) where the distance D between all object pairs is minimal. Each distance d between the objects is calculated pairwise. The method is valid for all distance metrics. A disadvantage of this method is the possible appearance of chaining effects when combining the clusters.

Complete linkage considers the furthest neighbors (max distance D) of the objects of a pair of clusters (A, B). Accordingly, the two clusters are combined where this maximal distance is minimal. As with the single linkage criterion, here all distance metrics can be used.

$$D(A,B) = \max_{a \in A, b \in B} \{d(a,b)\}$$

Average linkage is a strategy that is located between the complete and single linkage strategies concerning the clustering criteria. For this criterion the average distances of all objects of the clusters (A, B) are calculated pairwise. The two clusters with the lowest average distance are combined. According to the calculation of an average value the criterion is only appropriate for metrics that allow the building of average values.

The clustering process can be represented graphically by a dendrogram. A dendrogram is a tree diagram that displays the scale of distances and the objects. Two objects or groups of objects are bound together if they build a cluster. Each change of the cluster is shown by a new connection of objects and clusters. The dendrogram can be drawn in either a vertical or horizontal direction (cf. Bartholomew 2008, pp. 25). Figure 4.3 shows on the left side a table of sample data and on the right side the corresponding dendrogram. The two clusters (1, 2, 3) and (4, 5, 6, 7) are obvious in the data.

	1	2	3	4	5	6	7
1	–						
2	1.5	–					
3	2	2.5	–				
4	10	9	11	–			
5	9	13	12	3.5	–		
6	14	10	11	4	4.75	–	
7	13	12	11	4.5	4.25	3.75	–

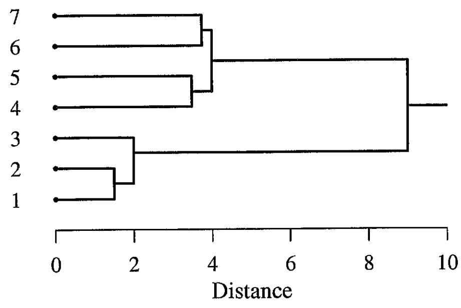


Figure 4.3: Sample data with the corresponding dendrogram (Bartholomew 2008, p. 27)

4.2.2 Partitioning Cluster Analysis

In contrast to the hierarchical cluster analysis, in partitioning cluster analysis an initial clustering is defined and then improved upon by changing the items of the clusters. For each step the costs concerning the distance metric are calculated. The step with the lowest cost is the basis for the next step. If no more changes can be made, the algorithm finishes. This strategy leads to a fix and a priori defined number of clusters. Nevertheless, the number of clusters is limited by the fact that each cluster must contain at least one element. Additionally, each element belongs to one cluster (cf. Kaufman and Rousseeuw 1990).

For the clusters, the algorithms tend to find the optimal partition of the data. As this problem is again very intensive regarding the resources, heuristical algorithms are used in most cases. Another difficulty with regard to the starting partition is that it has an influence on the final separation of the data. Because of this, the selection of the starting partition has to be plausible. Often, a hierarchical cluster analysis is first conducted on the data, with the partitioning analysis being used only for revising the results of the first analysis (cf. Bortz and Schuster 2010, p. 461). Alternatively, there are several formal techniques for detecting the optimal number of clusters in advance (cf. Everitt 2011, p. 126).

A general procedure for partitioning clustering is expressed by the “hill-climbing” algorithm introduced by Rubin (1967).

- Choose a proper starting partition with k clusters.
- Calculate the cluster criterion.
- Shift each person into a different cluster and re-calculate the partitioning criterion. The person stays in the new cluster if there is a maximum increase.
- Repeat steps two and three until there is no change in the partitioning criterion.

For detecting local maximums and minimums within the partitioning criterion, workarounds with different starting partitions are used (cf. Moosbrugger and Frank 1992).

The k-means algorithm is one representative of the hill-climbing algorithms. Here, the cluster criterion is based on the mean value of the dissimilarity measure of a group (cf. Everitt 2011, pp. 121). A comparison of different metrics applied on the k-means method for partitioning cluster analysis is described by Kumar and Annie (2012).

The medoid algorithm works with a representative of the objects in a cluster. This representative is calculated by minimizing the distance metric within a cluster; the resulting object is called a *medoid*. The cluster algorithm based on medoids is called **Partitioning Around Medoids** (PAM) (cf. Kaufman and Rousseeuw 1990). A comparison of different partitioning cluster analysis methods, especially for binary data, is presented by Li (2005).

4.2.3 Model-Based Clustering

The model-based clustering methods all assume the underlying data to be normally distributed (cf. Fraley and Raftery 2000). For these methods there is also a package for GNU R. It is called *mclust* (Fraley et al. 2012). The algorithms underlying this clustering method are described in detail by Everitt (2011, pp. 187). As the data collected in the experiments are all dichotomous, there is no normal distribution. For this reason, the model-based clustering could not be applied.

4.3 Item Response Theory

The classical test theory assumes that a person's psychometric construct can be directly measured. Furthermore, disregarding errors each person achieves a specific score for that construct. But, as errors occur in a test, only a observed score can be measured. Classical test theory concerns the relations between the true score, the observed score, and the error of a person (cf. Novick 1966).

In contrast to the classical test theory, item response theory (IRT) assumes a psychometric construct to be latent and only observable through responses on items that are solved by a specific probability related to the person's ability. In general, item response theory assumes that the difference between a person's ability and the difficulty of an item is a predictor for the probability of an individual's response (cf. Rasch 1980). The probability of answering an item in the correct way can be expressed by a function of the item difficulty (β) and the person's ability (θ)

$$P(X = 1|\theta, \beta) = f(\theta, \beta)$$

4.3.1 The Logistic Models

The logistic model calculation is based on the transformed logit function:

$$p(z) = \frac{e^z}{1 + e^z}$$

where z has to be specified more exactly.

As mentioned above, the probability of solving an item is determined by the difference of a person's ability θ and the difficulty of the item (β). This leads to a formula representing the probability of a response of 1 as a function of item and person parameter.

$$p(x_j = 1|\theta, \beta_j) = \frac{e^{(\theta - \beta_j)}}{1 + e^{(\theta - \beta_j)}} \tag{4.1}$$

$p(x_j = 1|\theta, \beta_j)$ is the probability of a response of 1 on the jth item with regard to a person parameter θ and the item difficulty β_j.

The first and simplest form of the test models related to the item response theory is a model where only one parameter is to be estimated. This model only differs in the difficulty of the items and is called the **1PL model**. Additionally, it is assumed that the latent variable is one dimensional. If the interest is only in fitting a logit function to empirical data, it is necessary to change the discrimination value α of the *item characteristic curve* (ICC) to better fit the data. Nevertheless, for the 1PL model the discrimination value has to be the same for all ICCs of the investigated items. Including the discrimination value into Equation 4.1 leads to the general formula describing the 1PL model.

$$p(x_j = 1|\theta, \alpha, \beta_j) = \frac{e^{\alpha(\theta-\beta_j)}}{1+e^{\alpha(\theta-\beta_j)}} \tag{4.2}$$

For $\alpha = 1$ the model is called the **Rasch model**. While the 1PL model fits the model to the data, the Rasch model assumes a fixed value for α and because of that tries to fit the data to a given model (cf. Ayala 2009, pp. 14).

Besides the restriction on one parameter, there are several other model restrictions related to the Rasch model. First, the resulting person and item parameters are versatile. Thus, the complete set of parameter values can be shifted without changing the probability of solving an item. Because of that, comparing of the person's parameters among different tests is not applicable. The convention of fixing one item to a specific value is the addition of all values to 0 (cf. Rost 2004, p. 121).

$$\sum_{i=1}^{k} \beta_i = 0$$

A very strong restriction for the Rasch model is the fact that the items have to be locally stochastically independent. Thus, the probability of solving one item has to be independent of the solution of a previous item. The independence is assumed only for a given person parameter (cf. Strobl 2010, p. 18).

In addition, the Rasch model has to fulfill specific objectivity. In particular, the comparison of two persons has to be independent of the item. It is irrelevant if the basis for the comparison is an easy or a difficult item. This restriction is fulfilled by parallel ICCs and the model formula that the Rasch model is based on.

Finally, the most important restriction concludes the one dimensionality of the item and person parameters. The latent variable can only measure one psychometric construct.

A great advantage of the Rasch model is that if the model is valid, there are sufficient statistics for both the items and persons. The person parameter can be estimated by the number of solved items while ignoring which concrete items are solved. The same can be done with the item parameters.

All the formulas above only operate on given parameters. In reality, these parameters have to be estimated. There are two major methods for parameter estimation in the Rasch model. Both are based on the maximum likelihood estimation. The two methods differ in estimating all parameters together in one step or in two successive steps. The formula underlying the methods is the likelihood function for one person i and all items $j = 1, ..., m$

$$L_{u_i}(\theta_i, \beta) = \frac{e^{r_i\theta_i - \sum_{j=1}^{m} u_{ij}\beta_j}}{\prod_{j=1}^{m}(1 + e^{\theta_i - \beta_j})}$$

where r_i are the marginal scores, u_{ij} are the scores of the item and person, and β and θ are the corresponding parameters for the item and the person.

A method that estimates person and item parameters in one step is called the **joint maximum likelihood estimation** (JMLE). As the items, as well as the persons, are stochastically independent, the likelihood function is the product of the probabilities for all items and persons. As there has to be an estimation of a parameter for each person, the number of parameters to be estimated increases when adding more data to the model. As parameter estimation becomes better when the data increases and the consistency of the estimation decreases with the number of estimations, this method is rarely applied.

Other methods estimate the parameters one after another. The **conditional maximum likelihood estimation** (CMLE) only estimates the item parameters. For this purpose a conditional likelihood function for the sufficient statistic of the marginal scores r_i is calculated:

$$h(u|r, \beta) = \frac{e^{-\sum_{j=1}^{m} s_j\beta_j}}{\prod_{i=1}^{n} \gamma_{r_i}(\beta)}$$

where $\gamma_{r_i}(\beta)$ is the elementary symmetric function of degree r_i. The likelihood function is logarithmized, differentiated, and equaled to 0. As it is impossible to calculate the parameters directly, an iterative, computer-aided method is chosen. After estimating the item parameters, the person parameters are estimated in a second step based on the item parameters. The bias that occurs due to the estimated parameters is ignored. A disadvantage of this method is that the ability of persons solving all or no items cannot be estimated. Nevertheless, the person parameters can be extrapolated even though the test is too easy or difficult for the given individual.

Another method to estimate the parameters one after another is the **marginal maximum likelihood estimation** (MMLE). For this method a distribution of the marginals must be assumed. In general, the normal distribution is used, but this can be invalid if the normal distribution is not appropriate for the investigated population. After multiplying the density function to the likelihood function, the resulting function is integrated over θ.

$$L_u(\beta) = \int P(u, \theta|\beta) d\theta$$

$L_u(\beta)$ does not contain any person parameter and, therefore, the item parameters can be estimated with the common method of maximizing the likelihood. Similar to the CMLE method, the MMLE estimates the person parameters by using the estimated item parameters, ignoring the bias (cf. Strobl 2010, pp. 28).

In addition to the 1PL model and its special case the Rasch model, there are other models that, as a commonality, do not require the presumptions of the 1PL model. In the 2PL model – also called the Birnbaum model – a second parameter δ is introduced for each item. It describes the gradient of an item. The result of this is that the item characteristic curves can intersect each other and there is no longer specific objectivity. As the marginals are not sufficient statistics anymore, the conditional maximum likelihood estimation does not work for the 2PL model. Instead, the marginal maximum likelihood estimation is used. The Birnbaum model is described by the following formula, which introduces the parameter δ_j for α in Equation 4.2:

$$P(u_{ij} = 1|\theta_i, \beta_j, \delta_j) = \frac{e^{\delta_j(\theta_i - \beta_j)}}{1 + e^{\delta_j(\theta_i - \beta_j)}}$$

If the model considers the influence of solving the item by chance or guessing, a third parameter must be introduced. This leads to the 3PL model:

$$P(u_{ij} = 1|\theta_i, \beta_j, \delta_j, \gamma_j) = \gamma_j + (1 - \gamma_j) \cdot \left(\frac{e^{\delta_j(\theta_i - \beta_j)}}{1 + e^{\delta_j(\theta_i - \beta_j)}} \right)$$

4.3.2 Parametric Tests for Model Fitting

In item response theory there are two main goals of the model application. On the one hand, a set of homogeneous items that measure a specific number of psychometric constructs is of interest. On the other hand, the goal is to create a measurement framework that fits well. For these reasons there are several model tests that can be applied on the the models. Here, only tests for the Rasch model are presented.

The parametric tests verify the model for sub-populations. The Martin-Löf test is a test for item homogeneity. Furthermore, tests for model fitting are presented. As the parametric tests are based on approximately χ^2-distributed statistics, they can only be applied if the number of possible response patterns is much smaller than the investigated population. For that reason in Section 4.3.3 nonparametric tests are introduced that are based on the Monte-Carlo method. These tests work on small sample sizes, as can be found in the experiments for this thesis described in Section 7.6.

4.3.2.1 Martin-Löf Test

The Martin-Löf test checks the uni-dimensionality of the items. Basically, the test is based on the assumption that if the items are one dimensional, the likelihood of the item parameters of the complete item set are the same as in two separated item groups. The basis for the test statistic is the likelihood for all possible person scores. The probability for a specific person score is the relative frequency.

$$L_G = \prod_{r=0}^{k} \left(\frac{n_r}{n} \right)^{n_r}$$

Here, n is the number of participants, n_r is the number of participants with a score of r, and k is the number of possible scores in the test. In addition, the relative frequencies of the person score as a combination of person scores in the two item sets are calculated.

$$L_{(1)(2)} = \prod_{t=0}^{k} \prod_{u=0}^{k} \left(\frac{n_{\{tu\}}}{n}\right)^{n_{\{tu\}}}$$

Again, n is the number of participants. For each score, all possible separations of the value into partial scores u and t are used for the calculation. For the test statistic the likelihood of the subtests is set into relation to the likelihood of the complete test. As the difficulty level of the items in the subset could be different, the result of that division is weighted by the ratio of the likelihoods of the item parameters for the complete test (0) and the subsets (1, 2).

$$T_{ML} = 2ln\left(\frac{\prod_t \prod_u \left(\frac{n_{\{tu\}}}{n}\right)^{n_{\{tu\}}}}{\prod_r \left(\frac{n_r}{n}\right)^{n_r}} \cdot \frac{L_C^{(1)} \cdot L_C^{(2)}}{L_C^{(0)}}\right)$$

The test statistic is assumed to have an asymptotic χ^2 distribution with $k_1 \cdot k_2 - 1$ degrees of freedom. k_1 and k_2 are the number of items in the two compared item sets (cf. Koller et al. 2012, pp. 90). The restriction of the asymptotic distribution requires a large enough dataset (cf. Verguts and De Boeck 2000, p. 78). Davier (1997) proposes an absolute frequency of 5 for each cell of the table with all combinations of sub scores; in total $(k_1 + 1)(k_2 + 1) \cdot 5$ datasets. More precisely, this is only a minimum as all cells have to have the same probability (cf. Davier 1997, p. 31).

In most cases, especially in pilot studies such as the one presented in Section 7.6, the number of datasets is much smaller. For that reason there are nonparametric model tests that are based on the Monte-Carlo method. They are described in Section 4.3.3.

4.3.2.2 Graphical Model Proof

The graphical model proof (GMP) is a test for how the estimated parameters represent observed data. Concretely, if the model is valid the estimated parameters should be the same in two sub-groups of the investigated population; or, if all parameter values of both groups are plotted, they should be located on the diagonal of the plot. If the item is not located on the diagonal and the confidence interval in both directions (confidence area) does not intersect the diagonal either, the item does not fit the model and it has to be excluded from the test (cf. Strobl 2010, pp. 39).

The separation into different groups has to be iterated with different criteria. In theory all possible separations must be tested. In practice a specific level of certainty has to be reached by testing appropriate subgroups. For example, if the test is used for finding differences in gender, a separation on gender purposes can be done. In general, if information on the population is gathered and not included in the test, this information can be used for separation of the groups.

4.3.2.3 Likelihood Ratio Test

Similar to the GMP the likelihood ratio test (LRT) compares the model fit of subgroups of the population. Contrary to the graphical method, the LRT can compare more than two groups. In contrast to direct comparison of the estimated parameters for each subgroup, for the LRT the likelihood is calculated for the estimated parameters of each group and for the complete population. Afterwards, the likelihoods are compared. If the the likelihoods are equal, the estimated parameters fit the model in the subgroups, as well as in the complete population and the model is assumed to be valid. Otherwise, it would be better to estimate the parameters for each group and build different models.

The fitting is calculated by using a ratio of the likelihoods of the estimated parameters:

$$LR = \frac{L_u(r, \widehat{\beta})}{\prod_{k=1}^{K} L_{u_k}(r_k, \widehat{\beta}_k)}$$

where u are the answers on the items, r are the marginals, K is the number of groups, and $\widehat{\beta}$ are the estimated parameters. If the model is valid, the likelihood for the complete population is the same as the product of the likelihood of the subgroups and, therefore, the likelihood ratio (LR) has the value of 1. If there are differences in the groups, the likelihood of the single groups is bigger since the model for the subgroups fit the data better and results in a likelihood ratio smaller than 1. To calculate the significance, the LR has to be transferred into another statistic $T = -2lnLR$, which is χ^2 distributed with $(K-1)\cdot(m-1)-(m-1)$ degrees of freedom. If the values for T are large, the resulting small p-values cause a significant violation of the model fit (cf. Strobl 2010, pp. 41; Fischer and Molenaar 1995, pp. 86). Again, as mentioned in Section 4.4, the population has to be big enough in comparison to the dataset as the χ^2 distribution can only be assumed for relatively large datasets.

4.3.2.4 Wald Test

The Wald test is another test for model fitting that is based on the idea of separating the population into two groups. In contrast to the LRT, the standard Wald test operates on an item level. The separation can be based on different criteria, such as gender or others. Again, comparisons of the estimated parameters have to be proven in several steps. In theory all possible separations have to be checked. The formula for calculating the Wald test for an item j is:

$$W_j = \frac{(\widehat{\beta}_{j,1} - \widehat{\beta}_{j,2})^2}{\widehat{\sigma}_{j,1}^2 + \widehat{\sigma}_{i,2}^2}$$

with $\widehat{\beta}$ representing the estimated item parameters and $\widehat{\sigma}$ are the variations of the estimations of both groups. To obtain a test statistic we calculate:

$$T = sign(\widehat{\beta}_{j,1} - \widehat{\beta}_{j,2})\sqrt{W_j}$$

where the function *sign* calculates the sign of the expression. So, if $\widehat{\beta}_{j,1} < \widehat{\beta}_{j,2}$ the function returns $-$, otherwise $+$. The test statistic T is normally distributed and has a two sided rejection region. This results in an invalid model for very small and very big values of T. In addition to the standard Wald test for item parameters, there is a variation that proofs the model fit for the complete items in one step. Nevertheless, the standard Wald-test makes it possible to identify items that violate the model in general, but the violation of single model assumptions cannot be identified (cf. Strobl 2010, pp. 44; Fischer and Molenaar 1995, pp. 89).

4.3.3 Nonparametric Tests

The nonparametric tests for the Rasch model are based on a Markov Chain Monte-Carlo algorithm. It was first introduced by Ponocny (2001) and improved by Verhelst (2008). The algorithm is especially suitable for small sample sizes. Basically, the lack of data is reduced by simulating data matrices that have equal row sums than the estimated dataset. For that purpose two columns of the original matrix are randomly chosen. Afterwards, the rows with different values ($\{0,1\}$ or $\{1,0\}$) are randomly changed. This procedure leads to a new matrix with equal margins. As the algorithm needs the matrices to be independent of the initial matrix and to occur with the same probability, not all simulated matrices can be used for the calculation. More precisely, for purposes of independence a predefined number of matrices are simulated, but skipped. The first matrix included in the test is the first after this burn-in period. Due to the fact that only two columns are changed in every step, the resulting matrices would be very similar. Because of that a step size is defined. So, every simulated matrix within this step size is skipped and only the last one in included in the test. Thus, for a step size of 32 and a burn-in period of 100 in total $100 \cdot 32 + 100 \cdot 32 = 6400$ matrices are simulated, but only 100 effective matrices are used (cf. Koller and Hatzinger 2013, pp. 4).

There is a test statistic for a global test on homogeneity and local stochastical independence. For the global test statistic, the phi correlation between all pairs of items is calculated for the original matrix. The test statistic is the sum of the differences between the correlations of the original matrix and the corresponding average item pair correlations of the simulated matrices. The model test then calculates the relative frequency of all test statistics of the simulated matrices with a higher value than the one of the original matrix. If the result is significant, there is a violation either in homogeneity or in the locally stochastical independence (cf. Koller and Hatzinger 2013). As mentioned before, the main goal of the investigations presented in Section 7.6 is the examination of a homogeneous itemset. Because of that, the test for homogeneity is presented in the next subsection. Additionally, the resulting items are proven to fit a Rasch model and to build a proper test framework for the examined psychometric construct. For that purpose a nonparametric test for local stochastic independence is presented at the end of this section.

4.3.3.1 Test for Homogeneity

A violation in homogeneity can be assumed if there are too few response patterns that are equal ($\{0,0\}$ or $\{1,1\}$). The test statistic is calculated by summing up the equal response patterns for each person v for the investigated items i and j.

$$T_1m(A) = \sum_{v=1}^{n} \delta_{ijv} \text{ where } \delta_{ijv} = \begin{cases} 1, & \text{if } x_{vi} = x_{vj}, \\ 0, & \text{if } x_{vi} \neq x_{vj}. \end{cases} \tag{4.3}$$

According to the global test, the test statistic is calculated for all simulated matrices (A_s). In the end, the significance is calculated as the relative frequency of all test statistics of the simulated matrices (n_{sim}) that are lower than the test statistic of the original matrix (A_0).

$$p = \frac{1}{n_{sim}} \sum_{s=1}^{n_{sim}} t_s \text{ where } t_s = \begin{cases} 1, & \text{if } T_s(A_s) \leq T_0(A_0), \\ 0, & \text{else.} \end{cases}$$

So, the test compares the observed test statistic with simulated test statistics. If the simulation results in more unequal item pairs, the observed items have a higher probability of being homogeneous (cf. Koller and Hatzinger 2013).

4.3.3.2 Test for Local Stochastic Independence

The assumption of local stochastic independence has two facets. On the one hand, for a proper test, the response patterns of the items are not allowed to be too similar. With regard to two items, this can be expressed by the number of equal response patterns. The test statistic presented in Equation 4.3 is suitable for that purpose as well. For a better identification in Section 7.6 it is renamed to **T1**. Nevertheless, the model test has to be modified. In contrast to the assumption of homogeneity, violation of the local stochastic independence is expressed by too many equal response patterns.

$$p = \frac{1}{n_{sim}} \sum_{s=1}^{n_{sim}} t_s \text{ where } t_s = \begin{cases} 1, & \text{if } T_s(A_s) \geq T_0(A_0), \\ 0, & \text{else.} \end{cases} \tag{4.4}$$

On the other hand, violation of the local stochastic independence can be a result of a learning effect within the items; participants learn from a certain answer on a previous item how to solve another. As Equation 4.3 counts the patterns $\{0,0\}$ and $\{1,1\}$, the test statistic is not appropriate for this purpose, because the learning has no effect on the response pattern $\{0,0\}$. For this reason, only those patterns that are both answered correctly are summed.

$$T_{1l}(A) = \sum_{v=1}^{n} \delta_{ijv} \text{ where } \delta_{ijv} = \begin{cases} 1, & \text{if } x_{vi} = x_{vj} = 1, \\ 0, & \text{else.} \end{cases}$$

Again, the model test presented in Equation 4.4 can be used. As mentioned above, all these tests have in common that a model violation can be assumed if the test is significant.

4.4 The Latent Trait Model

The latent trait model (LTM) is a special kind of factor analysis. It can handle dichotomous data that often occurs while modeling logistic models in the sense of item response theory (see Section 4.3). As the common item response theory models such as the one described above work on ordinal or better dichotomous data, mostly with missing data, the common factor analysis does not work (cf. Bond and Fox 2007, p. 252; Ayala 2009, p. 44).

> “Whenever one factor analyzes a correlation matrix derived from binary data, there is possibility of obtaining artifactual factor(s) that are related to the nonlinearity between the items and the common factors. These ‘factors of curvilinearity’ have sometimes been referred to as ‘difficulty’ factors and are not considered to be content-oriented factors.” (Ayala 2009, p. 44)

This was also mentioned by McDonald and Ahlawat (1974); “Hence the common factor analysis of a covariance matrix from a large enough sample of binary variables conforming to the linear model with t latent traits (distinct ‘content factors’) should yield just t common factors, no more, whether or not the difficulties vary” (McDonald and Ahlawat 1974, p. 87).

To avoid the problems of difficulty factors a non-linear factor analysis such as the one introduced by McDonald (1965) should be applied. An application of this is shown by Ayala (2009).

According to Bartholomew (2008), the task of building a latent trait model for binary data has the following objectives:

1. To explore the interrelationships between the observations.
2. To explain the interrelationships with a small number of latent variables, if possible.
3. To assign a score to each individual for each latent variable on the basis of the given responses.

The basis for all latent trait models on binary data is a data matrix with a response vector (see Section 4.3) for each individual. The columns contain a 1 if the answer to the corresponding question is “yes,” “right,” or something equivalent; otherwise they contain 0.

Besides the binary data matrix, there is a more compact way of displaying the data. The occurrences of the response patterns in the data are summed up in a table. For p variables there are 2^p possible response patterns. If p is large in comparison to the number of investigated individuals n (approximately $2^p \geq 5n$), there are several patterns that do not occur in the table. This discrepancy makes the calculation of model fitting difficult for small populations and many investigated variables (cf. Bartholomew 2008, pp. 209).

For this purpose Bartholomew (2008) introduced a goodness-of-fit test that is based only on response patterns. In particular, he proposes two test statistics that compare the observed frequencies of the response patterns to the estimated frequencies that are expected under the model. So, basically the tests measure the closeness of the observed data and the estimated parameters.

The log-likelihood ratio test statistic is based on the logarithm of the ratio of the observed frequency of the response patterns $O(r)$ and the expected frequency $E(r)$.

$$G^2 = 2\sum_{r=1}^{2^p} O(r) log_e \frac{O(r)}{E(r)}$$

where p is the number of items and r represents a response pattern.

Alternatively, the Pearson chi-square test statistic can be applied.

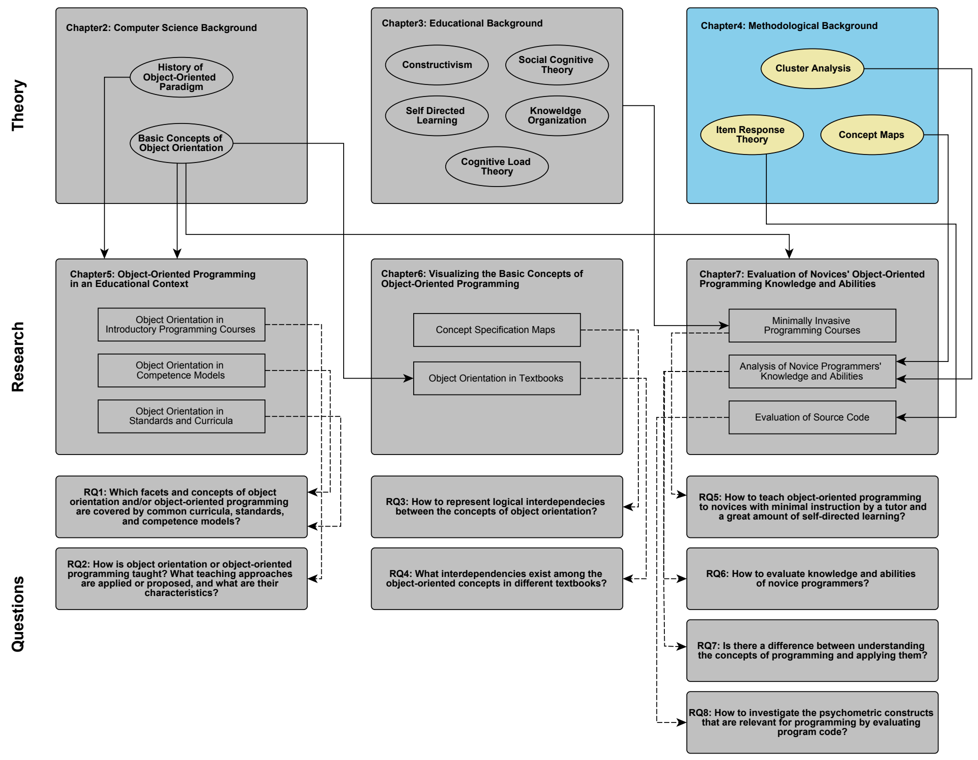
$$X^2 = \sum_{r=1}^{2^p} \frac{(O(r) - E(r))^2}{E(r)}$$

If the model is valid, both test statistics have an approximate χ^2 distribution with $2^p - p(q+1) - 1$ degrees of freedom with p for the number of items and q for the number of factors. As mentioned above, this is only true for large datasets in comparison to the total number of participants. In particular, each pattern should occur at least five times (cf. Verguts and De Boeck 2000, p. 78).

Relevance for this thesis

In this thesis concept maps are applied to evaluate conceptual knowledge of students at the start of their studies in computer science. First, the knowledge representation of novice programmers is investigated with regard to knowledge development during an experimental course (Section 7.5.2). Additionally, concept maps are applied to detect misconceptions in the knowledge representation of novice programmers (Section 7.5.3). Finally, the knowledge representation is compared to the program code produced during the experimental programming course (Section 7.5.4). In addition, cluster analysis is applied when investigating novice programmers' knowledge and misconceptions. Here, both presented clustering methodologies are applied. The model-based clustering was used in the publications but figured out to be invalid as described in Section 4.2.3.

In Chapter 7, a new application of item response theory is introduced. In Section 7.6 the item response theory is applied on the program code. For that purpose a set of items is defined that could be answered by analyzing the provided program code of participants of an experimental course. In contrast to a classic item response theory assessment, the items are not posed directly to the participants, but are implicitly contained in the underlying programming task. The presented model tests were applied to find a homogeneous itemset. Finally, the model-fitting tests were applied to present an evaluation of novice programmers' abilities.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

5 Object-Oriented Programming in an Educational Context - A Literature Review

After introducing the theory underlying this thesis, representations of object orientation in computer science education are provided. More precisely, an overview of the development and present studies of object orientation in introductory programming courses is presented. Additionally, language examples and their evaluation, as well as the selection process, are described in Section 5.1.2. Generally, the questions are answered, how object orientation or object-oriented programming is taught, and what teaching approaches are applied or proposed (**RQ2**).

Besides the educational representation in introductory courses for computer science and programming, there are representations of object-oriented concepts in the literature that is related to computer science education. So, in the second part of this chapter the occurrence of object orientation is investigated in recent competency models (Section 5.2), as well as in national and international educational standards and curricula (Section 5.3). The literature analysis shows, therefore, how object orientation and object-oriented programming are represented in common curricula, standards, and competency models (**RQ1**).

5.1 Object Orientation in Introductory Programming Courses

The question of why programming should be taught was investigated in a study on the pedagogical content knowledge on programming. Besides several other questions that are mentioned below, this central one was asked as well. One possible answer is that it is a powerful tool for learning problem-solving and design or thinking strategies. Another aspect is communication, because programming requires a thinking about how to tell a machine what you want it to do (cf. Saeli et al. 2011, pp. 77).

Another investigation on the role of programming in computing education was conducted by Schulte (2013). He summarized educational characteristics of programming and discussed programming and the goals of education. In detail, the following list points out his summary:

- “A basic requirement of programming is automation.”
- “Programming is a special type of interaction with the computer, connected with the loss of direct manipulation and the need to use an abstract notation.”
- “Programming includes coping with complexity in a specific form.”
- “A program addresses not only the computer as ‘reader’, but also (and probably even more) human reader, programming therefore includes aspects of a communication process.”
- “As there is often no clear path between the problem and the solution, programming includes a creative process of seeking and finding solutions.” (Schulte 2013, p. 20)

According to the role of programming in general, the development of object-orientation in introductory programming courses and the current state is presented. In an overview article on different programming paradigms in an educational context, Luker (1989) stated that, “[w]e should cast our vote for this style of programming [object-oriented programming], because it is a natural one. The language should enforce the use of objects, as do SMALLTALK and BETA” (p. 256). Five years later, the situation had changed slightly.

> “Today, it is impossible to avoid the term [object-oriented], it appears in nearly every book, journal and magazine which relates to computer software, and yet, while they rush to embrace it, many computer professionals and educators still do not understand what object orientation means. [...] However, if we fail to make a successful shift to the ‘new’ paradigm, its promise will remain unfulfilled.” (Luker 1994, S. 56)

Mitchell (2000) summarizes the paradigm shift in industry and tries to explain why it is difficult to integrate the “new” paradigm into computer science education. He proposes the following four guidelines for introducing the object-oriented paradigm. He suggests that students do not favor any specific problem-solving methodology, that there are differences between teaching approaches connected to the procedural or object-oriented paradigm, and that students’ former programming experiences have to be taken into account. Finally, the last statement has the strongest influence and has found its way into several publications: “The paradigm shift to objects first impacts the entire curriculum” (Mitchell 2000, p. 103).

Kölling (1999*a*) describes the influence of the shift on computer science education as follows: “For a long time, object-oriented programming was considered an advanced subject that was taught late in the curriculum. This is slowly changing: more and more universities have started to teach object orientation in their first programming course. The main reason for doing this is the often cited problem of the paradigm shift” (p. 1).

The most important change that came along with the paradigm shift, and has a very strong influence on education, is the change in view on processes. While in the procedural paradigm one process is responsible for the functionality of the complete program, in the object-oriented paradigm the objects act on their own as far as possible. This implies that in the procedural view records are changed by procedures from the outside, while in the object-oriented view, objects call methods to initiate the change of the values of an object. In particular, in the object-oriented paradigm, processes are bound directly to the objects (cf. Vujošević-Janičić and Tošić 2008, p. 71; Hubwieser 2007*b*, p. 209; Temte 1991, p. 75). Another major change in perspective concerns the implementation of variants. The object-oriented paradigm enables polymorphism. This encapsulates the conditional selection of the control flow depending on the "type" of the acting element (cf. Garrigue 1998, 2000).

Finally, another difficulty has to be considered when introducing object-oriented notions. Objects are implemented by references or pointers in almost all programming languages. More precisely, an object is only an allocated part in computer memory with a pointer or reference pointing on it. For students it is hard to learn if two references point to the same object. When switching from the procedural paradigm to the object-oriented paradigm, this understanding is crucial as the changing of values is different. While in the procedural paradigm access to a variable affects only the variable that is being addressed, in the object-oriented paradigm references related to other attributes could be affected as well (cf. Hubwieser 2006, p. 10).

The following subsection provides an overview on the methodology of introducing the basic concepts of programming in general and object orientation in particular. There is a large amount of investigation on different approaches. Shall the object-oriented concept be taught first or later? Is the modeling aspect first or is it the programming aspect?

The second important aspect is related to the language and environment in which to apply the concepts that are being taught in the introductory course. There are several types of languages or environments, each with advantages and disadvantages, which have to be taken into account when planning the course.

> "The use of a language or environment designed for introductory pedagogy can facilitate student learning, but may be of limited use beyond CS1. Conversely, a language or environment commonly used professionally may expose students to too much complexity too soon." (ACM/IEEE-CS Joint Task Force on Computing Curricula 2013, p. 43)

5.1.1 A Suitable Educational "Paradigm" for Introducing Object Orientation

> "Object Orientation [OO] is sometimes regarded as an advanced topic that is hard to teach. This might be true if you teach it to students who have a background in another programming paradigm, but our experience is that OO is ideal to use as a first paradigm." (van Roy et al. 2003, p. 270)

There are many sources describing the methodology of introducing the object-oriented paradigm in an introductory course. Most of them comprise the different approaches of "objects-first" or "objects-later". But, what are the differences and is it really important to distinguish the two? Here, an overview on the topic is presented, but no evaluation is given on the different ways. Besides the pure approaches, there are other methodologies for introducing programming. This affects the different aspects of object orientation in general (modeling or programming) and the time of introducing specific programming concepts in particular. Figure 5.1 gives an overview on all these approaches. *Modeling* includes the introduction of all modeling aspects. *Object* or *Class* imply the introduction of the concepts of object and class, whereas, *Object-oriented programming* includes the programming notions of the object-oriented paradigm. Finally, *Algorithm/Control structures* includes the notions of algorithms and their implementation including control structures.

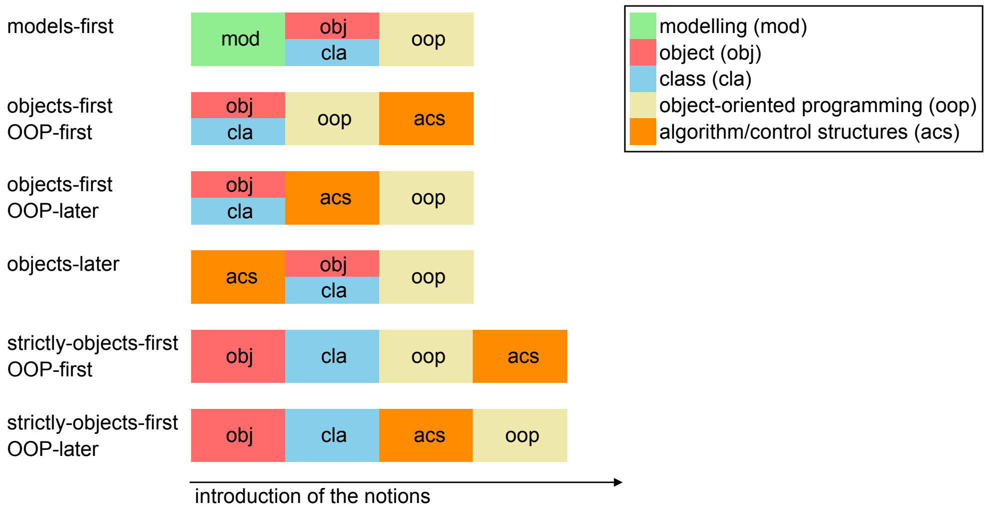


Figure 5.1: Overview of the different educational “paradigms” for introducing object orientation and the corresponding programming notions

5.1.1.1 When to Introduce the Object-Oriented Notions?

Lewis (2000) figured out one of the central aspects if you are talking about object orientation and the way to learn it. “A distinction must quickly be made between initially writing classes that define objects, and using objects defined by preexisting classes. [...] The evaluation of any approach should focus on file presentation of the underlying model” (p. 246).

Based on this study, Bennedsen and Schulte (2008) asked over 700 teachers about their understanding of objects-first. Their answers are categorized into three categories:

Using objects: “At the beginning of the course, the student uses objects implemented beforehand. When the student has understood the concept *object,* he moves on to defining classes by himself. Focus is on usage before implementation” (p. 23).

Creating classes: “The student both defines and implements classes and creates instances of the defined classes. Focus is on the concrete-creative part of programming” (p. 23).

Concepts: “This involves the teaching of the general principles and ideas of the object-oriented paradigm, focusing not just on programming but on creating object-oriented models in general. Focus is on the conceptual aspects of object orientation” (p. 23).

There are three common sequences of objects-first courses. Two of them use objects first and then create classes and introduce concepts in different orders. The third variant simply creates classes that are followed by introduction of the concepts (cf. Bennedsen and Schulte 2008, p. 23).

Diethelm (2007) introduced a stricter version of the objects-first approach. She states that most problems with object orientation – and especially the modeling aspect – lie in the use of classes instead of objects. Another aspect mentioned by Diethelm (2007) is an approach that applies modeling techniques for introducing the object-oriented notions. This approach is called models-first. Based on this idea, objects are first modeled on their own and then classified. Both modeling and programming should focus on objects rather than classes. This approach of Diethelm (2007) is called strictly-objects-first.

The objects-later methodology for learning object orientation also emphasizes the modeling aspect. In contrast to the objects-first approach, it starts with algorithmic concepts including control structures and introduces object modeling after that. The problems and effects of the objects-later approach has been discussed in literature, especially at the beginning of the change to the object-oriented paradigm; for example, see (DeClue 1996).

In several publications the objects-later approach – with the paradigm shift from procedural to the object-oriented paradigm – is said to be very difficult and not applicable in introductory courses (cf. Saeli et al. 2011, p. 80).

A synonym for the objects-first approach is introduced with the notion of starting with an object-oriented design. The modeling of objects is the first step when introducing object orientation. Furthermore, concepts like inheritance, polymorphism, and generalization are introduced by design (cf. Adams and Frens 2003).

An investigation of the predictors for success in an object-first course showed that the common predictors such as previous experience in programming and mathematics – which have an influence in an object-later course – have no influence. In their study they built a model with seven predictors for success and investigated those that had the smallest influence on the variance in a linear regression model. Besides previous programming experience and knowledge and abilities in mathematics, gender influences were measured. The influence of gender does not differ in the two approaches (cf. Ventura 2005).

Most courses have now changed to an objects-first approach, which is either more or less strict. Nevertheless, there were and are several reservations in the object-oriented approach. At the beginning of the paradigm shift in education, Decker and Hirshfield

(1994) listed the top ten reservations and attempted to discuss whether the restrictions are true or just fear of new things. In the end, they recommend an objects-first approach, even though it suffers reservations (cf. Decker and Hirshfield 1994). In a study published ten years later, an indirect comparison of the two main approaches described above was conducted. Decker (2003) investigated the performance of two student groups in an object-oriented CS2 course. The first group had an introductory course according to the objects-first approach, while the second group started with the procedural paradigm in the very beginning (objects-later). The group that used objects from the beginning performed better in the more advanced object-oriented concepts, although the second group was also introduced to the object-oriented concepts. Furthermore, the algorithmic performance was equal in both groups, which suggests that both approaches are successful in the procedural topics (cf. Decker 2003).

An application of the objects-first approach with emphasis on the modeling aspects of object orientation is implemented in the Bavarian Gymnasium[8] (cf. Hubwieser 2007*a*). After the introduction of the basic object-oriented notions using standard office applications, object-oriented programming is introduced by first introducing control structures and then object-oriented concepts (see Section 5.3.6.2). The different notions for introducing object-oriented programming forms the content of the next section.

5.1.1.2 What is the Most Suitable Order for Introducing Programming Notions Related to Object Orientation?

One possible approach which is based on “programming-first” is presented by Alphonce and Ventura (2002). In this approach object orientation is introduced by design. They mention that they have tried to solve the weaknesses of the “OOP-first” approach that is mentioned in the ACM/IEEE curriculum of 2001 (CC2001).

> “All introductory approaches discussed in CC2001 have distinct strengths and weaknesses. The programming-first approach suffers from other weaknesses beyond the ones quoted above. CC2001 notes that it has endured because of some significant strengths, notably that students are equipped with programming skills in their first year, and that subsequent courses can leverage these skills in their curricula and can also hone these skills.” (Alphonce and Ventura 2002, p. 70)

They stick very close to the Unified Modeling Language (UML). They describe the importance of design as follows:

> “In early programming assignments students are given a design, expressed in UML, along with a skeletal solution. The students’ task is to supply their own code to complete an implementation of the given design. In later assignments students are supplied with a plain-language description of

[8]Gymnasium is the Bavarian grammar school. It is a very specific type of school. For that reason it is labeled with the German word “Gymnasium” or the plural word “Gymnasien”.

> the requirements, from which they must develop a design, expressed in UML, before they begin writing code." (Alphonce and Ventura 2002, p. 72)

The first general decision that has to be made is about the order of programming and modeling. Bennedsen and Caspersen (2004) present a model-first approach. They start with the modeling of classes and objects in a programming task just before writing a single line of code. The programming concepts are introduced later on in a cascading approach. This emphasizes that object orientation is not just another solution or technology for programming issues.

Based on the paper of Alphonce and Ventura (2002), Bennedsen and Caspersen (2004) introduce an objects-first course with a strong emphasis on the modeling aspects, which includes conceptual modeling in the programming introduction.

> "The starting point is a class and properties of that class. One of the properties of a class can be an association to another class; consequently the next topic is association. This correlates nicely to the fact that association (reference) is the most common structure between classes (objects). Composition is a special case of association; composition is taught in the next round of the spiral. The last structure to be thoroughly covered is specialization. Specialization is the least common structure in conceptual models, and it bridges nicely to the second half of the course where the focus is on software quality and design." (Bennedsen and Caspersen 2004, p. 478)

According to the abstraction levels for the interpretation of UML class model, they find three advantages of the inclusion: a systematic approach to programming, a deeper understanding of the programming process, and the focus on general programming concepts instead of language constructs in a particular programming language (cf. Bennedsen and Caspersen 2004).

Another example of an iterative approach that is based on modeling and design aspects was conducted by Hadar and Hadar (2007). They switch between abstract modeling sessions and concrete implementation sessions in an iterative way. With this approach they try to emphasize the abstract thinking tasks. The change of the modeling and programming tasks should prevent having to choose one of the abovementioned approaches (cf. Hadar and Hadar 2007).

After deciding about the position of programming in the course, the order of programming concepts can also be different. Ehlert (2012) discusses one of the most central questions about object-oriented programming in computer science education. He provides an overview of the typical concepts of the introduction of object-oriented programming (OOP) and their order within the course (Ehlert and Schulte 2009*b*). These topics include notions of both programming paradigms and cover the main items of the object-oriented "quarks" (see Section 2.2): objects and classes, data types, variables and attributes, constants, methods and procedures, control structures, and inheritance and associations. An overview of the different order is shown in Table 5.1.

OOP-first	OOP-later
Classes and objects	Data types (incl. variables, constants)
Attributes (incl. data types)	Control structures
Methods (including control structures)	Procedures
Inheritance	Classes and objects
Association	Inheritance and association

Table 5.1: Comparison of the topics of the OOP-first and OOP-later approaches and their order of introduction, (cf. Ehlert and Schulte 2009*a*)

Ehlert (2012) states that OOP-first concentrates on the same topics as OOP-later. The only difference is their order.

> The main difference of both teaching approaches is the sequential order of topics. Both approaches focus more in programming as in the modeling, without giving up the modeling[9].

Ehlert (2012) investigates possible differences in the two approaches – OOP-first and OOP-later – concerning the outcome related to when object-oriented programming is introduced. The result is that there are no significant differences between the two approaches. The only factor differentiating the outcome is the personal preference of the teacher.

> No significant differences between the OOP-first approach and the OOP-later approach were observed or measured regarding the learning gain of pupils / students (p-values for the nine requested topics are between 0.18 and 0.83). OOP-later students had a (significantly) better subjective experience in the relevant fields ("school", "subject" and "topic") as well as in the dimensions of experience (emotional, cognitive and motivational experience)[9].

Besides the small sample size, another difficulty in the work of Ehlert (2012), as with others in the literature on the topic, is the missing distinction of objects-first or objects-later and OOP-first or OOP-later. While the first approaches relate to the notions of objects and classes and their position in an introductory course, the second two approaches are related to the programming aspects of object orientation such as inheritance and polymorphism, as well as the method and attribute concept (cf. Diethelm 2007, p. 22).

[9]Translated by the author from German. The original source is listed in Appendix C

5.1.2 An Appropriate Language for an Introductory Programming Course

Knudsen and Madsen (1988) focus on the fact that learning to program in an object-oriented manner is more than learning a new language that is object-oriented. Nevertheless, in the last few years a large variety of languages have been developed.

> "The use of 'safer' or more managed languages and environments can help scaffold students' learning. But, such languages may provide a level of abstraction that obscures an understanding of actual machine execution and makes is difficult to evaluate performance trade-offs. The decision as to whether to use a 'lower-level' language to promote a particular mental model of program execution that is closer to the actual execution by the machine is often a matter of local audience needs." (ACM/IEEE-CS Joint Task Force on Computing Curricula 2013, p. 43)

At the start of implementing the paradigm change in the early 1990s, there were only a few object-oriented languages that were applicable for educational purposes. This is why in the first years languages from industry that had mostly procedural roots were used. With time more explicit educational languages and environments were developed. DeClue (1996, p. 233) provides an overview of the languages that were found in the very beginning.

As the programming environments are mostly related to a specific language in a strong manner, they will be investigated in combination with each other. A first set is taken from industry and the selection is substantiated with the necessity to prepare students for industries' needs (Pears et al. 2007, p. 207). A second group provides either a subset of a common industrial language or an environment for one of those languages that is especially designed for educational purposes. Another set of languages is developed only for educational reasons. Most of those languages provide a graphical programming environment that is often related to a kind of gaming.

The category of languages with industrial roots are historically related to the development of the programming paradigm. So, the first set of languages has its origins in the "pre-shift" times (cf. Fleury 2000). Lewis (2000) presents in his paper on the myth about object orientation two widely spread languages in introductory courses. The first language is C++, which is a hybrid language (see Section 2.1).

> "The hybrid nature of C++ is one of the problems that educators have with the decision to make C++ the language of choice taught in high schools for Advanced Placement credit." (Lewis 2000, p. 248)

The other language is Java and is the most widely spread language (Pears et al. 2007). Even if the design of Java tend to be object-oriented, it is not a pure OO-language. "The existence of primitive types is one reason. [...] More important, the requirement that all methods be defined as part of a class does not automatically lead to an object-oriented design. Procedural abstraction could be allowed to dominate, destroying the conceptual elegance of a proper design" (Lewis 2000, p. 248).

Furthermore, there are languages designed especially for the object-oriented programming paradigm, but unlike C++, Java or recently C# they have not found broad application. A famous candidate, which has its role in industry, but was designed with education in mind, is Smalltalk of Allan Kay (cf. Wren 2007, p. 10). Another candidate of this set is Eiffel of Bertrand Meyer. There are several other languages that are less famous. One recent example is Grace, which was developed especially for novice programmers and is purely object oriented (cf. Black 2013).

Another set of languages includes those that are related to one of the above, but mostly only with a subset of concepts. BlueJ is based on Java, but omits the main-method concept. In contrast, all objects are either constructed by another object or by the user in the development environment (cf. Bergin et al. 2005; Kölling and Rosenberg 2001, 1996; Kölling et al. 2003). Another candidate of this set is the Java version of Karel the Robot, which was designed in the 1980s and focused on the current programming paradigm in the given time (cf. Bergin et al. 2005). A comparison of Karel J Robot and BlueJ is described by Borge (2004). An alternative, especially for the first contact with object orientation in the very beginning of an introductory course, is a Java library that provides simple geometric objects that can be displayed in an easy way. Furthermore, there is a simple “sandbox” application for the first tries of the students. After becoming familiar with the objects, attributes, and methods that are provided by the library, simple applets using the library can be implemented by the students on their own. The “ObjectDraw” library is described in detail by Bruce et al. (2001).

In the set of languages that are designed for educational purposes, there are some languages that are only applied in a few courses and others that are applied around the world. Nearly all examples have in common that they provide a kind of setting, like a game or movie, where objects can interact with each other or with the environment. The capabilities of interaction and the kind of objects vary strongly through the different languages. While a language such as Alice provides a 3D environment and a huge set of possible objects and interactions (cf. Sattar and Lorenzen 2009), Robot Karol has only a few objects and a limited scenery where the robot can act. Another difference in the languages is in the programming itself. While Robot Karol has its own languages for programming, Scratch or Alice provide pre-defined programming elements that can be altered and combined.

An investigation of an introductory course based on Alice as programming language pointed out some observations that give a good impression of the notion of this set of programming languages. Cooper et al. (2003) observed a strong contextualization of objects and classes. This emphasize the notion that object-oriented programming is implementing objects of the real world. Furthermore, the notion of encapsulation is emphasized, because the objects can be moved in the environment, but the spatial coordinates are not accessible in a direct way. In addition, this fact points out the sense of message passing between objects. Nevertheless, the environments of this set provide the used control structures for selection and, therefore, hide any syntax. The syntax problems are kept out, which is an advantage in the beginning, but has to be considered in the later educational process (cf. Cooper et al. 2003, p. 193).

Another classification of programming languages and its education is conducted by Kelleher and Pausch (2005). They introduce a taxonomy for introductory programming languages based on the languages' notions and basic features. Furthermore, an example is given for each element in the taxonomy. The list of programming languages and environments is completed by a list of the programming concepts that are implemented in the languages (cf. Kelleher and Pausch 2005, pp. 59).

In advance of the development of the BlueJ system mentioned above, Kölling (1999*b*) investigated environments and languages around object orientation that were common in the 1990s. He defines seven criteria for a programming environment to be suitable for novices and the object-oriented paradigm: ease of use, integrated tools, object support, support of code reuse, learning support, group support, and availability (cf. Kölling 1999*b*). Additionally, Kölling (1999*a*) defines a set of requirements for an educational programming language. He states that programming "languages are not good or bad per se; they are good or bad for a specific purpose" (Kölling 1999*a*, p. 3). The requirements (clean concepts, pure object orientation, safety, high level, simple object/execution model, readable syntax, no redundancy, small, easy transition to other languages, support for correctness assurance and suitable environment) are checked for the most common object-oriented programming languages C++, Java, Eiffel, and Smalltalk. The results of these investigations led to the development of BlueJ.

5.2 Object Orientation in Competency Models

Although there is a long tradition in publishing educational standards and recommendations of curricula in computer science, there is a lack of competency models. In fact, there is no model that is empirically founded and well established in research. Weinert (2001) defines competency as "a roughly specialized system of abilities, proficiencies, or skills that are necessary to reach a specific goal. This can be applied to individual dispositions or to the distribution of such dispositions within a social group or an institution" (Weinert 2001, p. 45). In recent years, there are several efforts to develop competency models. The models that have been developed are at different stages of completion. The MoKoM-project, which covers competencies for informatics modeling and system comprehension is almost finished. Another model by Bennedsen and Schulte (2006) is just a suggestion and not yet implemented or even validated. These two projects mentioned are described in a more detailed way in the following subsections. The facets related to object orientation are of special interest.

5.2.1 Competency Model of Object Interaction

In 2006 Bennedsen and Schulte introduced a theoretical base for measuring the understanding of object interaction. This model was revised in 2013 (Bennedsen and Schulte 2013). Although the title suggests a competency model, only a taxonomy model is presented in the paper. In fact, there is no relation to any definition of competency at all. Nevertheless, the defined hierarchy of developing expertise in the understanding of object interaction is tested and validated.

The first version in 2006 contained four levels in the hierarchy: interaction with objects, interaction with object structures, interaction on dynamic object structures, and interaction on dynamic polymorphic object structures. In the revision in 2013 the concept of polymorphism, which was the focus of the first version, was excluded due to concentration on the basic concepts that are necessary for executing object-oriented programs. The following list presents the revised categories and their descriptions.

Interaction with objects "The student can understand simple forms of interactions between (a couple of) objects, such as method calls and creation of objects (including simple method calls to initialize an object like adding objects to a container in the beginning). The student is aware that the results of method calls depend on the identity and state of the object(s) involved. The objects do not change state" (Bennedsen and Schulte 2013, p. 13).

Interaction on object structures "The student is able to comprehend a sequence of interactions, in which the history of changes has to be taken into account. Interaction is dynamically changing a structure of objects including iteration through object structures. The structure is created and changed explicitly via creations, additions and deletions" (Bennedsen and Schulte 2013, p. 13).

Interaction on dynamic object structures "The student knows the dynamic nature of object structures, understands the overall state of the structure and is aware of reference semantics. The student takes into account that the interaction on the structure or elements of it can lead to side-effects (e.g. implicit changes in the structure). E.g. several elements in a structure refer to the same object, so a change in one reference has effects on others" (Bennedsen and Schulte 2013, p. 13).

The last hierarchy level including polymorphism is dropped out as the concept is excluded in general.

Besides the taxonomy of the understanding of object interaction, Bennedsen and Schulte (2006) introduced a test instrument to validate the model. The test on the initial model led to the exclusion of polymorphism. Nevertheless, the taxonomic character of the model could be proved. A broad test of the revised instrument has not yet been applied, but the model has been proven on a small sample.

Although Bennedsen and Schulte (2013) tried to define a competency model, they do not refer to any common definition of competency. Furthermore, as the authors state themselves, the design of the hierarchy levels is just one alternative out of many others. This results in difficulties in putting each possible example in the right category. In addition, the test instrument faces the same difficulties as former ones. It includes concepts in the assessment items that are not meant to be tested anyway. The use of given classes such as *ArrayList* make the test instrument very specific for a special population.

5.2.2 Competency Model on Informatics Modeling and System Comprehension (MoKoM)

The MoKoM project started the development of a competency model for computer science in 2009. More precisely, MoKoM focuses on informatics modeling and system comprehension. Development of the model was conducted in two major steps. For the first step, a model was derived on a theoretical basis. For that purpose, expert papers and computer science curricula were investigated (Kollee et al. 2009; Magenheim 2005; Magenheim, Nelles, Rhode, Schaper, Schubert and Stechert 2010; Schubert and Stechert 2010). According to the theoretical approach, the existing model was refined by expert interviews based on the critical incident theory (cf. Magenheim, Nelles, Rhode and Schaper 2010). Linck et al. (2013) describe the final version of the model. In the last step the model has been applied in secondary schools with a developed assessment tool (cf. Neugebauer et al. 2014).

The MoKoM-model consists of five major competency dimensions: system application (K1), system comprehension (K2), system development (K3), dealing with system complexity (K4), and non-cognitive skills (K5). All these dimensions have sub-competencies below the first level and some of those have a third level.

The dimensions, sub-competencies, and the third dimension related to object orientation are listed below.

K3.4.1 Know & apply object-oriented terminology

K3.4.2 Execute object-oriented decomposition

K3.4.3.2 Develop object diagrams

K3.4.3.4 Develop analysis class diagrams

K3.5.4.1 Develop design class diagrams

K3.6.1 Know & use object-oriented programming

K3.6.2.2 Transform class diagrams to source code

Although the model mainly covers modeling aspects, the object-oriented programming is explicitly mentioned. Nevertheless, the object-oriented modeling is the main focus.

5.3 Object Orientation in National and International Education Standards and Computer Science Curricula

From a conceptual view, as in Section 2.2, a definition of some basic concepts is useful. These could be the quarks by Armstrong (2006) or the threshold concepts of Eckerdal et al. (2006). Relating those basic concepts to education leads to something such as the “Trucs” of Meyer (2006) and Pedroni (2009). By adding teaching implications that do not correspond to a specific course, something similar to the fundamental ideas of Schwill (1994) arise. The fundamental ideas shall enable the students to face changes in the subject of computer science where paradigm changes are quite common.

> "Only these fundamentals seem to remain valid in the long term and enable students to acquire new concepts successfully during their professional career in that these concepts will often appear to be just further developments or variants of subjects already familiar and then are accessible more easily using ideas learned before." (Schwill 1994, p. 1)

A set of observable learning objectives such as those of Anderson and Krathwohl (2009) are provided in national and international standards. Here, two candidates are presented that focus on the representation of object orientation, and especially object-oriented programming.

Besides the representation of object orientation in educational standards on computer science, their implementation in national and international curricula is of interest. As stated in Section 2.1, development of an object-oriented paradigm started in industry and it took a long time until it started to have an influence in education. A first try was published by Pugh et al. (1987). They investigated the benefits of including object-oriented concepts in existing courses. In this work of the late 1980s, object orientation was still only an additional approach for the present paradigm. A few years later Meyer (1993) suggested starting with object orientation early in the curriculum and providing the other approaches afterwards. In the following subsections, the most extensive international curriculum in computer-science education, published by an ACM/IEEE joint task force (ACM/IEEE-CS Joint Task Force on Computing Curricula 2013), and the implementation of object orientation are presented. Furthermore, a representative of a university curriculum for the bachelor degree in computer science is presented in this thesis. As the experiments presented in Chapter 7 only took place at the TU München, that curriculum was chosen (Technische Univerisät München 2014). In addition, the focus lies on the representation of object orientation in secondary education. For that purpose both a national and an international representative of computer-science educational standards are scrutinized. Furthermore, the general guidelines for the Abitur in computer science (Einheitliche Prüfungsanforderungen (EPA) Informatik) (Ständige Kultuministerkonferenz 2004)) of the Kultusministerkonferenz (KMK) and two national curricula for computer science in secondary education are analyzed. First, the Bavarian curriculum (Bayerisches Staatsministerium für Unterricht und Kultus 2004) was chosen because there is a compulsory subject in computer science. Second, the curriculum of North Rhine-Westphalia (Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen 2014) was chosen because it is one of the most current curricula and the state has the most citizens in Germany.

5.3.1 The ACM/IEEE Joint Task Force Computer-Science Curriculum

The last volume for the computer-science curriculum was published in 2001. A revision of this version was published in 2008. Based on this new version, a complete new volume of guidelines was developed during the last years with several drafts. The final version was published in 2013. Similar to the volumes before, the new volume has a revised body of knowledge and introduces several examples for courses that are implementing the curriculum.

There are 18 knowledge areas that form the knowledge body of the curriculum. These knowledge areas are not supposed to form particular courses in a curriculum. More precisely, most courses will pick topics from several knowledge areas. To provide advice on which topics should be covered by all students and which topics should be reserved for special courses, the topics are identified by a certain level. They are either "*core*" or "*elective*" and if they are core elements they can be "*tier1*" or "*tier2*". If they are *tier1 core*, they should be covered by all students. *Tier2 core* should be covered by almost every student and additionally the elective elements should be provided, but do not have to be covered by all students. Nevertheless, only covering the core elements is not sufficient for an undergraduate degree.

Besides the knowledge areas, a sub categorization is conducted. The knowledge units provide topics and learning outcomes, which have certain mastery levels that are borrowed from Bloom's taxonomy. These mastery levels are: familiarity, usage, and assessment (cf. ACM/IEEE-CS Joint Task Force on Computing Curricula 2013).

The coverage of object orientation in the knowledge areas or knowledge units can be seen in the following lists. The first list displays the topics related to object orientation and the corresponding knowledge areas and knowledge units. The bold topics are the knowledge units with the corresponding knowledge areas. Each item of the list is accompanied by the page number – in ACM/IEEE-CS Joint Task Force on Computing Curricula (2013) – in parentheses. Additionally, the level is added in square brackets and for the learning outcomes the mastery levels are supplied.

Information Management/Data Modeling

- Object-oriented models [Core-Tier2] (p. 114)

Programming Languages/Object-Oriented Programming

- Object-oriented design [Core-Tier1] (p. 157)
  - Decomposition into objects carrying state and having behavior
  - Class-hierarchy design for modeling
- Definition of classes: fields, methods, and constructors [Core-Tier1] (p. 157)
- Subclasses, inheritance, and method overriding [Core-Tier1] (p. 157)
- Dynamic dispatch: definition of method-call [Core-Tier1] (p. 157)
- Subtyping [Core-Tier2] (p. 157)
  - Subtype polymorphism; implicit upcasts in typed languages
  - Notion of behavioral replacement: subtypes acting like supertypes
  - Relationship between subtyping and inheritance

- Object-oriented idioms for encapsulation [Core-Tier2] (p. 157)
  - Privacy and visibility of class members
  - Interfaces revealing only method signatures
  - Abstract base classes
- Using collection classes, iterators, and other common library components [Core-Tier2] (p. 157)

Programming Languages/Basic Type Systems

- Generic types (parametric polymorphism) [Core-Tier2] (p. 159)
  - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism

Programming Languages/Advanced Programming Constructs

- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods [Elective] (p. 163)

System Development Fundamentals/Algorithms and Design

- Fundamental design concepts and principles [Core-Tier1] (p. 169)
  - Abstraction
  - Encapsulation and information hiding

The second list displays the learning outcomes related to object orientation and the corresponding knowledge areas and knowledge units.

Information Management/Data Modeling

- Describe the main concepts of the OO model such as object identity, type constructors, encapsulation, inheritance, polymorphism, and versioning. [Core-Tier2, Familiarity] (p. 114)

Programming Languages/Object-Oriented Programming

- Design and implement a class. [Core-Tier1, Usage] (p. 157)
- Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Core-Tier1, Usage] (p. 157)
- Correctly reason about control flow in a program using dynamic dispatch. [Core-Tier1, Usage] (p. 157)
- Compare and contrast (1) the procedural/functional approach (defining a function for each operation with the function body providing a case for each data variant) and (2) the object-oriented approach (defining a class for each data variant with the class definition providing a method for each operation). Understand both as defining a matrix of operations and variants. [Core-Tier1, Assessment] (p. 157)

- Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Core-Tier2, Familiarity] (p. 157)
- Use object-oriented encapsulation mechanisms such as interfaces and private members. [Core-Tier2, Usage] (p. 157)

Programming Languages/Basic Type Systems

- Discuss the differences among generics, subtyping, and overloading. [Core-Tier2, Familiarity] (p. 159)

The curriculum covers nearly all facets of object orientation. Here, object-oriented programming is contained within the document. It is even a knowledge unit within the programming languages knowledge area. Nevertheless, the modeling aspect still has serious priority.

5.3.2 Curriculum of the Bachelor Degree in Computer Science at the Technische Universität München

In addition to the very general and broad curriculum of the ACM/IEEE Joint Task Force, a university curriculum and its implementation of object orientation is analyzed in this section. The setting of Technische Universität München (TUM) was chosen for the analysis and the mandatory courses were retained. In general, only the bachelor degree has such courses. The master's degree consists only of elective courses. The bachelor curriculum of TUM contains 11 mandatory courses that are distributed over six semesters. Each course is described within the curriculum with its content and definitions of the corresponding learning outcomes.

The first course containing object-oriented concepts is the "Introduction into Computer Science 1" (IN0001). Here, **objects, classes, methods, inheritance, abstraction**, and **polymorphism** are introduced. The participants are enabled to solve algorithmic problems, and simple distributed and concurrent programs in Java or a similar object-oriented programming language.

In addition to IN0001, there is a practical course "Fundamentals of Programming (Exercises & Laboratory)" (IN0002), which is based on the concepts of IN0001. Furthermore, the participants are enabled to get along with Java or a similar object-oriented programming language. They can develop small applications on their own using basic computer-science concepts.

All the other courses cover non-object-oriented concepts of computer science. Interestingly, TUM introduces object-oriented notions at the very beginning. Nevertheless, a more detailed view on the course materials shows that the introduction process follows an objects-later or at least an OOP-later approach according to Section 5.1.1.

5.3.3 General Assessment Guidelines (EPA) in Computer Science

After having looked at a representative of higher education, secondary education will be addressed next. The first investigated document is the general assessment guideline (Einheitliche Prüfungsanforderungen (EPA) Informatik) for the A-levels. It was conducted by the German Kultusministerkonferenz (KMK) in 1989 and revised in 2004. Basically, the EPA consists of two parts. The first part comprises the written and oral exams. The first part also contains a brief description of the observable learning objectives that form the basis for the exams. The second part provides several task examples.

The only object-oriented observable learning objectives are related to the modeling aspects. Nevertheless, there are competencies such as the ability to implement graphical user interfaces that focus on object-oriented programming. The provided task examples clarify the needed concepts. Again, the basic concepts of **object, class, attribute**, and **method** are emphasized. A concrete definition of object-oriented programming competencies is missing.

5.3.4 Educational Standards of the Computer Science Teachers Association (CSTA)

An international approach for defining a standard for computer science education is provided by the Computer Science Teachers Association (CSTA), which is associated with the Association for Computing Machinery (ACM) (The CSTA Standards Task Force 2011). Over a couple of years, a set of competencies was defined that should be included in the teaching of computer science. In contrast to the national standards of the German Society for Computer Science (Gesellschaft für Informatik; GI), the CSTA standards address all school levels including the primary school. The goal of the standards is "to be coherent and comprehensible to teachers, administrators, and policymakers" (The CSTA Standards Task Force 2011, p. 1).

The standards are organized in two dimensions. The first dimension distinguishes the students based on their age level. Three different age levels are defined. The first level starts with elementary school students who should be introduced to fundamental concepts. Grades six to nine are grouped in level 2. They "begin using computational thinking as a problem-solving tool" and "begin to appreciate the ubiquity of computing and the ways in which computer science facilitates communication and collaboration" (The CSTA Standards Task Force 2011, p. 8). The third level (grades 9 to 12) focuses on the application of concepts and the creation of real-world solutions.

The second dimension defines content areas that are called strands. In the standards five strands are mentioned: computational thinking, collaboration, computing practice, computers and communication devices, and community, global, and ethical impacts. An overview can be seen in Figure 5.2.

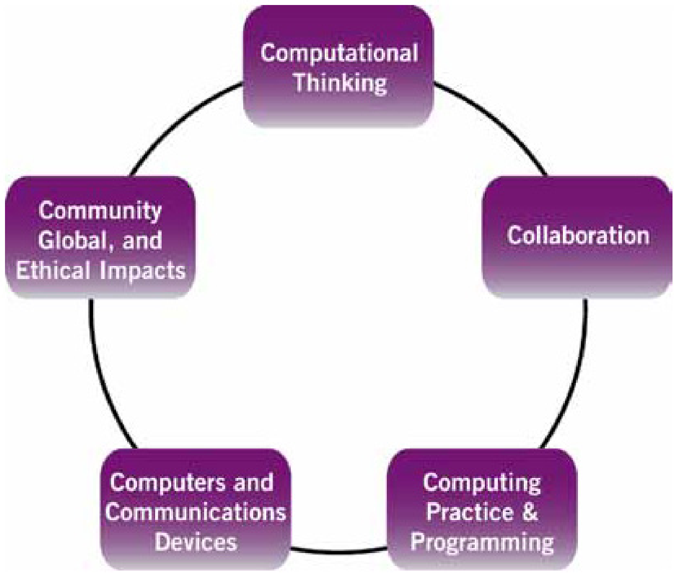

Computational Thinking
Collaboration
Computing Practice & Programming
Computers and Communications Devices
Community Global, and Ethical Impacts

Figure 5.2: Overview on the computer science strands of the CSTA standards (The CSTA Standards Task Force 2011, p. 10)

There are observable objectives formulated as a competency facet for every strand and level. The objectives directly related to object orientation are mentioned below. Thus, the lists contain the objectives for each level according to the definition above.

For the very early students of level 1, there are no objectives that are related to object orientation.

The students of level 2 will be able to:

- "Analyze the degree to which a computer model accurately represents the real world." (p. 16)
- "Understand the notion of hierarchy and abstraction in computing including high-level languages, translation, instruction set, and logic circuits." (p. 16)

These objectives are related in an abstract manner. They are associated with computational thinking in general. However, as abstraction is one of the key concepts of object orientation (see Section 2.2), it is mentioned in this section assuming a relation to this concept in both objectives.

The students of level 3 will be able to:

- "Use predefined functions and parameters, classes and methods to divide a complex problem into simpler parts." (p. 18)
- "Discuss the value of abstraction to manage problem complexity." (p. 18)
- "Decompose a problem by defining new functions and classes." (p. 21)

- “Use tools of abstraction to decompose a large-scale computational problem (e.g., procedural abstraction, object-oriented design, functional design).” (p. 21)

In addition to definitions of the standards given by the CSTA (The CSTA Standards Task Force 2011), a lot of examples are given to illustrate activities for applying the standards in schools.

Generally, the standards do not scope object orientation in a specific way. Only parts of computational thinking can be related to some concepts of object orientation. Similar for the national standards, the only aspect of object orientation is the modeling one. Object-oriented programming is, again, only a tool that is not mentioned at all. Only in the activities and the extra courses added to the third level are the programming issues a topic.

5.3.5 Educational Standards of the German Society for Computer Science (GI)

The educational standards for computer science in secondary education were published in 2008 by the German Society for Computer Science (Gesellschaft für Informatik (GI) 2008). The standards were developed to ensure up to date and accurate education of computer science in schools, and to address computer-science teachers, as well as administrative deciders and teachers’ instructors. Development of the standards is based on national and international educational standards in other subjects. Additionally, general teaching principles are taken into account. Originally, the standards were published in German. If not noted, the English translations are taken from (Brinda et al. 2009).

The GI standards are divided into two main parts. The first part covers the concepts, while the second part includes processes applicable to computer science. The content section itself is again divided into five sections: information and data, algorithms, languages and automata, informatics systems, and finally informatics, man and society. The processes are also divided into five sections: model and implement, reason and evaluate, structure and interrelate, communicate and cooperate, and represent and interpret. The sections are related pairwise, as presented in Figure 5.3.

According to the dimensional separation of computer science, the observable learning objectives are listed for each section; first, some objectives for all levels, followed by some that are only for levels five to seven and eight to ten.

As this thesis addresses the concepts around object orientation, only the objectives related to this topic will be presented.

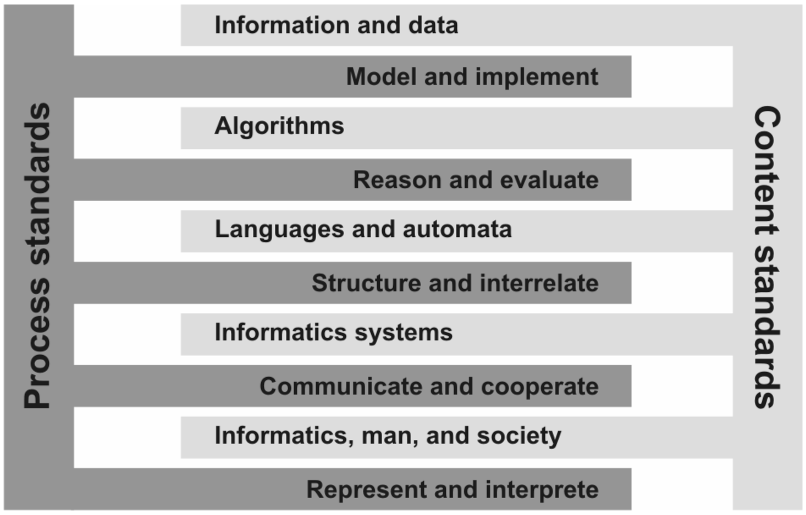


Figure 5.3: Overview of the concepts and process dimensions of the educational standards for computer science of the GI, (Brinda et al. 2009, p. 289)

In the content dimension there is no objective directly related to object orientation, which should be observable at all levels. But, in levels five to seven the students

- understand the terms “class”, “object”, “attribute”, and “attribute value” and apply them[10].
- understand the manipulation capabilities for attribute values of objects in age-appropriate applications and reflect how they support the presentation of information[10].
- represent the objects of the respective application in a suitable form[10].

Again, the process dimension does not contain objectives for all levels. In contrast to the content dimension, there are objectives for both levels that are presented in the following two lists. The first list covers the levels five to seven, while the second list covers levels eight to ten.

The students:

- identify objects in informatics systems and identify attributes and their values[10].
- create charts and graphs to illustrate simple associations between real-world objects[10].

The students:

- develop object-oriented models for simple issues and represent them with class diagrams[10].

[10]Translated by the author from German. The original source is listed in Appendix C

Besides the formulation of the observable learning objectives, in the last part of the GI standards a variety of examples for each section are given. Additionally, the intentions underlying the objectives are explained in a detailed way.

In total, the focus of object orientation in these standards lies on the modeling aspects. Implementation of these models is excluded in the text. All programming aspects in the standards are related to algorithm implementation.

5.3.6 Curricula of German Grammar Schools

In addition to the educational documents previously presented, examples of national secondary school curricula are presented. In Bavarian Gymnasien there is a compulsory subject for computer science. Additionally, North Rhine-Westphalia, the state with the most citizens, is included in the investigation.

5.3.6.1 The Grammar School in North Rhine-Westphalia

The curriculum for secondary education in North Rhine-Westphalia (NRW) is one of the most current. It is oriented on competencies. In contrast to the Bavarian curriculum described in the Section 5.3.6.2, computer science in NRW is not a mandatory subject, but an elective subject in higher secondary education. Basically, the curriculum defines an overall subject competency that can be divided into two parts. First, there are competency areas that contain the basic dimensions of the processes in the subject. Furthermore, there are areas of knowledge that define the basic concepts. These two facets lead to the competency expectations that are based on observable applications. The competency and knowledge areas are displayed in Figure 5.4.

The curriculum contains two competency levels. The first level is called the introduction stage (Einführungsphase). After gained competencies have been developed, the students enter a new level, the qualification stage (Qualifikationsphase), where the former competencies should be improved and new ones gathered. The knowledge and competency areas stay the same.

In the introduction stage only the knowledge area “data and their structure” contains the basic concepts of object orientation. Similar to all of the other documents investigated, the modeling aspects are the focus. Nevertheless, implementation of classes in a non-specified programming language is a competency that students should gather. Additionally, they should be able to use predefined documented class libraries. The complete knowledge area is displayed in the following list. The competency areas are added by an abbreviation in parentheses for argue (A), model (M), implement (I), represent and interpret (D), and communicate and cooperate (K).

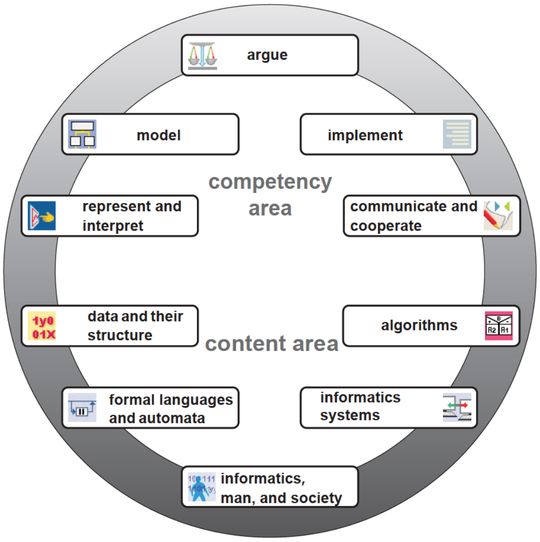


Figure 5.4: Overview of the competency and content areas of the NRW curriculum in computer science at higher secondary education facilities – Translated by the author from German. The original source is listed in Appendix C

The students

- identify objects, their properties, their operations and their relationships in the analysis of simple problems (M),
- model classes with their attributes, methods and associations (M),
- model classes using inheritance (M),
- assign simple data types, object types, or linear data structures to attributes, parameters, and return values of methods (M),
- assign a visibility to classes, attributes, and methods (M),
- represent the state of an object (D),
- graphically represent the communication between objects (M),
- represent classes, associations and inheritance in diagrams (D),
- document classes by description of the functionality of the methods (D),
- analyze and explain an object-oriented modeling (A),
- implement classes in a programming language including documented class libraries (I)[11].

[11]Translated by the author from German. The original source is listed in Appendix C

Within the qualification stage, object orientation is again only represented in the knowledge area “data and their structure”. Similar to the introductory stage, the focus is on modeling aspects. The following list presents the competencies of the subsection “objects and classes”. Those that are the same as above should be improved, while the others should be gathered for the first time.

The students

- identify objects, their properties, their operations and their associations in the analysis of problems (M),
- graphically represent linear and non-linear structures and explain them (D),
- model classes with their attributes, methods and associations specifying multiplicities (M),
- model abstract and non-abstract classes using inheritance by specializing and generalizing (M),
- assign simple data types, object types, and linear and nonlinear data structures to attributes, parameters, and return values of methods (M),
- use possibilities of polymorphism in the modeling of appropriate problems (M),
- assign a visibility to classes, attributes, and methods (M),
- represent classes and their associations in diagrams (D),
- document classes (D),
- analyze and explain object-oriented modeling (A),
- implement classes in a programming language including documented class libraries (I)[12].

5.3.6.2 The Bavarian Gymnasium

In 2004 the Bavarian state introduced computer science as a subject for all Gymnasium students in grades six and seven and for students in grades nine and ten who chose the science and technology track. The courses in computer science are elective courses for students in the final grades (cf. Hubwieser 2012).

The courses and the underlying curriculum are based on object-oriented modeling.

> “From the point of view of general education it seems that among all themes of informatics it is object oriented modelling that promises the most benefit for the students. Thus we chose it as the central theme of our course.” (Hubwieser 2006, p. 2)

[12]Translated by the author from German. The original source is listed in Appendix C

This leads to a partitioning of the learning process into three steps. In the first two grades, the general concepts of object orientation and especially of the modeling aspects are introduced. Therefore, the students model standard software documents in an object-oriented manner. In the middle two grades, students concentrate on modeling the real world by using database systems and others. Finally, the students should apply object-oriented programming to simulate their models. The final two grades are for specialization as they are only elective and address students with great interest in computer science. The implementation of this methodology can be seen in the corresponding textbooks; for example, Frey et al. 2004; Hubwieser 2007*c*, 2008*a*, 2009, 2010.

According to the ideas of Hubwieser (2012), a curriculum for the subject was conducted. Now, the topics and observable learning objectives mentioned in this curriculum, which can be found at the Bavarian state institute for school quality and educational research in Munich[13], will be presented.

In Table 5.2, parts of the curriculum (Bayerisches Staatsministerium für Unterricht und Kultus 2004) that are related to object orientation are presented. For a better orientation in the original text, the section number for each item is mentioned in the first column.

The topics of the curriculum cover the basic concepts of object orientation, as pointed out in Section 2.2. Besides conceptual knowledge, application of the concepts to examples from the context of the students is the main focus of the curriculum. At the end of grades nine to eleven the students apply their knowledge on a bigger project, mostly in teamwork.

Mühling et al. (2010) conducted an investigation on the application of the curriculum. Additionally, the main ideas and examples for the topics, especially those related to object-oriented programming, is described by Hubwieser (2006, 2007*a*).

Section	Text passage
NT 6.2.2	Considerations about the structure of graphs lead to the object-oriented perspective. The students recognize that each of the graphics objects has certain properties and is associated with a class of similar objects. • Objects of a vector graphic: attribute, attribute value, and method • Description of similar objects by class: line, rectangle, ellipse, text box

Table 5.2: Text passages related to object-oriented concepts in the curriculum of the Bavarian Gymnasium (Translated by the author from German. The original source is listed in Appendix C) - continued on next page

[13]http://www.isb.bayern.de - last access 10.12.2014

Section	Text passage
NT 6.2.3	The understanding of these concepts is deepened in practical work with word processing software; it is shown that individual objects can be associated with each other. The students recognize that many everyday connections can also be described by relations between objects, so these terms have a more general meaning. • Improve the representation of information by changing the appropriate attribute values • The contains-association between objects; design of object and class diagrams
NT 6.2.4	Various animations, such as presentation software provides for the design, help students to understand the principle of the method.
NT 6.2.5	They realize that hierarchical orders are enabled by the contains-association between objects of the same class. • Advanced application of the contains-association: folder containing folders
NT 7.2.1	The students will learn that content relationships between documents may lead to networked structures, for which a hierarchical representation is not enough. • The association 'refers to' between objects
Inf 9.2	They realize that the structure of classes and their associations can be represented very clearly in class diagrams. To take advantage of the model and check its usefulness, they implement it with a relational database system. • Object (entity), class, attribute and value range • Associations between classes, cardinality, graphical representation • Implementation of objects, classes, and associations in a relational database system: data set, table, range of values, concept of keys

Table 5.2: (contd.) Text passages related to object-oriented concepts in the curriculum of the Bavarian Gymnasium (Translated by the author from German. The original source is listed in Appendix C) - continued on next page

Section	Text passage
Inf 10.1.1	By using a suitable development environment for object-oriented modeling the students repeat and clarify the known terms and notations of the object-oriented perspective with simple examples. This will clarify that objects essentially represent a unit of attributes and methods. • Object as a combination of attributes and methods • Graphical representation of classes and objects, description of static associations through object or class diagrams
Inf 10.1.2	The students learn to describe the changes of objects using states and transitions, as well as to document with state diagrams. By implementing these state models in object-oriented programs, they set the states by values of attributes (variables) and assign method calls to the transitions. • State of objects: Determination by states of attributes, state transition by value assignment • Life cycle of objects from the instantiation to initialization to release
Inf 10.1.3	The students recognize that essential processes of a system are based on communication between its objects. For the full description, the dynamic associations between objects or classes have to be learned in addition to the already learned static ones. For this purpose, the young people get to know appropriate graphical notations and develop possibilities for the implementation of these associations in a programming language. • Communication between objects by calling methods; Interaction diagrams; Data encapsulation • Implementation of the contains-association, references to objects
Inf 10.2	Young people use hierarchical structures to order their realm of experience. They realize that these often can be represented through a special kind of association between the classes of a model. The students learn the concept of inheritance and apply it. In particular, they deal with the possibility of increasing specialization by changing inherited methods. • Generalization and specialization by super- or subclasses, representation in class diagrams, inheritance • Polymorphism and overriding methods

Table 5.2: (contd.) Text passages related to object-oriented concepts in the curriculum of the Bavarian Gymnasium (Translated by the author from German. The original source is listed in Appendix C) - continued on next page

Section	Text passage
Inf 11.1.1	A first implementation with an array quickly shows the limits of this static solution and leads the youth to a dynamic data structure such as the simply linked list. They learn its principle functionality, as well as the recursive structure and apply the concept of reference on objects. The youth realize that the recursive structure of the list suggests a recursive algorithm for many of its methods. They understand that a universal applicability of the class list is only possible if attention is paid on a clear separation of structure and data. • Implementation of a simply linked list using references by a suitable software pattern (composite); Implementation of the methods to add, find, and delete

Table 5.2: (contd.) Text passages related to object-oriented concepts in the curriculum of the Bavarian Gymnasium (Translated by the author from German. The original source is listed in Appendix C)

5.4 Summary

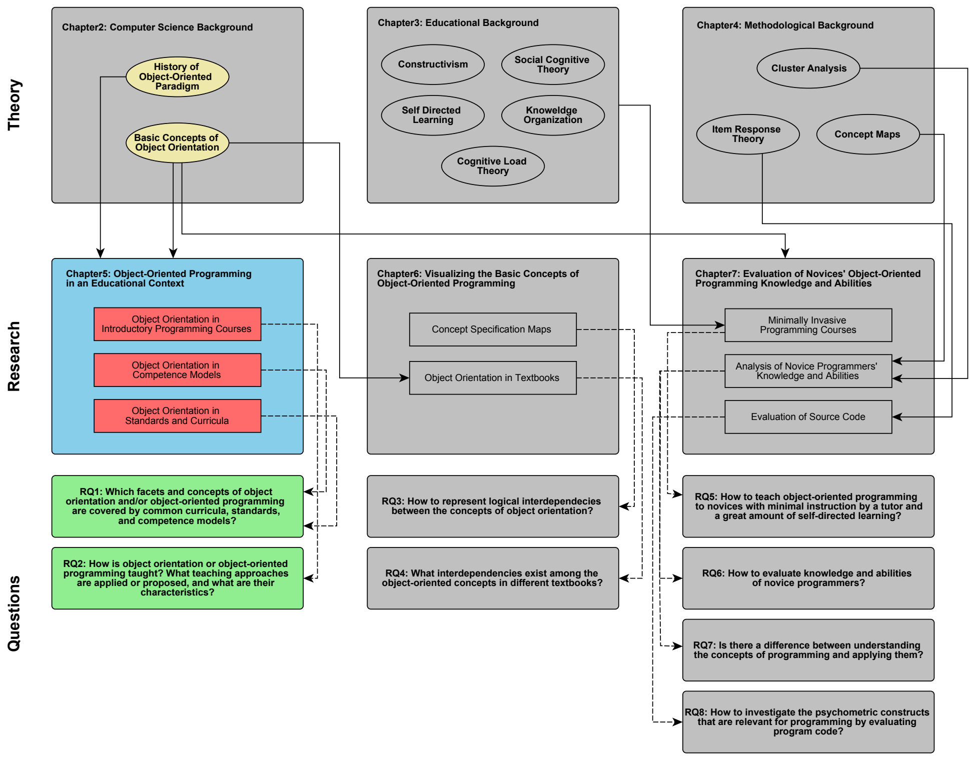
The last sections have shown the application of object-oriented notions in an educational context. Examination of the items on object orientation in national and international standards has shown that the programming aspect is of minor interest. The majority of the items cover the modeling aspects of object orientation.

National and international curricula are developed based on the standards for computer science. Again, the representation of object-oriented programming is investigated. The extensive curriculum of the ACM/IEEE joint task force includes object-oriented programming, as well as modeling. The national curricula of the Bavarian and North-Rhine Westphalian grammar schools are examples for concrete curricula in secondary education. Again, the modeling aspects are in the foreground, especially in the first grades. Nevertheless, the implementation aspects are introduced. All these analyses emphasize the modeling aspects. Object-oriented programming is only seen as a tool for implementing the corresponding models (see Section 5.3).

According to the curricula and standards, the competency models that are introduced first in computer science also have an emphasis in object-oriented modeling (see Section 5.2). In summary, common curricula, standards, and competency models cover the modeling aspects of object orientation in most cases. In contrast, object-oriented programming is only mentioned as a side effect or as a tool for implementing object-oriented models. The basic concepts of object orientation mentioned in Section 2.2 are covered in almost all investigated documents. This provides the answer to **RQ1**, which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competency models.

RQ2 inquires how object orientation or object-oriented programming is taught, and what teaching approaches are applied or proposed. The different educational approaches for introducing object orientation focus mainly on the programming aspects. First, there are the educational "paradigms" for the introductory courses. Mainly, there are two different types. The first type, "objects-first" and its derivations, handle object orientation as something different from the former programming paradigm and, because of that, introduce it independently at the very beginning of the course. The notions of object, attributes, and methods are the most important concepts. In the meantime, this approach is seen as state of the art in computer science education. On the other hand, there is the "classical" procedural or "objects-later" approach that regards object orientation as an extension of the procedural paradigm and, because of that, starts with the common imperative procedural notions (see Section 5.1.1).

The last and most controversial aspect of object orientation in an education context is the question of which language is the most suitable. The literature review conducted in Section 5.1.2 shows that there is a large amount of different languages available for an introductory course. The most suitable choice depends on the purpose the language has to fulfill. The setting of the course and its participants also have a strong influence. All kinds of languages including "industrial" languages, educational subsets, or languages especially designed for education have advantages and disadvantages. Which of them is the most appropriate for a given course cannot be generalized.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

6 Visualizing the Basic Concepts of Object-Oriented Programming

In this chapter, a graphical representation of the associations between concepts is introduced. These concept specification maps show the concepts related to a topic, on the one hand, and their relations among each other, on the other hand (**RQ3**).

Additionally, an application of these maps is shown. Therefore, an analysis of the textbooks from the introductory courses at several universities is presented (**RQ4**). The method of information visualization and the results of the analysis are also shown in (Berges and Hubwieser 2012, 2013).

6.1 Related Work

A broad study on textbooks was conducted by Raadt et al. (2005) who investigated the content of textbooks that are used in universities in Australia and New Zealand. They first conducted a pilot study to figure out which text features are important.

> "The content of each text was divided into chapter content, language reference, glossary, bibliography and index. The size of these elements was measured in complete pages. Within each chapter, the proportion occupied by exercises and examples can be separated from other chapter content. For the purposes of comparison, an example is seen as a complete, continuous item of source code. Exercises contained within a chapter and at the end of a chapter were measured together. Exercises range from reflective questions, which may consist of a single line of text, through to projects which may occupy several pages."(Raadt et al. 2005)

Furthermore, they investigated correspondence of the textbooks to the ACM/IEEE curriculum 2003.

Quite a new study conducted by Börstler et al. in 2009 focused on examples in textbooks. They examined a list of 13 textbooks that are commonly used in introductory courses. Besides investigating the examples, Börstler et al. (2009) built categories for the books by classifying them into "object-oriented" or "traditional" approaches.

Pedroni (2009) investigated the so called "Trucs" which were introduced by Meyer (2006). The *Testable, Reusable Units of Cognition* (Trucs) were defined for "teaching a class on the topic, writing a textbook or course notes, defining a standard curriculum, preparing exam questions, assessing job candidates' claims that they master the topic, and in general to compare and consolidate educators' understanding of the area"(Meyer 2006). The application of "Trucs" for teaching a class on a specific topic is the content of the thesis of Pedroni. She developed a graph for the dependencies between the "Trucs" and the parts of them, the "notions". On the one hand, these graphs are "a resource for students with programming experience, which supports them in relating preexisting knowledge to the course contents" (Pedroni 2009, pp. 60).

In addition to helping students, the dependencies between Trucs and notions support teachers "in designing their course by helping check prerequisite violations, identify missing topics,and adapt the teaching to common misconceptions of [the] novice programmer" (Pedroni 2009, p. 61).

Similar to the "Trucs" Steinert (2010) investigated learning objectives in computer science. He conducted an analysis of several educational sources like curricula or assessment tasks. For that reason he introduces a graphical representation of the included learning objectives. The learning objectives are separated into a cognitive and a knowledge dimension according to the taxonomy of Anderson and Krathwohl (2009). Furthermore, interdependencies between the learning objectives are illustrated. Learning objectives can be connected with a strong primacy relation, which expresses one objective to be prerequisite for the corresponding. Additionally, there are weak primacy relations, which express a didactical precedence (cf. Steinert 2010, p. 40).

6.2 Concept Specification Maps

A methodology to extract concepts and their interdependencies among each other from a certain text is presented here. For the purpose of obtaining an overview of the concepts related to object orientation, object-oriented programming, and object-oriented design, a graphical representation of the concepts and their interdependencies is introduced. These maps can be applied with any topic and their concepts and interdependencies extracted from any kind of literature, even course materials. A similar approach is presented in Section 6.1. However, the introduced methodology of the "Trucs and notions"-graph of Pedroni et al. (2008) is based on a normative basis of concepts that are related to each other by experts.

The first idea is to clarify the structure of definitions, as in educational literature there are only vague specifications rather than exact definitions of the kind: "One concept consists of or contains other concepts". The final version of the map is based on both kinds, the exact definitions and the vague specifications. Later in this chapter only "specifications," meaning exact definitions of the kind above, and softer specifications that are distributed though the text are spoken of.

In the map all definitions or specifications of a concept are symbolized by circles that are connected by a bold line to the specified concept, and by outgoing arrows to the specifying concepts. For better readability the items are enumerated. This allows for the referencing of nodes in the text. Furthermore, in most cases the items consist of more than a few words. The "connection-structure" expressed by the circle would have to be much bigger and the complete map would grow to an unreadable format.

A sample is shown in the following sentence and in Figure 6.1. It was extracted from a textbook during the analysis described below and was selected due to the fact that it is quite simple in its structure.

> (12) “A class is a collection of fields that hold values and methods that operate on those values.” (Flanagan 2005)

The concepts *field* and *method* are specifying the concept *class*. The number (12) in the circle in Figure 6.1 corresponds to the one in parenthesis in the specification above.

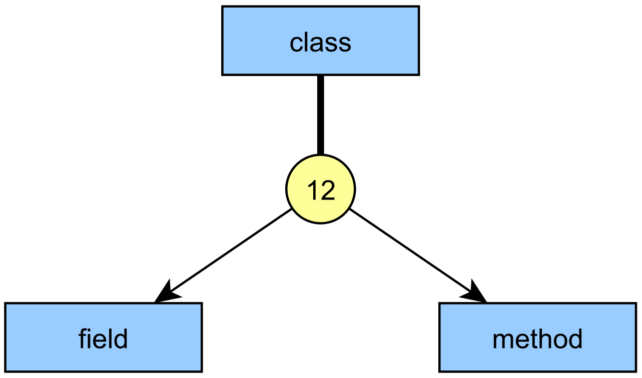


Figure 6.1: Example of a specification node (12) with one specified concept (class) and two specifying concepts (field and method)

In Section 2.2, the “quarks” of object orientation that were identified by Armstrong (2006) built the skeleton of concepts related to object orientation. As mentioned, she found eight concepts, which she defined as the “quarks” of object orientation: *inheritance, object, class, encapsulation, method, message passing, polymorphism*, and *abstraction*. For all these concepts there is a short section where several definitions are shown. Each of these sections ends with a combined definition of the given concept. The text analysis of the “quarks” definitions (cf. Armstrong 2006) is shown in the list below as a sample for the process of building the map. The specified concept is in bold letters and the specifying concepts are in italic typeset. The numbers that identify the specification later in the graphical representation are added in parentheses at the end of each specification. As the specifications are embedded in the text, a beginning is added for each sentence and indicated in square brackets. For reasons of formatting and clarifying the elements, no citations for the statements are used, although they are cited from Armstrong (2006).

> [**Inheritance** is] a mechanism that allows the *data* and behavior of one *class* to be included in or used as the basis for another class. (1)
>
> [An **object** is] an individual, identifiable item, either real or abstract, which contains *data* about itself and descriptions of its manipulations of the data. (2)
>
> [A **class** is] a description of the organization and actions shared by one or more similar *objects*. (3)
>
> [**Encapsulation** is] a technique for designing *classes* and *objects* that restricts access to the *data* and behavior by defining a limited set of *messages* that an object of that class can receive. (4)
>
> [A **method** is] a way to access, set or manipulate an *object*'s information. (5)
>
> [**Message passing** is] the process by which an *object* sends *data* to another object or asks the other object to invoke a *method*. (6)
>
> [**Polymorphism** is defined as] the ability of different *classes* to respond to the same *message* and each implement the *method* appropriately. (7)
>
> [**Abstraction** is] the act of creating *classes* to simplify aspects of reality using distinctions inherent to the problem. (8)

These definitions were used to build a sample map, which is shown in Figure 6.2.

The maps are structured in a hierarchical manner starting with the concept *inheritance*. For the design of the maps, the yEd-editor[14] was used. It has the ability to structure the graph with a breadth-first search, starting with the first concept such as *object-oriented programming and design* or as in the example with *inheritance*.

As the maps illustrate concepts and their specifications and relation to each other in a text, they are called concept specification maps (CSM).

[14] http://www.yworks.com - last access 10.12.2014

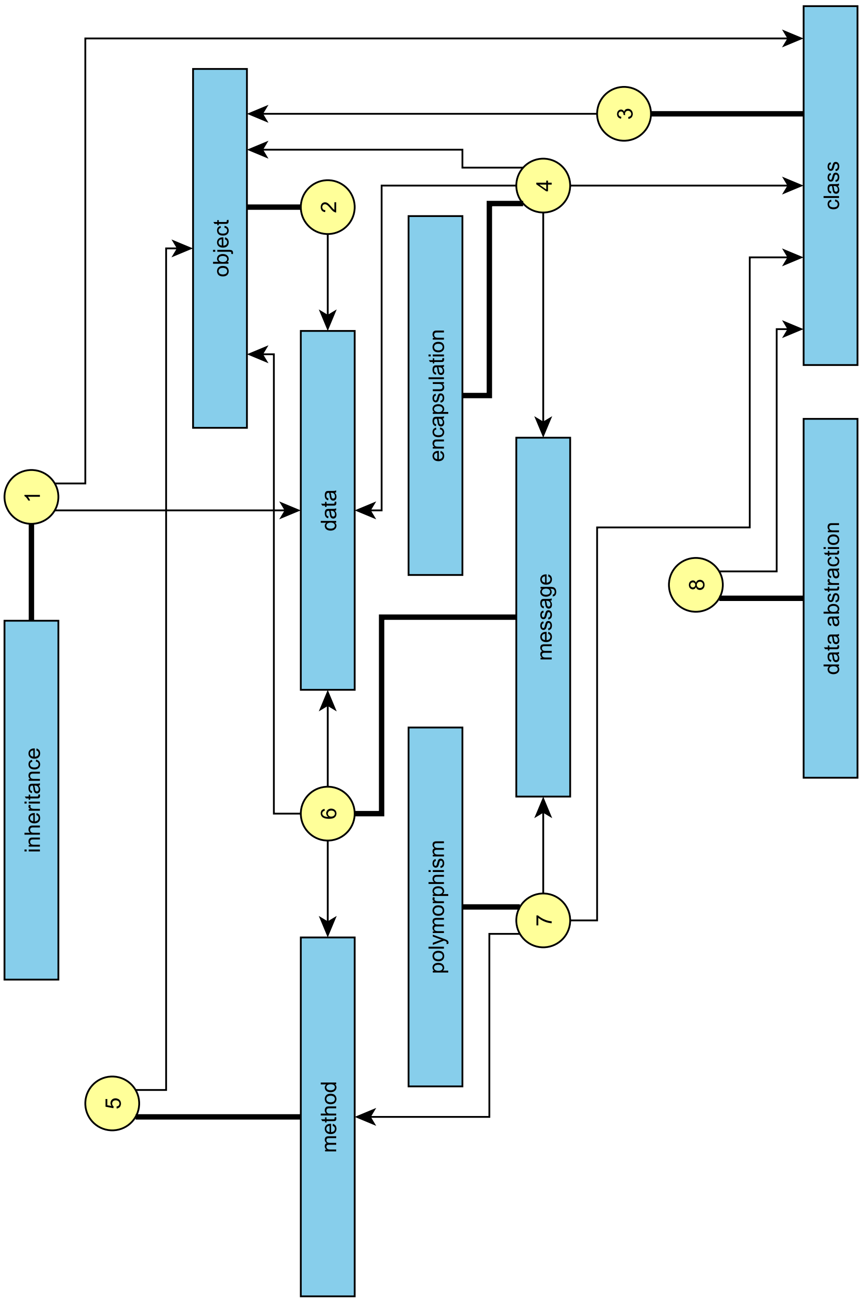


Figure 6.2: Sample concept specification map of the “quarks” definitions by Armstrong (2006) – arrows: “specifying” - bold line: “specified”

6.3 Concept Specification Maps of Textbooks

The ideas presented above led to the graphical representation of the concepts included in textbooks. In contrast to the textbook analysis presented in Section 6.1, the maps should be as simple as possible. Because of that the presented investigation concentrates on the text and did not include any information on the didactic that could be included in the text. Due to the topic of this thesis, the investigation focuses on concepts related to object-oriented programming and design in computer science education although, of course, the methodology is applicable to any topic in computer science and beyond.

The first step for creating the maps is the extraction of text passages from the selected literature. Therefore, a simple algorithm is applied for the detection of given concepts. First, the given text is divided into sentences by splitting the text at each full stop. In an automatic step all those sentences with a given word are extracted. Additionally, two sentences in front of and behind it are added to one text passage. The given concept is highlighted with a bold typeset.

In a second step that has to be done manually, all the text passages are rated. They are either “relevant” (1) or “irrelevant” (0) for the further analysis. A text passage is rated as 1 if the given concept is specified in a closer way within it. That means there is either a concrete definition of the given concept or there is a description or further explanation of the concept. For example, all parts that are necessary for a concept are described.

Table 6.1 shows a few examples of rated text passages. The rating is added to each item in the second column. The specified concept is written in capital letters and according to the automatic step in **bold** typeset. The specifying concepts are in *italic* typeset and the main sentence is in `typewriter` typeset. In the conducted textbook analysis, the concepts are extracted into separate columns, but here are included and emphasized by a different typeset due to readability reasons.

Text passage	Rating
These difficulties become even greater when we allow the possibility of concurrent execution of programs. The stream approach can be most fully exploited when we decouple simulated time in our model from the order of the events that take place in the computer during evaluation. We will accomplish this using a technique known as delayed evaluation. `We ordinarily view the world as populated by independent OBJECTS, each of which has a` *`state`* `that changes over time.` An object is said to "have state" if its behavior is influenced by its history. A bank account, for example, has state in that the answer to the question "Can I withdraw $ 100 ?" depends upon the history of deposit and withdrawal transactions. (Abelson et al. 1996)	1
Java programs are written by combining new methods and classes that you write with predefined methods and classes available in the Java Application Programming Interface (also referred to as the Java API or Java class library) and in various other class libraries. Related classes are typically grouped into packages so that they can be imported into programs and reused. `You 'll learn how to group your own CLASSES into packages in Chapter 8.` The Java API provides a rich collection of predefined classes that contain methods for performing common mathematical calculations, string manipulations, character manipulations, input/output operations, database operations, networking operations, file processing, error checking and many other useful operations. Familiarize yourself with the rich collection of classes and methods provided by the Java API (java.sun.com/javase/6/docs/api/). (Deitel and Deitel 2012)	0
These problems will likely be solved because the potential value of increased software reuse is enormous. *Classes* normally hide the details of their implementation from their clients. `This is called INFORMATION HIDING.` As an example, let us consider the stack data structure introduced in Section 6.6. Recall that a stack is a last-in , first-out (LIFO) data structure – the last item pushed (inserted) on the stack is the first item popped (removed) from the stack. (Deitel and Deitel 2012)	1

Table 6.1: Sample text passages with the rating (specifying text passage: 1 - otherwise: 0) and corresponding concepts marked by a different typeset (specified concept: bold - specifying concepts: italic)

For selecting the concepts there are two different options. First, if the concepts are known before the start of the analysis, all relevant concepts can be analyzed at once. The nodes of the concept specification maps are then known before and the analysis concentrates on the edges in the map. A variation of this method is to include concepts that are connected to the given concepts, but are not analyzed in the investigation. This variation has been conduced on the “quarks” in the sample at the beginning of this chapter.

The second methodology is an iterative procedure. The analysis starts with an initial concept such as *object orientation* or *inheritance*. In the first step only the text passages related to this concept are analyzed. During the investigation the list of specifying concepts lead to the new items to be analyzed. The procedure ends either when there is a kind of saturation or if an a-priori defined level is reached. For example, the concepts are limited to a specific programming paradigm such as *object-oriented programming*. The procedure would finish with the concepts that are necessary for *object-oriented programming*, but are related to the procedural paradigm. Which method is chosen depends on the goals of the investigations. This methodology has the advantage that the concepts included in the map are all related to the initial concept. Nevertheless, there is also a serious disadvantage as concepts related to the topic in general, but not directly connected to the initial concept, are not found and are, therefore, excluded from the analysis.

After analyzing all text passages, a list of the relevant elements (those that contain either a specifying or a specified concept) and the corresponding concepts are created. Additionally, all items are ordered by the specified concept and enumerated. In the last step of data gathering, all specifications with the same concept list are combined to one item. For this purpose the reduced text passages are concatenated and the items in the table are renumbered.

Then, in a final step, the concept specification map is created on the basis of the gathered list. Each item in the final table becomes a “specification node” and the concepts are added as “concept nodes”. The edges are drawn based on the type of concepts they connect to the specification.

6.4 Object Orientation in Textbooks of Introductory Courses

Due to the fact that in recent years nearly every introductory course addresses object orientation, textbooks regarding the concepts of object orientation and object-oriented programming that are recommended in those introductory courses are analyzed. To get a normative base for the comparison, courses are selected that provide a first contact to object-oriented programming for their participants. Every course recommends a list of textbooks to help the students get into the field of programming. As Armstrong stated in her paper on the “quarks” of object orientation, one “reason that learning OO is so difficult may be that we do not yet thoroughly understand the fundamental concepts that define the OO approach”(Armstrong 2006, p. 123).

Besides the differences in teaching methodologies and the different aspects of object orientation in computer science education (see Section 5.1), textbooks "are an important component of teaching introductory programming. They are a major source of example programs and also work as a reference for how to solve specific problems" (Börstler et al. 2009, p. 127).

For extracting concepts from textbooks, a representative list of books that are recommended and used in introductory courses has to be built. To avoid arbitrariness in the selection of the introductory courses, first the universities are selected on a normative base. In the first attempt the search focused on the nine technical universities of Germany, namely *RWTH Aachen, TU Berlin, TU Braunschweig, TU Darmstadt, TU Dresden, Universität Hannover, Universität Karlsruhe, TU München and the TU Stuttgart*. For the selection of the textbooks, the introductory courses on object-oriented programming, which provide the first contact to the object-oriented paradigm, are analyzed. All textbooks that are recommended by the course leaders are listed in Table 6.2. The number in the last column is the total number of recommendations within the courses. The list contains German books, as well as English books.

To be able to compare the results with international universities, the German books are excluded and a list is formed with those that are either written in English or have at least an English translation. The list concludes with only three books that are recommended in at least two universities.

Due to this small sample size, the list of universities were expanded and, therefore, more introductory courses are included in the investigation. For that reason, the QS World University Ranking 2011[15] was taken as a basis for the university list. The "top-ten" of the ranking for Engineering and Technology faculties were added. The final list of all 19 universities is shown in Table 6.3. They are in alphabetical order.

Following the analysis of all the introductory courses of the resulting 19 universities, the result is a list of 19 textbooks, which are shown in Table 6.4. The first five books, highlighted in a gray color, are recommended by more than one introductory course and are, therefore, used in the investigation.

For the analysis of the textbooks, the methodology chosen is incremental and finishes at a previously defined level for creating the concept specification maps. For every book, the three concepts of *object orientation*, *object-oriented programming*, and *object-oriented design* are combined to one common starting point in each map. From this initial concept, all concepts are investigated that belong to the object-oriented paradigm. The decision whether a concept belongs to the paradigm or not is made on the basis of the theoretical background described in Section 2.2. With the results of the analysis partly presented in (Berges and Hubwieser 2013), an example of interpreting the constructed concept specification maps (CSM) is given. The complete maps can be seen in Appendix A. There are different kinds of results. First, there are results based on the pure analysis of the textbooks, statistical values, as well as an analysis of the structure of the produced concept specification maps. A further set of results is based on the comparison of the textbook analysis with the analysis of the "quarks" that was conducted as an example when introducing the maps in general.

[15] http://www.topuniversities.com - last access 10.12.2014

Author	Title	#
Eckel, B.	Thinking in Java	3
Abelson, H. and Sussman, G.J.	Structure and Interpretation of Computer Programs	2
Sedgewick, R. and Wanye, K.	Introduction to programming in Java	2
Ullenboom, Ch.	Java ist auch eine Insel	2
Krüger, G.	Javabuch	2
Schiedermeier, R.	Programmieren in Java	2
Deitel, H. and Deitel, P.	How to program Java	1
Flanagan, D.	Java in a Nutshell	1
Bishop, J.	Java gently	1
Bloch, J.	Effective Java	1
Felleisen, M.	How to Design Programs	1
Sebesta, R.W.	Concepts of Programming Languages	1
Mitchell, J.C.	Concepts in Programming Languages	1
Broy, M.	Informatik1. Programmierung und Rechnerstrukturen	1
Gumm, H.P. and Sommer, M.	Einführung in die Informatik	1
Pepper, P.	Programmieren Lernen	1
Ratz, D. and Scheffler, J.	Grundkurs Programmieren in Java	1
Echtle, K. and Goedicke, M.	Lehrbuch der Programmierung mit Java	1
Mössenböck, H.	Sprechen Sie Java?	1
Küchlin, W. and Weber, A.	Einführung in die Informatik	1
Doberkat, E.-E. and Dißmann, S.	Einführung in die objektorientierte Programmierung mit Java	1

Table 6.2: Textbooks recommended in introductory courses at the German technical universities

Name	Country
RWTH Aachen (RWTH)	GER
University of California, Berkeley (UCB)	USA
TU Berlin (TUB)	GER
TU Braunschweig (TUBS)	GER
Massachusetts Institute of Technology (MIT)	USA
University of Cambridge (CAM)	UK
TU Darmstadt (TUD)	GER
TU Dresden (TUDR)	GER
Leibniz Universität Hannover (LUH)	GER
Karlsruhe Institute of Technology (KIT)	GER
Imperial College London (ICL)	UK
TU München (TUM)	GER
California Institute of Technology (Caltech)	USA
Tsinghua University (TU)	CN
National University of Singapore (NUS)	SG
Stanford University (STAN)	USA
Universität Stuttgart (UST)	GER
The University of Tokyo (UT)	JP
ETH Zurich (ETH)	CH

Table 6.3: Selected universities as the base for the textbook analysis

Author	Title	#
Eckel, B.	Thinking in Java (Eckel 2006)	4
Abelson, H. and Sussman, G.J.	Structure and Interpretation of Computer Programs (Abelson et al. 1996)	2
Sedgewick, R. and Wayne K.	Introduction to programming in Java (Sedgewick and Wayne 2008)	2
Deitel, H. and Deitel, P.	How to Program Java (Deitel and Deitel 2012)	2
Flanagan, D.	Java in a Nutshell (Flanagan 2005)	2
Bishop, J.	Java gently	1
Bloch, J.	Effective Java	1
Felleisen	How to design Programs	1
Sebesta, R. W.	Concepts of Programming Languages	1
Mitchell, J. C.	Concepts in Programming Languages	1
Roberts, E.	The Art and Science of Java	1
Stroustrup, B.	The Design and Evolution of C++	1
Meyers, S.	Effective C++	1
Meyers, S.	Effective STL	1
Meyers, S.	More Effective C++	1
Gamma, E., Helm,R. , Johnson, R. & Vlissides, A.	Design patterns: elements of reusable object-oriented software	1
Bloch, J. & Gafter, N.	Java puzzlers	1
Barnard, D.T; Holt, R.C. and Hume J.N.P.	Data structures: an object-oriented approach	1
Meyer, B.	Touch of class	1

Table 6.4: Final list of textbooks that are used by or recommended to the students in the investigated introductory programming courses

6.4.1 Structures in Introductory Textbooks

As mentioned above, text passages are extracted from the books and rated with 1 if they contain a specification of a particular concept. The first concept investigated in each book is “object-oriented programming” combined with “object-oriented design”. Overall, 36460 text passages were examined. The ratio of 1-rated elements to all passages is 0.4% to 0.7%. A statistical overview of all five books can be seen in Table 6.5.

Book	# text passages	# 1-rated elements	1-rated frequency
Abelson et al. (1996)	839	4	0.5%
Deitel and Deitel (2012)	18726	69	0.4%
Eckel (2006)	9222	35	0.4%
Flanagan (2005)	5729	37	0.6%
Sedgewick and Wayne (2008)	1944	13	0.7%

Table 6.5: Statistical overview (number of text passages, number of 1-rated elements, frequency of 1-rated items) on all five books

The second column displays the absolute number of text passages investigated for a book. This number depends on the structure of the book related to the concepts of object orientation. As the incremental methodology for creating the concept specification map is chosen, the number of investigated text passages is related to the number of concepts that are used to specify object orientation. It is possible that some concepts are not involved in the investigation due to the fact that they are not directly related to object orientation.

The third column displays the absolute number of 1-rated elements, while the relative numbers are in the last column. They are independent from the detailedness of the concepts around object orientation, but are a measure of the amount of examples, language-related text passages, or how compact the specifications are.

As can be seen in Table 6.5, the ratio of 1-rated text passages is almost the same for all books. Despite that, the number of text passages differs a lot. This may result from the fact that this analysis was started with the concept *object-oriented programming and design*. If a book specifies a lot of concepts that are related to object orientation, a lot more text passages have to be analyzed. Therefore, the total number of text passages can be a first indicator of how much object orientation is in a book. This is seen in the books that were investigated. The book by Abelson et al. (1996) is known to be a book that is addicted to the imperative procedural paradigm, whereas the book of Deitel and Deitel (2012) is related to the object-oriented paradigm (cf. Börstler et al. 2009).

Besides the simple statistics on the amount of investigated text passages, the concept specification maps give an overview on the structure of the books related to a given topic. In this case, the structure of object orientation is outlined. Table 6.6 displays which concepts are referred to in order to specify the term “object-oriented programming and design”. Therefore, the nodes that are connected directly to the node with this concept had to be analyzed. In Table 6.6 the first column contains the author by which the book is identified and the second column shows all of the relevant concepts.

Book	Concepts
Abelson et al. (1996)	class, inheritance, state
Deitel and Deitel (2012)	class, composition, encapsulation, inheritance, object, polymorphism, visibility
Eckel (2006)	class, composition, data abstraction, inheritance, message, object, polymorphism, reference
Flanagan (2005)	class, encapsulation, field, information hiding, inheritance, method, object
Sedgewick and Wayne (2008)	object, reference

Table 6.6: Concepts defining “object-oriented programming and design” directly

As a specification in the sense of the concept specification maps does not mean that there is one explicit definition with all the given concepts in it, the nodes that are directly connected to a node are of special interest. Most of the specifications examined in this investigation were collected from all through the textbook. This means that there is more than one chapter or section in the textbook that addresses the given topic. If we assume that students learn the concepts of object-oriented programming and design with the selected textbook, the directly connected concepts also have to be well understood.

In selecting a book for an introductory course to object orientation, the concepts related to object orientation are important. The different educational methodologies for introducing object orientation need several concepts in a different order for teaching (see Section 5.1.1). The dependencies resulting from the chosen methodology and selected concepts can be found in the concept specification map of the recommended literature. One indicator, especially for the number of concepts, is the total amount of relevant text passages. Another indicator is the number of concepts in direct connection to *object-oriented programming and design*.

Apart from this, Table 6.6 shows that the first four books appeal to almost the same concepts. Particularly, *class* and *inheritance* are connected in all these books to *object-oriented programming and design*. The first three books in Table 6.6 also have *object* in common. These three concepts are the first three “quarks” defined by Armstrong (2006). These concepts can, therefore, be seen as the most important concepts in object-oriented programming.

Another important indicator of a suitable textbook for an introductory course is the way object orientation is defined or specified; for example, with the concepts *class* or *object*. This implies different didactical methodologies for introducing object-oriented programming, which are shown in Section 5.1.1.

To compare textbooks with the situation in the course concerning the content, it is not only of interest how the main topic is introduced in the textbook but also which concepts related to this topic are the main ones. So, in addition to the concepts that specify object-oriented programming directly, the concepts that have the most specifying concepts and those which specify the most concepts are also examined in this investigation. These concepts ought to be the central concepts in relation to the given topic.

The investigation of the five textbooks leads to a characterization of the books concerning object orientation. According to Deitel and Deitel (2012) and Sedgewick and Wayne (2008), *class* is the concept that is specified the most, whereas Eckel (2006) emphasizes *object*, Flanagan (2005) *method*, and Abelson et al. (1996) *state*. These results reflect the pure statistics on the text passages introduced above. The first three books are oriented on objects, the book by Flanagan (2005) is mainly a reference that has its focus on methods, and the last book is related to the imperative procedural paradigm.

The other interesting group of concepts that are in the main focus of a book in relation to a topic contains those that specify many other concepts. In the example of the five books that were investigated, these concepts and the number of related concepts can be found in the following list. For each book the specific concept and the number of connections to other concepts are on one line.

Eckel (2006): *object* - 9 edges
Deitel and Deitel (2012): *object* - 15 edges
Flanagan (2005): *method* - 6 edges
Abelson et al. (1996): *state* and *variable* - 2 edges
Sedgewick and Wayne (2008): *method* - 4 edges

The last investigation of textbook structure and the corresponding content structure that was conducted on the five example textbooks addresses the related concepts to a specific topic. In contrast to the last investigations where the edges are the focus, the nodes and concepts related to “object orientation” are counted. Table 6.7 presents an overview of these results. In the second column there are the number of nodes and in the third column the number of specified concepts with the number of specified “quarks” in parentheses. The concepts *object-oriented programming and design* are not counted. An analysis related to the “quarks” is the content of the next subsection.

Book	# Nodes	# Concepts
Deitel and Deitel (2012)	41	18 (7)
Eckel (2006)	35	17 (5)
Flanagan (2005)	28	17 (5)
Sedgewick and Wayne (2008)	14	11 (3)
Abelson et al. (1996)	4	6 (1)

Table 6.7: Number of nodes (specifications) and concepts in each book

Again, the ranking of the books is based on the absolute number of nodes and concepts and reflects the results of the investigations that are described above. The equality in the number of concepts of the first three books (Eckel 2006; Deitel and Deitel 2012; Flanagan 2005) is striking. They all consist of 17 or 18 concepts. The intersection of the lists of concepts leads to nine concepts: *object-oriented programming and design, method, object, class, encapsulation, inheritance, argument, parameter*, and *variable*. The last three concepts expand the list of the "quarks". Except for the first, the others cover the concepts defined by Armstrong (2006). It can, therefore, be concluded that these books have in common a kind of a core of object-oriented programming and design. The exact representation of these core concepts is the content of the next subsection.

6.4.2 Representation of the "Quarks" in the Textbooks

In addition to the pure structure analysis that was conducted in the section above, how the textbooks correspond to a topic such as object orientation is investigated. For the purpose of rating textbooks with regard to their usability in a specific context, a comparison of the content of textbooks to a given set of concepts is important. In the example of the five selected and investigated textbooks, this is done on the basis of the core concepts of object orientation. As considered above, the investigation by Armstrong (2006) resulted in a list of definitions of object orientation. These "quarks" are seen here as the core concepts of object orientation, since they are conducted from a broad literature analysis. A concept specification map on the specifications given by Armstrong (2006) is created by constructing a sample map in Section 6.3 with the non-iterative methodology. All of the integrated concepts are connected to object orientation by design. Nevertheless, the explicit node "object-oriented programming and design" is not included in the map. This map and the maps of the textbook analysis are the basis for the following investigation. To be able to compare the maps with each other, some modifications in the naming of the nodes have to be applied. Therefore, the concepts of *message* and *message passing* as well as *data abstraction* and *abstraction* are considered to be the same. The short forms are only abbreviations of the longer node names. By applying the modifications, the node identifiers of the intersection of the nodes of both kinds of maps are the same.

For the concept specification map of the "quarks" (see Figure 6.2) the specifications of the object-oriented concepts presented by Armstrong (2006) are taken. For matching the "quarks" and the concepts from the textbooks, only the concept nodes are matched. The specifications and, therefore, the corresponding nodes are not considered in the process. The original figure of the "quarks" map is shown in dark-gray and builds the basis for the matching maps. If a concept is specified in a textbook and is related to object orientation directly or by other concepts, it is highlighted in blue. The restriction that it is connected with object orientation in one or another way results from the textbook analysis, as only concepts that are in relation to object orientation are included in the analysis. The edges that are contained in the textbook maps, are colored in black. If there is an exact match of a node it is colored green, otherwise it is colored red if there are edges that could not be matched, or colored gray if there is no matching.

A node is matching in an exact way if all the edges can be matched and there are no edges without a match connected to the node. The resulting concept specification maps are shown in Figures 6.3 to 6.7.

The book by Eckel (2006) covers all concepts except *polymorphism*, *encapsulation*, and *data abstraction* (Figure 6.5). The textbook by Flanagan (2005) lacks the concepts of *message passing, polymorphism*, and *abstraction*. The other “quarks” are covered in Figure 6.6. In the book by Deitel and Deitel (2012), only the concept *abstraction* is missing (Figure 6.4). Sedgewick and Wayne (2008) and Abelson et al. (1996) both cover the concept *object* (Figure 6.7 and Figure 6.3). Furthermore, Sedgewick and Wayne (2008) cover the concepts *method* and *class*. Only the two books by Eckel (2006) and Flanagan (2005) need the concept *data* for specification of the “quarks” concepts.

Only the specification of *class* with *object* can exactly be found in the textbooks. Most specifications can partially be matched. Obviously, the books with only a few matched concepts have only little matches in the edges. On the other hand, the books with a lot of matching concepts also provide a lot of matching edges.

If the “quarks” are considered to be the essence of object-oriented programming, the concept specification maps can be used as an overview for matching of the most relevant concepts. The investigated textbooks can be divided into two groups. The textbooks of Deitel and Deitel (2012), Eckel (2006), and Flanagan (2005) have a high degree of matching of the concepts defined as “quarks”. However, in the textbooks by Sedgewick and Wayne (2008) and Abelson et al. (1996), the concept *object-oriented programming and design* is present, but the main related concepts are not specified.

Matching the concept nodes of the maps is done here with an overview map of the main concepts of a specific topic. This is only an example and can, of course, be applied on course materials or other educational literature. For this reason, the concept specification maps are an appropriate tool for finding material that matches the content of a course or textbook, and it can be applied for measure given materials concerning their fit to a specific topic.

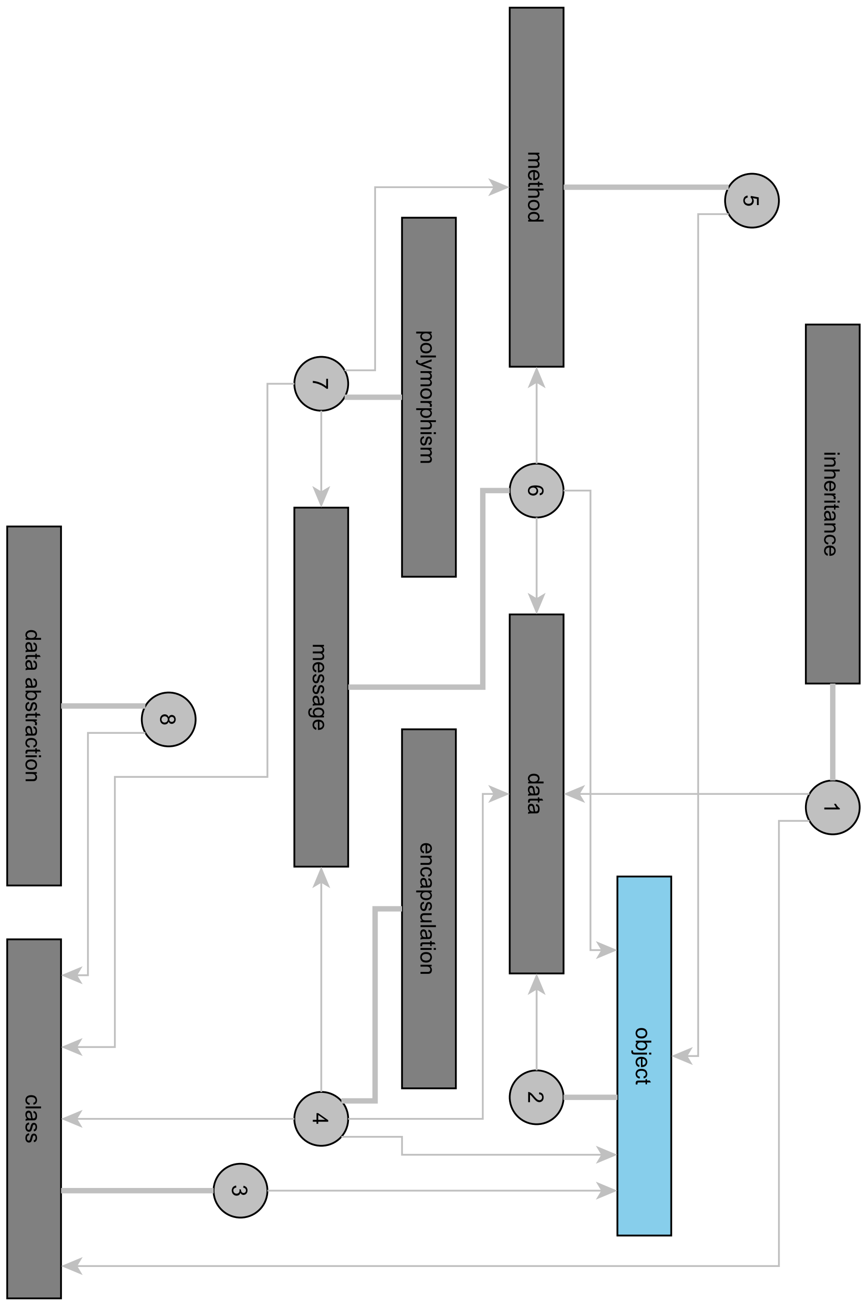


Figure 6.3: Matched concept specification map of the “quarks” of object orientation and (Abelson et al. 1996) - (gray color: no matching - black edges, blue concepts: matching - red node: no exact matching - green node: exact matching)

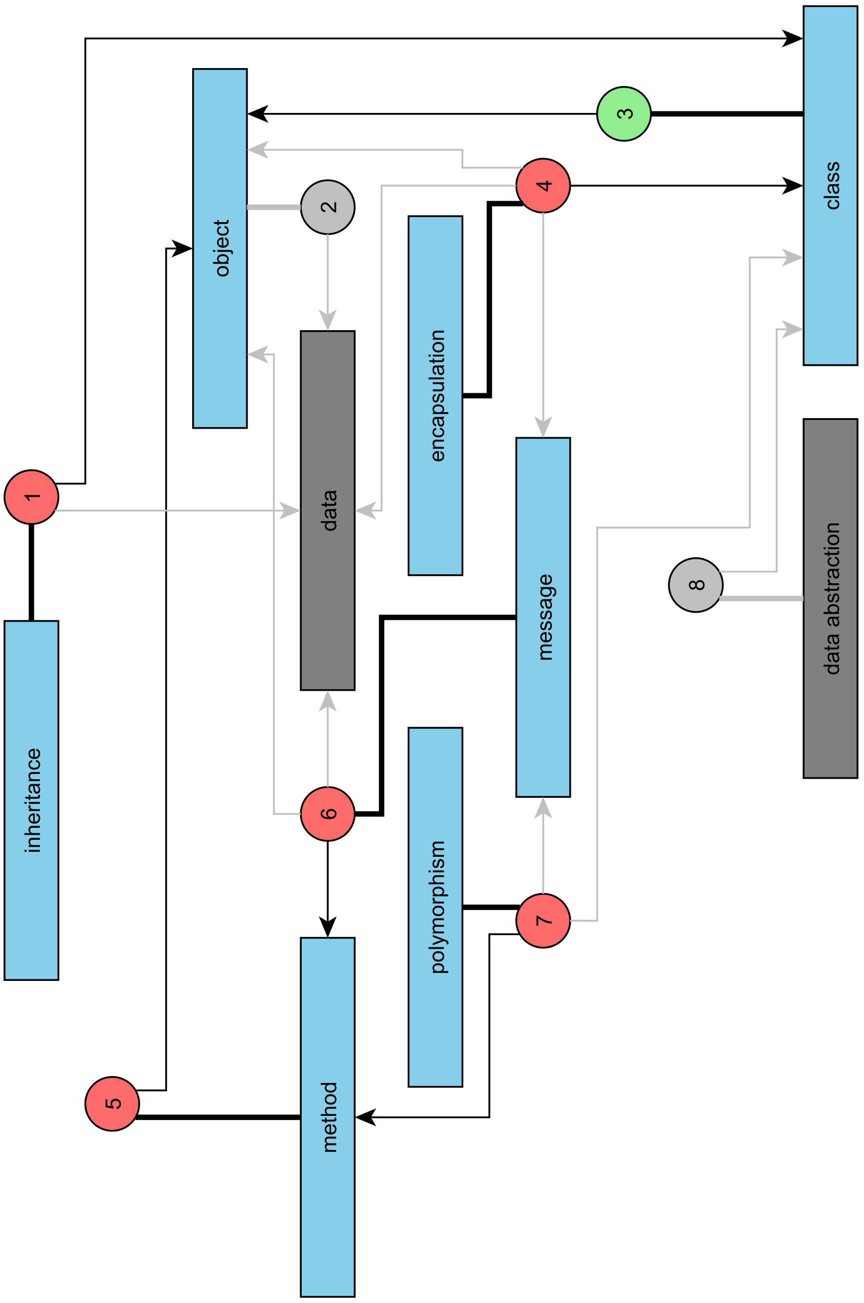


Figure 6.4: Matched concept specification map of the “quarks” of object orientation and (Deitel and Deitel 2012) - (gray color: no matching - black edges, blue concepts: matching - red node: no exact matching - green node: exact matching)

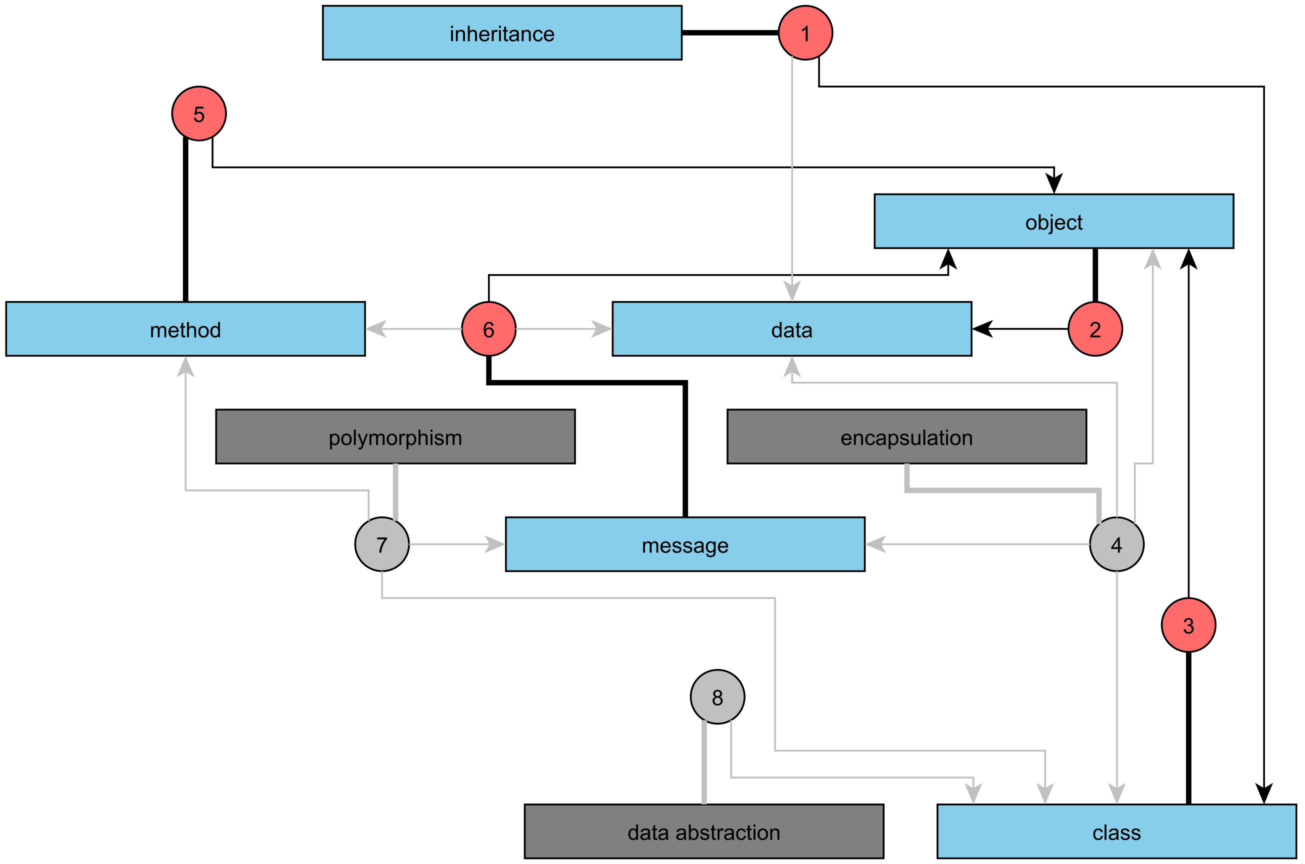


Figure 6.5: Matched concept specification map of the “quarks” of object orientation and (Eckel 2006) - (gray color: no matching - black edges, blue concepts: matching - red node: no exact matching - green node: exact matching)

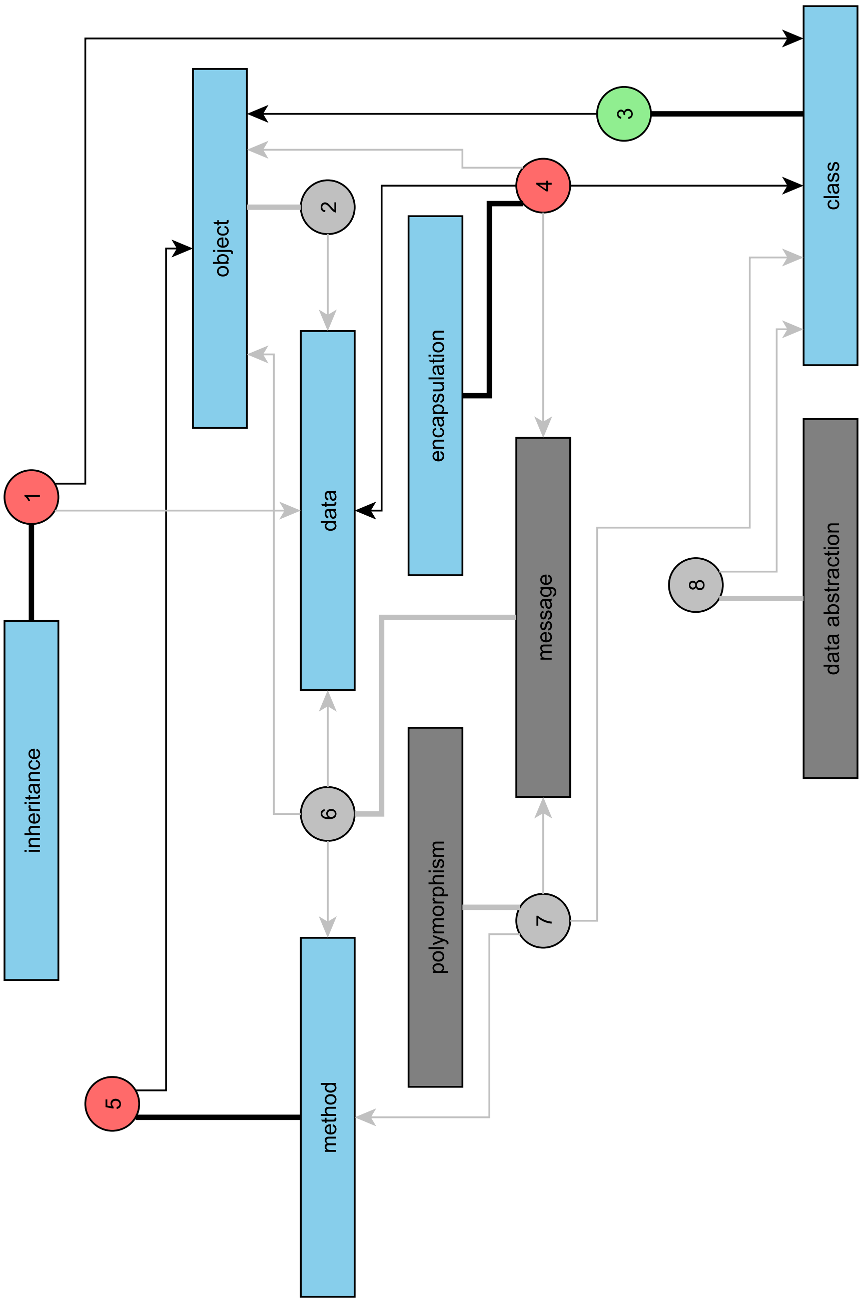


Figure 6.6: Matched concept specification map of the “quarks” of object orientation and (Flanagan 2005) - (gray color: no matching - black edges, blue concepts: matching - red node: no exact matching - green node: exact matching)

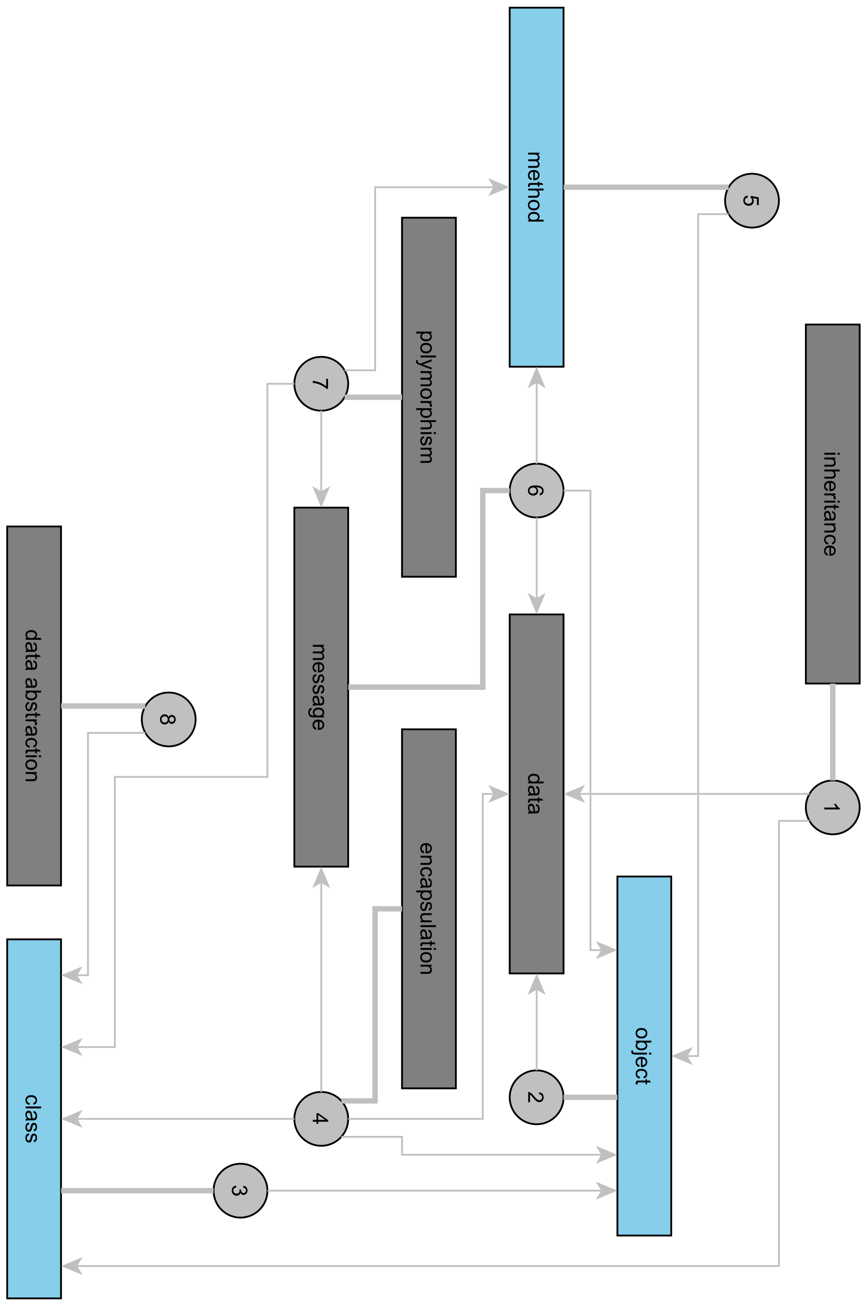


Figure 6.7: Matched concept specification map of the "quarks" of object orientation and (Sedgewick and Wayne 2008) - (gray color: no matching - black edges, blue concepts: matching - red node: no exact matching - green node: exact matching)

6.5 Threats to Validity

During the development process of the concept specification map – and especially during the textbook analysis – several problems occurred. Some of them could be solved immediately, but others remain due to different reasons. Particularly, the problems corresponding to the qualitative analysis methodology and the problems with large concept specification maps are enlightened in a more detailed way.

6.5.1 Missing Intercoder Reliability and Agreement

The rating of the text passages to create a concept specification map depends on the person who is rating the passages. Usually, the coding has to be done by more than one person. While the procedure described in Section 6.3 is very time consuming (for the five books it took nearly two weeks), there were no resources to obtain a second coding for the text passages. If a second rating of the text passages had been conducted, it would have been necessary to calculate an intercoder reliability coefficient. Over 36000 text passages were rated and at least 25% of the rated documents should have been coded twice. Due to restricted resources, it was not possible to find anyone to analyze the nearly 10000 passages.

6.5.2 Large Concept Specification Maps

In addition to the separate analyses of the five textbooks, a common concept specification map of all the books was formed. For this purpose the list of all 1-rated passages are combined and the duplicates removed, using the same methodology as for the optimization during the analyses of the books. For the textbook example, this results in a complex map with 27 concepts and 98 specification nodes. The map, which can also be seen in Appendix A, is only an example for a very complex map. It illustrates that it is possible to create such maps, but there are problems with the layout if there are too many concepts in the map. Although the concept specification maps give a good overview on the concepts in a given text, a complex map is hard to interpret. In most cases it is only possible to illustrate the rough structure. The part displayed in Figure 6.8 shows that there are many edges connected to three concepts. In Appendix A it can be seen that these concepts are *object, method*, and *class*, which emphasizes the character of the selected books. A possible solution for the problem could be the union of the specification nodes on a defined set of rules to reduce the number of edges. Another approach that could enable the union process is the splitting of the specification nodes in binary associations of concepts. After that, all equal edges with their specification nodes are combined. The mapping of the new nodes to the original nodes could be managed by adding numbers for the split ones, like it is done for sub-paragraphs in a text.

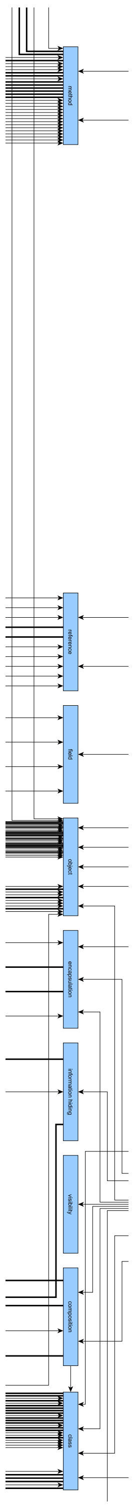
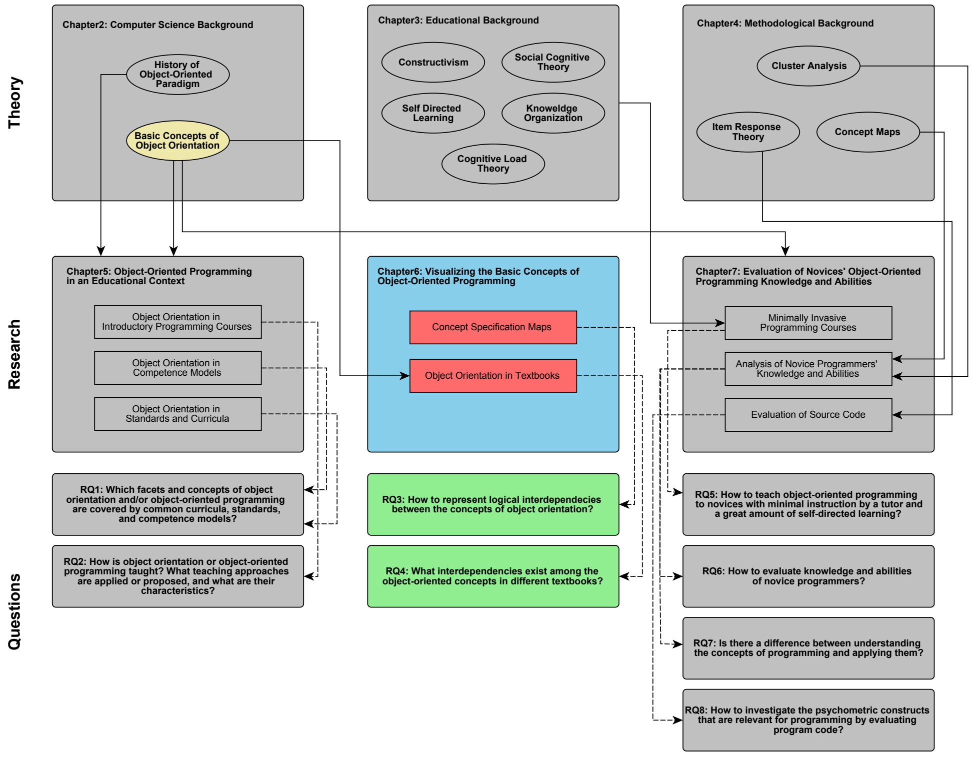


Figure 6.8: Part of a large concept specification map

6.6 Summary

According to research question **RQ3**, the sections above have shown an approach for displaying content structures in texts. Concept specification maps systematically show specifications and definitions of a given topic and their interdependencies. The methodology enables analysis of the content of teaching resources such as textbooks or course material. Besides generally introducing the methodology, two examples are shown to find evidence for interdependencies among object-oriented concepts in different textbooks (**RQ4**). First, a textbook analysis was conducted on textbooks recommended for introductory courses in universities around the world. According to the topic of this thesis, the suitability of the textbooks to object-oriented concepts was investigated. As the results show, there are slight differences among the books. For example, the main focus of the textbooks can be seen in the maps.

Another application is the comparison of concepts provided in a text to a normative basis as the “quarks” of Armstrong (2006). Although the maps enable an overview of the concepts of a given text, there are still problems with the investigations of large text elements. The maps, as they are presented, become very confusing when many concepts are involved. Furthermore, the qualitative analysis process and the surroundings of the calculation of the intercoder reliability are very labor intensive. Nevertheless, in the next chapter the methodology of the concept specification maps is applied on the course material of an introductory programming course.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knowledge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

7 Novices' Object-Oriented Programming Knowledge and Abilities

In recent years there has been a paradigm-shift to object-oriented programming in computer science education which has had several consequences.

> "The paradigm shift means that topics that have for years been considered 'advanced' and relegated to senior courses will now be presented as fundamental ideas to freshmen. Today proponents of objects call for objects first – anything less is counter productive." (Mitchell 2000, p. 99)

When Mitchell wrote his paper the discussion of whether or not there is a paradigm shift had reached its end (see Section 2.1).

In the meanwhile, most courses have adopted the paradigm shift; new educational material is introduced, textbooks on object orientation are widespread, and new programming languages – especially for educational purpose – are being developed. The difficulties for introductory programming courses caused by the shift are described by Manaris (2007). Hubwieser (2007*a*, 2008*b*) has explained the difficulty of learning to program regarding the object-oriented paradigm in relation to the level in the taxonomy of Anderson and Krathwohl (2009). Hubwieser (2008*b*) "pointed out why it is so difficult for students to understand even simple OO-programs: firstly because the learning objectives easily reach the most difficult category (D6) of the taxonomy *(creating meta-cognitive knowledge* e.g. by learning different programming strategies) in this taxonomy. Secondly because there is a big number of objectives that has to be reached before the first program can be really understood" (Hubwieser 2008*b*, p. 143).

Nevertheless, by now almost all introductory courses have implemented the shift. But, how can programming abilities be assessed in this new paradigm and how do novice programmers face their first assignments? At the start of the 2008 winter term we introduced a new programming course just before the first semester to find new options of evaluating novice programmers' knowledge and abilities. Therefore, we reduced instruction for freshmen to a very low level. The focus of the investigations was on the representation of cognitive knowledge through *concept maps* (see Section 4.1) and the programm code produced by the participants. The basic notions underlying the course design are related to the question about how to teach object-oriented programming in educational environments with little instruction from a tutor, and a great amount of self-directed learning (**RQ5**).

In particular, the development of knowledge of the novice programmers and especially their misconceptions are investigated in Sections 7.5.2, 7.5.3, and 7.5.5. These sections, together with Section 7.5.1, present an answer to the research question of how to evaluate knowledge and abilities of novice programmers (**RQ6**).

Additionally, in Section 7.5.4 the differences between the knowledge of programming concepts and their implementation are investigated. This proposes an answer to the question of whether there is a difference between understanding the concepts of programming and applying them (**RQ7**).

Lastly, in Section 7.6 the program code is evaluated with item response theory (see Section 4.3). Here, the focus lies on the question whether it is feasible to find a homogeneous itemset that is generated out of the resulting program code (**RQ8**).

The results of this chapter are published in (Berges and Hubwieser 2010), (Hubwieser and Berges 2011), (Berges et al. 2012) , and (Berges and Hubwieser 2015).

7.1 Related Work

In this section an overview of different related work is given. For this purpose several investigations found in the literature are presented. The related work is grouped by the different investigations executed on the novice programmers. Below, the results of program code analysis (Section 7.5.1), investigations on conceptual knowledge (Sections 7.5.2-7.5.4), and an application of the item response theory on program code (Section 7.6) are shown. These are also the three main categories for the related work. Additionally, the related work on self-directed learning and introductory programming courses in general is presented.

7.1.1 Minimally Invasive Education

In the mid-1980s there were investigations of learning on complex technical devices without any instruction. This so-called instruction-less learning paradigm was described by Shrager and Klahr (1986). The participants of the study should handle a programmable computer-controlled toy. They were given some description of the functionality of the toy, but no verbal description or help. Similar to our study, Shrager and Klahr (1986) wanted to investigate the results that the participants produced when left on their own. They recorded a verbal protocol of the participants' thinking. In the end they found that all participants could handle the toy and had, therefore, learned something about the toy without any instruction.

A study on a variation of self-directed learning was conducted by Rieman (1996). Based on exploratory learning that is related to the instruction-less learning paradigm used in the investigation described above, the learning processes of office program users were investigated. The activities of the participants were documented by a diary study. Additionally, Rieman (1996) performed structured interviews to get more information on the learning strategies of the participants. The results of the study showed that exploratory learning without any material is problematic. Participants need help from materials such as documentations or manuals. Another important result expressed in Rieman's study is that participants only learn the aspects they need for a specific task.

An application of the social cognitive theory and especially the self-directed learning is shown by Dangwal and Mitra (2005). Based on the theory of constructivism (see Section 3.1) Mitra (2000) introduced "minimally invasive education". Here, instruction is only provided when it is required or desirable. Additionally, the duration of instruction is as small as possible.

> "Intervention points can be detected by monitoring learner progress. Such points occur when the learner is observed to have reached a plateau and is doing similar tasks again and again. At this point intervention consists of a demonstration of some new application or capability of the PC followed by discovery learning by the learner. Another type of intervention point occurs when learners are seen to be collectively developing an incorrect concept. At such points, the instructor needs to point out the incorrectness of their understanding through demonstration, and not through direct instruction. This should be followed by a phase of rediscovery, if necessary guided by an instructor." (Mitra 2000, p. 19)

Under the leadership of Sugata Mitra, computer kiosks that are accessible to the public were installed in southern India. Children could use the computer without any instruction. The results of this "Hole-in-the-Wall" project are described in detail by Mitra (2000) and Dangwal and Mitra (2005). In this thesis, this idea was transferred to programming skills in general.

A non-traditional lecture that was quite similar to the courses that the following investigations are settled in, is presented by Isomöttönen et al. (2013). They emphasized learning by programming (doing) and gave, in general, no instruction. More precisely, they introduced implementation sessions and support sessions followed by feedback sessions for reviewing the produced code. Participation in the programming sessions is voluntary, but specific help is provided in the sessions. To evaluate their course design, they posed a questionnaire to the participants. The results cover our findings on the self-assessment of the learning gain.

7.1.2 Introductory Programming Courses

Pedroni and Meyer (2010) proposed organizing the specific subject domain knowledge of object-oriented programming in Trucs (Testable, Re-usable Units of Cognition), which are described in detail in Section 6.1. For the application of the so called Truc-notion graphs in computer science, Pedroni chose an introductory programming course. To construct the graph and the corresponding course she set out the following criteria:

> "Developing a domain model (with its Trucs, notions, and links) and a course model (with its lectures and underlying notions) entails various

> subtasks: (1) define the concepts and skills that the Trucs should represent and identify the associated notions; (2) construct the links between notions resulting in a Truc-notion graph, create concise Truc descriptions by producing the contents of technical sections such as summary, role, applicability, benefits, and pitfalls, and collect common confusions and sample questions; (3) create course models as sequences of lectures each covering a series of notions." (Pedroni 2009, p. 56)

The model shows the interdependencies of the concepts with several types of connections; one if a Truc requires another, one if it refines another, and one if it depends on another. For application of the Trucs of object orientation in an introductory programming course, she built a set of "28 Trucs ranging from procedural programming concepts such as Conditional and Loop to object-oriented concepts such as Inheritance and simple data structures as for example Linked list" (Pedroni 2009, p. 65). Each Truc owns a description, examples, and common confusions. The Trucs and notions combined by relations form a restricted skeleton for the learner. This contradicts what we want to show in our course model, because as Pedroni states in her thesis, it is quite difficult:

> "[...] finding a starting point where no prerequisites are necessary, [...] presents challenges in ensuring that the entire subject area is covered and in sequencing the concepts without prerequisite violations." (Pedroni 2009, p. 151)

This major problem of all courses inspired us to design a course where the participants select their individual starting point. Each participant can profit from their unique set of prerequisites without any influence of instruction.

Bruce et al. (2004) published a study based on semi-structured interviews. They conducted a phenomenological study on how students learn to program, which led to five categories describing the process of learning to program.

Following "the act of learning to program is experienced as following the set structure of the unit in order to 'get through'. When going about learning to program this way, a student's primary intent is to keep up with set assignments. Where marks are to be gained, students will focus on those tasks. Time might be seen as the major factor in determining whether or not the student successfully completes the unit. Students going about learning to program this way are significantly affected by the structure of the course and the way the material is presented" (p. 148).

Coding "the act of learning to program is experienced as learning to code. Students going about learning to program this way see learning the syntax of the programming language as central to learning to program. They are driven by their belief that they need to learn the code in order to program. This may involve rote learning. As in Category 1 [following], time is a major factor because of the amount of syntax that needs to be learned or practiced in order to get through the course. This may lead to frustration. Students going about learning to program this way may desire extra guidance towards specific solutions and examples of code. Time taken to explore concepts and discover their own solutions may be seen as wasted” (p. 149).

Understanding and Integrating “the act of learning to program is seen as learning to write programs through understanding and integrating the concepts involved. When going about learning to program this way students see understanding as integral to learning. They seek understanding of a ‘bigger picture’ over the small tasks they are undertaking as part of coursework. It is not enough to type in the code and ‘see if it works’, rather these students seek to understand what they have done in order to affect the particular outcome” (p. 150).

Problem Solving the act of “learning to program is experienced as learning to do what it takes to solve problems. When going about learning to program this way the student begins with a problem and sets out to discover the means to solve that problem. The understanding that is sought in Category 3 [understanding and integrating] is a fundamental component of this category. As in Category 3, understanding is obtained through adopting a ‘big picture’ perspective, or trying to see the problem and the program as part of a broader context” (p. 152).

Participating or Enculturation the act of “learning to program is experienced as learning what it takes to be a part of the programming community. Understanding what it means to learn to program encompasses the way that programmers think as well as what a programmer actually does. The previous focal elements of syntax, semantics and the logic of the programs are acknowledged in this category, but the essence of what it means to learn to program extends to the actual programming community” (p. 153).

After defining the categories out of the semi-structured interviews, Bruce et al. (2004) gave some implications of these categories for teaching and learning. In particular, they gave an answer to the question of how students can be helped on their way of learning to program. These ideas were considered when we constructed our introductory courses described in this chapter.

In the meantime many institutions offer an introductory programming course with a preceding course. These courses have a lot of different goals and use different contents and methodologies (cf. Dierbach et al. 2005; Edmondson 2009; Faessler et al. 2006; Gill and Holton 2006; Vihavainen et al. 2011).

7.1.3 Novice Programmers

An early study on the difficulties for novice programmers was conducted by Bonar and Soloway (1983) in the early 1980s. They interviewed several participants of their introductory course. The course was based on the procedural paradigm with the programming language Pascal. Nevertheless, the results are still valid.

> “We find it quite interesting that novices seem to understand the role or strategy of statements more clearly than the standard semantics.” (Bonar and Soloway 1983, p. 12)

Similar results to those from this thesis are presented in an early study by Perkins et al. (1988). They identified different types of novice programmers by applying the method of clinical studies. For that purpose, they documented results in a measurement of the time for programming relative to the time of thinking. The both extremes are called “stoppers” and “movers”. The “stoppers” became stuck in thinking about the problem they faced, while the “movers” implemented without any thinking. Another way of differentiating the novice programmers is by their attitude towards bugs:

> “Some novices seem to take the inevitable occurrence of bugs in stride, while others become frustrated every time they encounter a problem. The former seem to recognize that mistakes are part of the process of programming, part of the challenge. They study their bugs and try to use the information they gain. The latter students appear to view bugs more as reflecting on the value of their performance. For them, programming mistakes are so obvious - they show up on the screen as incorrect output or in a program that will not run or gets stuck in the middle.” (Perkins et al. 1988, p. 267)

Additionally, Perkins et al. (1988) propose solutions for helping students get into programming more easily. First, “close tracking” means that the students should read the implemented code from the perspective of the computer and analyze exactly what each single line represents. Nevertheless, accurate “close tracking is a mentally demanding activity. It requires an understanding of the primitives of the language and the rules for flow of control. In addition, as the student proceeds through the code, the student must map its effects onto changes in what might be called a ‘status representation’, specific to the problem” (Perkins et al. 1988, p. 269). The second method gathered from the documentation is called “tinkering”. The students write some code and then apply small changes to the code in the hope of making it run-able. Last, “breaking problems down” is not only a challenge concerning programming, but a general method in the field of problem solving. As Perkins et al. (1988) found in their documentation, most students do not even recognize that they have to break a problem down. And if they recognize it, they are not able to find suitable chunks (cf. Perkins et al. 1988, p. 274).

The self-assessment of the previous knowledge is important for our research on novice programmers. Bergin and Reilly (2005*a*) try to figure out factors for success in a programming course. They asked their students for assessments of their programming experience and their non-programming computer experience. They investigated the students and separated the students into groups with and without previous experience. However, no “significant differences were found between students with or without previous programming experience or between students with or without non-programming computer experience and performance module” (Bergin and Reilly 2005*a*, p. 413). In contrast, the investigations of this thesis for understanding programming-related concepts show an alignment in the knowledge of participants with and without any previous knowledge in programming.

An early study by Bonar and Soloway (1985) shows that novice programmers prepare plans to implement their programs. The freshmen define bug generators as the process to bridge the gap in their programming knowledge. “Thus, in writing a program, a novice will encounter such a gap and be at what we have called an impasse. In order to bridge these gaps, the novice uses patches. By their very nature, these patches are likely to be incorrect" (Bonar and Soloway 1985, p. 140).

Besides the investigation of Bonar and Soloway (1985), Pea (1986) classified language-independent conceptual bugs of novice programmers. "These misunderstandings [...] have less to do with the design of programming languages than with the problems people have in learning to give instructions to a computer" (p. 26). The bugs found in program code gathered in many years of Logo programming courses are classified into three categories: parallelism bugs, intentionality bugs, and egocentrism bugs.

Parallelism bugs are underlain by "the assumption that different lines in a program can be active or somehow known by the computer at the same time, or in parallel" (p. 27).

Intentionality bugs describe bugs "in which the student attributes goal directedness or foresightedness to the program and, in so doing, 'goes beyond the information given' in the lines of programming code being executed when the program is run" (p. 29).

Last, "**[e]gocentrism bugs** are those where students assume that there is more of their meaning for what they want to accomplish in the program than is actually present in the code they have written" (p. 30).

All these categories have one "superbug" in common. The students assume a kind of hidden mind in the programming language that can interpret the ideas the programmer had when writing the code.

Spohrer and Soloway (1986) tried to figure out two common perceptions about misconceptions. They found that there are only a few bug types that lead to the majority of students' misconceptions and most misconceptions have no relationship to the language, but are of general purpose.

Putnam et al. (1988) also investigated students' misconceptions on the basis of interviews that were related to a programming test. They list misconceptions for several elementary programming concepts like assignments, input or output statements or loop, and conditional constructions.

Another early investigation of the difficulties related to learning to program is published by du Boulay (1988). After introducing general areas of difficulties, du Boulay introduces several examples of misapplied analogies for array or other data structures (cf. du Boulay 1988).

A general study on problems novice programmers have to face is published by Robins et al. (2001). After defining and explaining of programming levels such as novices and experts, they conducted a study on an introductory course. Teaching assistants were asked to fill in a checklist on common difficulties the students struggled with. In our investigation we also asked the tutors to document the questions the participants asked.

A broader investigation on misconceptions, especially in programming, was conducted by Clancy (2004). He describes extensively several different types of misconceptions that can occur during programming introduction. After a general literature review on the term of misconception, different causes for misconceptions are given; for example, over- and under-generalization of knowledge transferred from language, mathematics, or former programming experiences. Another cause of misconceptions by novices might be a confused computational model.

> "A high-level language provides procedure and data abstractions that make it a better problem solving tool, but which hide features of the underlying computer from the user. These abstractions, especially if they have no real-life counterparts, can prove quite mysterious to the novice." (Clancy 2004, p. 90)

Examples for these abstractions are the input statements, constructors, and destructors, or the construct of recursion. For this thesis, the method of drawing concept maps is applied for finding misconceptions focusing on the knowledge misconceptions and, because of that, mainly on the over- and under-generalization of knowledge.

Another part of the study that Clancy (2004) published investigates the influence of a learner's attitude toward learning to program. He provides several comments from students, which show common beliefs or attitudes.

> "Some students expressed the belief that arrays of records made the programs containing them difficult for people to read . ... [Quote from student] S11 : 'It looks like too much is happening, you know, when you first read the program. Oh, no! Arrays! Records! Everything else!' ... Experts, on the other hand, advocated the use of such data structures as arrays of records, and explained that they were expected as an alternative to freezing the program to work with a fixed number of individual variables." (Clancy 2004, p. 94)

There are several investigations about the problems and difficulties that novice programmers have. Hanks et al. (2004) investigates the advantages and disadvantages of pair programming concerning the problems that occur during an introductory course. The general publication by Robins et al. (2006) on problems encountered during introductory computer-science courses built the basis for the investigations conducted by Hanks (2007). Lahtinen et al. (2005) provide a list of programming concepts as well and asked students and teachers in a survey to express their self-assessment on the difficulties they had when learning the programming concepts. They found that the "most difficult programming concepts were *recursion* (C4), *pointers and references* (C6), *abstract data types* (C9), *error handling* (C11) and *using the language libraries* (C12)". (Lahtinen et al. 2005, p. 16)

Another interesting result that Lahtinen et al. (2005) found was that the students' difficulty level with the concepts is lower than the level of the teachers, which has strong implications on teaching. A similar study was conducted by Milne and Rowe (2002) a few years earlier. They assume the difficulties to result from "the lack of understanding by the students of what happens in memory as their programs execute. Therefore, the students will struggle in their understanding until they acquire a clear mental model of how their program is 'working'—that is, how it is stored in memory, and how the objects in memory relate to one another" (Milne and Rowe 2002, p. 63).

Another list of misconceptions is presented by Holland et al. (1997). In contrast to the list of Robins et al. (2006), the list of Holland et al. (1997) only contains misconceptions related to objects. Additionally, Holland et al. provide examples to avoid or encounter the misconceptions.

Sajaniemi et al. (2008) present another way to obtain representations of the understanding or misconceptions that students have. They let the participants of an introductory course draw a picture of the program state at a specific moment. The drawings were analyzed in a qualitative manner by coding the concepts of interest. Besides the information that concepts could be put in correct associations, the information of incorrectly associated concepts is of interest. Sajaniemi et al. (2008) found two groups of "errors"; those that are misconceptions related to object orientation and those that are misconceptions of the studied Java program (cf. Sajaniemi et al. 2008, p. 23). The results that the visualization of the knowledge grows during a course match our results found in the representation of knowledge with concept maps.

7.1.4 Conceptual Knowledge

Above all of the comparisons of theoretical methodologies, introduced in Section 4.1, Keppens and Hay (2008) introduced three applications of some of the methods with the purpose of introducing them into programming. "Firstly, the closeness index and linkage analysis were suggested as suitable assessment methods for determining a student's understanding of a programming language's basic concepts. Secondly, chain-spoke-net differentiation was put forward as an effective method to evaluate a student's awareness of software libraries. Thirdly and finally, a qualitative simulation-based approach was proposed to assess a student's model building ability" (Keppens and Hay 2008, p. 41).

The study on concept maps most related to the one presented here was conducted by Sanders et al. (2008). They investigated several introductory courses into object-oriented programming in different countries. Based on the concepts that Armstrong (2006) proposed as the "quarks" of object orientation (for a detailed description see Section 2.2.1), as well as some misconceptions they found in literature, they asked their students to draw concept maps on the basis of the two given concepts "class" and "instance". The resulting maps were analyzed by normalizing the edge labels and concatenating the concepts in the nodes and the labels on the edges to sentences. These sentences were investigated with the method of qualitative text analysis. In the end they found that most students have problems relating "class," "instance," and "object" in the proper way. The other common problem that they wanted to investigate could not be found in the maps. The relations of "data" and "behavior" to "class" were done in a proper way by most of the students. In addition to these and several other results on different concepts of object orientation, Sanders et al. (2008) found that "making concept maps seems to be a useful exercise" (Sanders et al. 2008, p. 336).

In my investigation of the students' concept maps (see Section 7.5.2), similarities are found to the results of Sanders et al. (2008).

> "Our analysis confirmed earlier research indicating that students do not have a firm grasp on the distinction between 'class,' 'object,' and 'instance'. Earlier results suggested that some students think of classes as just being data storage (like arrays or structs); we found that while many students do connect classes with data, even more make the connection to behavior." (Sanders et al. 2008, p. 336)

Besides the detection of cognitive knowledge by drawing concept maps, Renumol et al. (2010) investigated cognitive processes by applying a qualitative text analysis of transcribed verbal protocols of students programming. Renumol et al. (2010) divided the participants of the study into two groups. The one group they called the "effective" participants since they could realize the programming questions with only a small effort. The other group they called the "ineffective" group since they needed a great effort for solving the problem. Both groups were analyzed by a verbal protocol that was coded initially with 34 cognitive processes. During the analysis, eight further cognitive processes were found: confusion, hypothesis, interrogation, iteration, monitoring, recollection, recurrence, and translation. According to the findings in this thesis, confusion represents an interesting new cognitive process.

> "It is a CP[cognitive process] which creates a disordered mental state. It prompts the mind to switch randomly and/or quickly between various concepts or decisions, due to lack of clarity in thinking. Confusion can occur when a subject lacks order in thinking as some new data or knowledge does not assimilate with the existing data or knowledge. It can also be due to lack of relevant knowledge or failure in applying the acquired knowledge when needed. Confusion blocks or diverges the path to an objective." (Renumol et al. 2010, p. 14)

Furthermore, the study by Renumol et al. (2010) supports our findings on the differences in the cognitive level when programming.

> "Based on the analysis, it has been observed that programming is an interplay of lower and higher CPs[cognitive processes] and needs various cognitive skills. The results show the importance of human factors in the programming process. Amalgamation of so many CPs increases the processing load to the brain and makes the programming process difficult and complex to learn and practice." (Renumol et al. 2010, p. 17)

Another study was conducted by Hubwieser and Mühling (2011*a*) on non-major computer science students. Four concept maps were gathered during an introductory course. These maps were evaluated and compared to expert maps and were investigated by structure. Hubwieser and Mühling (2011*a*) described two interesting results. First, the "ruggedness of the maps has been increasing [...] over the course. This might indicate that students are learning in a way that favors creating clusters of new knowledge instead of integrating it into an existing model, confirming the *knowledge-as-elements* perspective of the Conceptual Change theory" (Hubwieser and Mühling 2011*a*, p. 377).

Second, they investigated the number of correct edges and associations between concepts. In the course they investigated, there was an increase in the knowledge gained. The development of knowledge is also part of the investigation described in the sections below.

In a second investigation on a non-major course, Hubwieser and Mühling (2011*b*) searched for patterns in concept maps that were used quite frequently by the students. Those "knowpats" were extracted from the concept maps the students drew during the semester. After a reduction of the maps to undirected graphs, they were found by using the AcGM algorithm, which detects subgraphs. An adaption of the algorithm to concept maps leads to a method for finding frequently used subgraphs in the maps of students. In the presented investigation of conceptual knowledge of students the same scoring method of Hubwieser and Mühling (2011*b*) was applied.

In another study Hubwieser and Mühling (2011*c*) investigated conceptual knowledge in a longitudinal way. They let the students of a non-major introductory course draw concept maps during the semester. After rating the maps of all four tests and normalizing the labels of the edges, they found that the participants were only using a very small set of labels. Mainly, only *contains* and *has* were used in the maps.

7.1.5 Program Code Evaluation

Hansen (2009) analyzed a student survey about the engagement and frustrations in programming projects that accompanied their computer-science introductory courses. The participants were asked to fill in a survey on how engaging or frustrating a course assignment was. Furthermore, Hansen (2009) introduced a metric for assignments by defining "niftiness" as the subtraction of frustration from engagement. As the score ranges from 0 to 10 for both values, scores from -10 to 10 for niftiness are possible. After evaluating a couple of projects they introduce the niftiest and the least nifty projects with given reasons for their score (cf. Hansen 2009).

Literature provides different scoring methods for object-oriented source code. Börstler et al. (2008) proposed three categories according to several criteria for evaluating object-oriented example programs. Sanders and Thomas (2007) introduced a checklist for scoring object-oriented programs by investigating concepts and misconceptions in object-oriented programming. Truong et al. (2004) built a framework for static code analysis of students' programs. They summarized common poor programming practices and common logic errors from literature and a survey conducted on teaching staff and students. The framework works on a XML basis and enables the students to receive feedback on their programs and rate the code automatically.

Currently, there are several code-testing tools that have been developed for educational purposes. Most of them work online based on several platforms. In contrast to the tools introduced by Drasutis et al. (2010) and Vihavainen, Vikberg, Luukkainen and Pärtel (2013), for example, a method of code evaluation based on the item response theory is introduced. This method is currently manual, but that can in the future be automatized. The automatic testing or scoring of code produced in response to an assignment is difficult. Kemkes et al. (2006) investigated the scoring of the International Olympiad in Informatics. They score the code with 1 if it runs successfully on a given set of input data; otherwise, it is scored with 0. They argue that "current practice yields tests with too little diversity of difficulty – too difficult for most and too easy for a few – and hence with poor discrimination among the majority of contestants" (Kemkes et al. 2006, p. 230). Alternatively, in Section 7.6 a method is presented to evaluate code on the basis of the item response theory (see Section 4.3) simply by the presence of syntax elements in the code.

Syntax elements were also the field of investigation for Luxton-Reilly et al. (2013). Those authors investigated differences in the correct solutions of the students. They defined a taxonomy for that purpose that distinguishes the code on the distinct themes of structure, syntax, and presentation. Structure means different control flow in the code, while syntax means differences in the code with the same control flow structure. Finally, presentation means variation in the identifier names or number of whitespaces, for example. In contrast to our investigation based on the item response theory, Luxton-Reilly et al. (2013) evaluated the code on a structural level.

7.2 Minimally Invasive Programming Courses

In the past, most computer science learning was done with little self-instruction or self-directed learning. The contact to computers was limited to organizations that could afford a computer; later on there was one computer for a whole family and it was mostly used for information and communication technology(ICT) purposes. Even in 2003, Robins et al. (2003) stated:

> "Most novices learn to program via formal instruction such as a computer science introductory course ('CS1'). This sets the topic of novice learning and teaching in the context of an extensive educational literature. Current theory suggests a focus not on the instructor teaching, but on the student learning, and effective communication between teacher and student." (Robins et al. 2003, p. 156)

This is supported by Feldgen and Clua (2003) who took change of motivation into account by introducing web and game programming into their introductory programming courses. But, how does learning to program work in times of the internet and a broad distribution of technical devices in society?

In an attempt to answer this question we investigate students who attempt to learn programming on their own. To simulate a self-instructional environment, we constructed introductory programming courses with little direct instruction. The participants are left on their own with nothing but the course materials and a peer tutor. The idea of this teaching concept is based on Sugata Mitra's "Hole-in-the-Wall" project and his "minimal invasive" instruction that was described in Section 7.1.1. The influence of any kind of teacher (including peer tutors) on the learning process should be as little as possible. A prerequisite for handling these "minimally invasive programming courses" is small group sizes so that there is a good tutor to group ratio with regard to the mentoring.

Although students are put in groups, each student works individually on his/her own assignments according to the ideas of Turkle and Papert (1990). This was done because we wanted to investigate the individual learning outcomes and the limitations of self-directed learning in an introductory object-oriented programming course. However, the students were actively encouraged to talk with each other. These interactions and those with the tutor who originates from the peer-group of students should reinforce the advantages of learning to program in peer groups, as mentioned by Vihavainen, Vikberg, Luukkainen and Kurhila (2013), Cottam et al. (2011) or Hanks et al. (2009). Nevertheless, the role of the tutor is quite unique and, is demonstrated in the results of the experiments below (see Section 7.5). With the assistance of a tutor in the presented study, an attempt is made to counter the problems – concerning the absence of an effective computer model – that were mentioned by Ben-Ari (1998) in his paper on constructivism in computer-science education.

> "The computer science student is faced with immediate and brutal feedback on conclusions drawn from his or her internal model. More graphically, alternative frameworks cause bugs. Computer science is unlike school physics: intuition and manipulative facility are not sufficient for passing a course, and the consequences of misconceptions are immediately exposed." (Ben-Ari 1998, p. 259)

The minimal amount of instruction and the advantages of peer tutoring should strengthen the comfort level of students. A study on the influence of comfort level and motivation on success in programming shows a significant positive correlation (cf. Bergin and Reilly 2005*b*). Although the study was conducted on a very small group of students, the results emphasize the importance of countering the fears of novice programmers. Additionally, the motivation aspect can be negotiated, as participants join voluntarily. The idea of the minimally invasive programming courses faces these facts and tries to give students the option of solving the problems on their own.

This leads to a learning situation that is characterized by Boytchev (2011) as "wild learning". Boytchev (2011) explained the basic idea underlying the courses as follows:

> "When we give a toy to a child, we just show quickly how it is used. Then the child continues to play with the toy and to explore its functions." (Boytchev 2011, p. 1)

The results of the paper are far from being generally transferable, but the idea of letting a novice programmer simply explore the new "toy" of programming is worth mentioning and the success could be reproduced during the subsequent courses, as can be seen in Section 7.5.1.

The didactic methodology underlying the minimally invasive programming courses contradicts most studies that focus on learning to program. Börstler and Sperber (2010) described two ways of didactic methodologies. Both are based on instruction. In the next section a setting for a small sample course implementing the minimally invasive idea is shown and the results, which are presented in the relevant sections, show that this idea also works well.

7.3 A Preliminary Course for the Introduction into Computer Science

There have been several suggested solutions for the problems occurring with teaching how to program (see related work in Sections 7.1.2 and 7.1.3), but up to now there has been no general solution solving the problem in all its facets. Contrarily, the omni-presence of computer media in our society has the effect that the population of freshmen at universities studying computer science is more heterogeneous than ever. The reasons for choosing to study informatics at university are as varied as there are programming languages the prospective students may know. In his pilot study on the observations “that many students who enroll for a first computer science course do so with some very limiting misconceptions of what the discipline entails”(Greening 1998, p. 145), Greening (1998) found various reasons for why students study computer science. Although two thirds of the participants of the study enrolled for computer science because of their personal interest in gaining important skills, there were those who enrolled only for logistical or career advantage reasons.

Additionally, in his study Greening (1998) asked for the expected first-year programming language. He expected a majority of students to answer with “Pascal” or “BASIC,” but two thirds of the students were unable to answer even this very basic question. About 20% gave a reasonable answer and about 13% answered with a non-programming language or other misconceptions.

In autumn 2008, we began offering a new course before the first semester began; the aim was to make the different groups of freshmen more homogeneous. In this course we provide a short introduction to object-oriented programming and decrease the fear of programming itself. The following description of the course was first presented in (Berges and Hubwieser 2010).

Before introducing the course with its prerequisites and the course design, a definition of a novice programmer by Winslow (1996, p. 18) is presented. A novice programmer

- lacks an adequate mental model of the area,
- is limited to a surface knowledge of the subject or has fragile knowledge (something the student knows but fails to use when necessary),
- uses general problem solving strategies (copy and paste),
- tends to approach programming through control structures,
- uses a line-by-line, bottom-up approach to problem solving.

7.3.1 Prerequisites for the Courses

The necessity for the installation of the course arises from the German lecture system at universities. During the semester there are mainly lectures with very little time for practical work. Nevertheless, it is possible for students to study computer science without any prior programming knowledge. This implies that students without such prior knowledge should also somehow be accommodated. For this purpose we developed and installed specific programming courses that take place before the start of the first semester (cf. Hubwieser and Berges 2011). Further on the courses are called “preprojects” due to their assignment character.

To explain how we set up the course, the prerequisites of prospective students at the TU München first have to be looked at. There were about 300 to 400 freshmen in the relevant study paths in the years from 2008 to 2010. In 2011, there were about 600 new students because the first run of the eight-year lasting Gymnasium in Bavaria ended. Together with the end of the nine-year lasting Gymnasium in Bavaria, this caused two age groups to finish school at the same time.

In the first three runs in the years 2008 to 2010 the majority of participants had only a few previous contacts to concepts in computer science; only a minority had previous knowledge of object-oriented concepts. However, since 2011 with the introduction of object-oriented concepts in Bavarian school (cf. Hubwieser 2006, 2012, Section 5.3.6.2) most participants now have contact with object-oriented concepts, while only the minority has no experience at all.

All of the programming beginners were invited to voluntarily join our course. They were sent their invitation together with their enrollment material. After that they had to enroll to the course, giving a self-assessment of their previous knowledge.

Participation rates in the preprojects from 2008 through 2011 are shown in Table 7.1. Actually, 40 to 70% of each age group participated in our courses.

Year	Enrollments	Prospective students
2008	200 (67%)	298
2009	136 (49%)	279
2010	170 (42%)	404
2011	228 (40%)	570

Table 7.1: Number of preproject’s absolute and relative enrollments between 2008 and 2011

To get the courses included into the already existing preliminary program that takes place during the weeks just before the start of the winter-term, the design had to be discussed within the department. Because of this, we had to adapt our course to the main introductory course. This had an impact on the design of the course including the programming language and the concepts described in the course materials. The demand that no concepts of the introductory course should be instructed in detail by the preprojects had a strong influence. However, it supported our goal to give as little instruction as possible. The consequences of these demands on the course design are discussed in the next subsection.

7.3.2 Design of the Course

Group homogeneity is important for enhancing individual learning as stated by Pinto (2012). Although the students were assigned to groups, they were asked to solve their assignments on their own. This was done to allow for the individual learning gain to be investigated. As described in the previous section, students of different study paths were invited to participate in the course. The students ranged from majors to non-majors in informatics. Within these different study paths further differences were noted in the students' previous knowledge, which resulted from the different school backgrounds (Schulte and Magenheim 2005). Therefore, the separation on the basis of self-assessed previous knowledge seems to be the best way to get the required homogeneity.

The purpose of self-directed learning in a group requires similar prerequisites (see Section 3.2). The idea of learning in peer groups enforces the need for homogeneity. Within the registration process for the course all students were asked to self-assess their prior programming experience according to one of three levels:

(1) I have no experience at all.

(2) I have already written programs.

(3) I have already written object-oriented programs.

Based on this information, the groups were composed to be as homogeneous as possible. The demands of the programs the students had to master differed according to their level of programming experience.

As mentioned in the subsection above, the department had many limitations that had to be taken into account when organizing the courses. The limitation with the most influence was the time slot available for the course. Only two weeks were allocated for the course. In addition, only five adequate rooms with at most 20 working places were available. Another restriction was that no student who wanted to participate in the course could be refused.

Based on these restrictions, a course was implemented that took two and a half days. Each half day was three hours. The projects the students had to manage were adjusted to this time slot.

For organizational reasons (working spaces in the rooms) and for adequate mentoring, the students were divided into small groups of 10-12 individuals. Each group was coached by a tutor who was usually an experienced student from the fifth or higher semester. As peer tutoring affects the motivation of students in a positive way (Wigfield et al. 2012; Carter et al. 2011), the comfort level is generally increased by building small groups and assigning a peer tutor. Although this is the biggest predictor (cf. Wilson and Shrock 2001), the tutors were advised to help the students with practical tips or explain the worksheets or the IDE, but were strictly told to not give any instruction beyond this to avoid a bigger influence of the tutors on the results. Besides the tutor, students had access to instructional sites on the internet. The documentation of Java 6[16] and the online version of the book “Java ist auch eine Insel”[17] were recommended.

7.3.2.1 Gathering the Appropriate Topics for the Course

Another restriction placed by the department on the course was that no concepts of the introductory course into object orientation and object-oriented programming could be taught in detail. As it was our intention to offer no direct instruction, this did not pose a problem. Nevertheless, a selection of concepts had to be made. The main concepts of object orientation are discussed in Section 2.2. However, not all concepts can be taught on a freshmen level. Due to the very short time period available for each group, we decided to reduce the course to the very basic concepts. The grammar school textbooks in Bavaria were used as a basis for this reduction (Frey et al. 2004; Hubwieser 2007*c*, 2008*a*, 2009, 2010). According to the comparison of the “objects-first” and the “objects-later” strategies of Ehlert and Schulte (2009*a*) (see Section 5.1.1), and the corresponding list of object-oriented concepts that form a typical teaching sequence leading to the ability of object-oriented programming (classes and objects – attributes (incl. data types) – methods (incl. control statements) – inheritance – association), we designed our worksheets implementing this sequence. The items *inheritance* and *association* were omitted, because we did not expect the students to understand them without instruction. Ehlert and Schulte (2009*a*) argued that arrays are one of the most difficult concepts of programming in general. Nevertheless, arrays are mentioned on the worksheets without explaining the underlying details such as references. The participants were thought to get along with the object-oriented concepts such as *object, class, method, attributes,* and *data encapsulation*, as well as the concepts that are necessary for implementing control structures such as *conditional statements* or *loops*. Additionally, the input and output from and to the console and the main-method are included as relevant topics.

Due to the department’s restrictions we had to use Java as the programming language, although there are several known didactic difficulties. In particular, there are several concepts in Java that cause a high intrinsic cognitive load (see Section 3.4). For example, the fact that simple types are not regarded as classes or the concepts related to the main-method (cf. Kölling 1999*a*). In addition to the programming language, the programming environment was also given by the department. The participants should have seen Eclipse at least once. According to didactic guidelines (see Section 5.1.2; Kölling and Rosenberg 1996; Kölling et al. 2003), we suggested using the BlueJ-IDE[18] to develop and test the first objects and classes. For the participants who assessed themselves to be without any previous programming knowledge, this choice provides the capability to get into object orientation more easily. After the first steps, the students had the choice to continue their project work using the Eclipse-IDE[19]. It was recommended that the second group also use BlueJ first and then Eclipse. Since

[16] http://docs.oracle.com/javase/6/docs/api/index.html - last access 10.12.2014
[17] http://openbook.galileocomputing.de/javainsel/ - last access 10.12.2014
[18] http://bluej.org - last access 10.12.2014
[19] http://www.eclipse.org - last access 10.12.2014

the second group state previous knowledge in programming in general, the switch from one platform to the other was conducted much earlier. For the third group we omitted BlueJ and the participants started with Eclipse from scratch. Additionally, they were introduced to NetBeans, especially for building graphical user-interfaces.

Due to the fact that students should actively construct their knowledge and abilities, the programming course design represents an application of the constructivistic approach (Hadjerrouit 1999; Section 3.1). The assignment that each student received contained a description of a small programming project for them to work on from scratch. As the previous knowledge in the groups differ, each project focused on different concepts related to different levels of previous knowledge. Every project was open-ended to prevent the more experienced students of each group from becoming bored.

The first level students were asked to program a "Mastermind" game. The main idea of this project was to create a class, use objects, declare attributes, and use the concepts of assignment and method call. At the end of the project, arrays and the declaration of methods constituted the final goals.

The second level group had to implement a tool for managing results from a sports tournament (e.g., a football league). Besides very simple algorithmic thinking, the focus of this group lay in the application of chance. To organize the upcoming data, fields and other data structures are important.

The group of the third level had to program a version of the dice game "Yahtzee". The main focus here lay in the use of class hierarchies, advanced data structures, and algorithmic thinking.

7.3.2.2 Design of the Course Material

All assignments and the main concepts that are useful during the programming process are put together on worksheets according to the idea of minimally invasive programming courses. In total, the participants of the preprojects got four worksheets that are related to each other. The topics of the worksheets are based on the concepts introduced in the didactic methodology of objects-first (see Section 5.1.1) and on the concepts underlying the compulsory subject in Bavarian Gymnasien (cf. Hubwieser 2012). There are different investigations of learning objectives that are related to the textbooks for this type of school (cf. Steinert 2010; Hubwieser 2008*b*). Hubwieser (2008*b*) listed the concepts contained in the textbook related to object-oriented programming and the concepts involved in the learning process. Due to the small time slot for the courses and the limitation of the worksheets in the current investigation only a small part of the concepts are included. Notably, the state of an object is omitted. Furthermore, the understanding of references and generalization are eliminated from the worksheets in order to handle the content within the short time slot. Most of the concepts mentioned in the textbook are reduced to a very basic level. The complete worksheets can be seen in Appendix B.1. If the students had been presented with a textbook, as done in the regular introductory course of the first term, the students would have hardly been able to figure out what information is relevant for their projects. Using the worksheets, we were able to define exactly what information the students would receive.

The first sheet described the task itself and the programs used. Each task is presented by a small project description (see Table 7.2). Additionally, the students receive an initial short overview of the course. In addition to the programming environments mentioned above, ObjectDraw[20] is introduced as a tool for graphically exploring objects, attributes, and methods. Furthermore, StarUML[21] as a tool for object-oriented modeling is described.

Level	Description
Easy (1)	**Mastermind** A player fixes at the beginning a 4-figure numeric code that is composed from ten digits. Another player tries to find out the code. Therefore, he puts a numeric code of the same kind as a question. On each turn the guesser gets the information how many elements he has guessed right and how many are in the right position, or how many digits were right, but not in the right position. The goal of the game is to guess the code as fast as possible, but in twelve steps at most.
Mid (2)	**Sports Administration Tool** In this project the results of a sports game (e.g., soccer) should be recorded and evaluated. The program should provide the option to record a game result and print out a table. A game is always played by two teams and ends with a result. Whether a game can end undecided, depends on the type of sport. The teams contain a fixed number of players.
Difficult (3)	**Kniffel/Yahtzee** The known game Kniffel or Yahtzee is one of the most sold dice games in the world. For the play one needs five dice. In every round one may throw the dice up to three times. Besides, one may put the dice aside that "fit" and throw the remaining ones again. After the third throw, at the latest, one must decide which field one wants to value with the result. There are two blocks on which one can put down the result. The first block is the "collective block." Here the dice with the suitable numbers are added and put down on the suitable field. If the sum of the points of all six fields is bigger than 63, a bonus of 35 points is added. The second block contains the fields "three of a kind," "four of a kind," "full house," "small street," "big street," "Kniffel/Yahtzee," and "chance". When you have "three of a kind" or "four of a kind," all eyes count. For "full house" you get 25 points, for the "small street" 30 and for the "big street" 40 points. There is the highest score (50) for the "Kniffel/Yahtzee". All eyes can be put down on "chance." The winner is the player who achieved the most points from both blocks and the bonus.

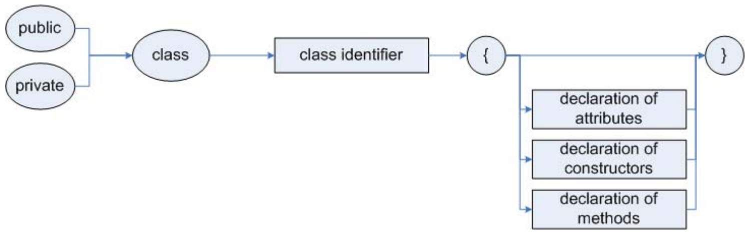
Table 7.2: Description of the project's tasks with the corresponding levels of previous knowledge (1-3)

[20] http://www.pabst-software.de/doku.php/programme:object-draw:start - last access 10.12.2014
[21] http://staruml.sourceforge.net/en/ - last access 10.12.2014

The second sheet introduces the basic concepts of object orientation: *object, class, attribute,* and *method*. It emphasizes the concepts of *data encapsulation* and *information hiding*, which students should adhere to as early as possible. All the concepts are illustrated by geometric objects and their graphical representations. The ideas are based on the textbook used by the Bavarian Gymnasien (Frey et al. 2004). The graphical representations are shown in Figures 7.1-7.3.

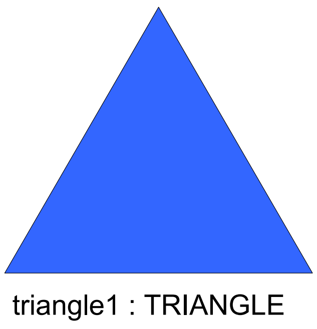
The third sheet presents the implementation of those concepts in Java. The concepts, inspired by (Müller and Weichert 2011), are described by text, as well as by syntax diagrams. Although the cognitive load theory (see Section 3.4) emphasizes the need for compact information representation to avoid extraneous load, the information is presented in plain text, as well as in a diagram. Nevertheless, the diagrams, as well as the textual description, contain all necessary information. So, the participants can choose the information representation by their personal preference. A sample is shown below. First, the textual description is given and then the graphical equivalent.

Class:
A class is initiated with the key word class, followed by the class identifier, which is usually capitalized. Enclosed by curly brackets, follows the class body with attributes, constructors and methods.

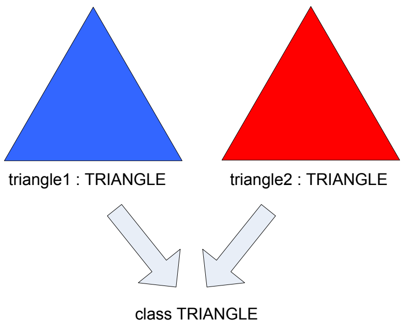


In addition, a small sample code is presented on the sheet. It only contains the basic structure of a Java class and the most common types of comment. The sample code makes the orientation within the code easier for the novice programmers.

The last sheet presents the concept of algorithms and the control structures (*sequence, conditional statement, loop*). Therefore, we introduce two different algorithm modeling techniques to enable the freshmen to get a graphical representation of their developed algorithms. First, we introduce structograms, which are closer to code than others. This technique is especially useful for participants who already have experience in programming. As a second modeling technique we introduce control flow charts, which provide quite an intuitive method of presenting algorithms. For both charts we only introduce the basic modules for the main concepts of programming: statement, sequence, iteration, and conditional statement. The description of the charts on the worksheets can be seen in Appendix B.1.

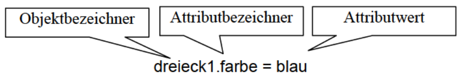


(a) Representation of objects

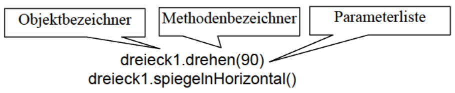


(b) Representation of building classes out of objects

Figure 7.1: Graphical representation of objects and classes on the worksheets



(a) Representation of the dot notation for attributes



(b) Representation of the dot notation for methods

Figure 7.2: Graphical representation of the dot notation

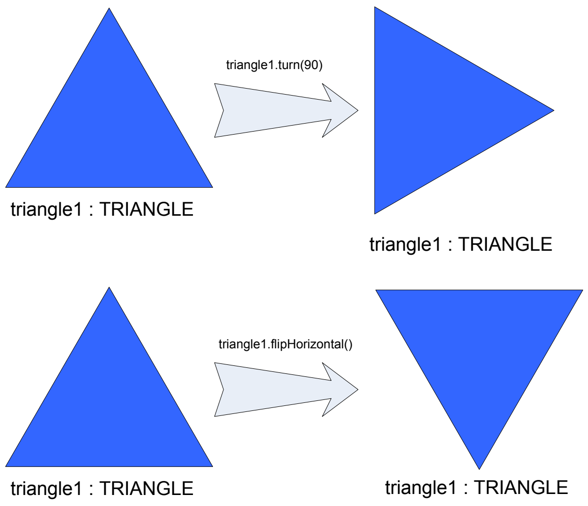


Figure 7.3: Graphical representation of the effects of methods on objects

After introducing the modeling aspects, we introduce the concepts for implementing algorithms in Java. In addition to the control structures of iteration and conditional statement, we introduce methods for input and output. The output is conducted via the standard output on the console. As the input in Java is difficult (cf. Mössenböck 2014, p. 315), we use the input framework[22] of Hanspeter Mössenböck. The class provides methods for reading values of different data types directly from the console simply by including the file `In.java` in the project.

The basic object-oriented concepts of the worksheets are presented in Figure 7.4. The concept specification map (see Section 6.2) shows the interdependencies of the concepts we introduced during the course. Obviously, the concept *object* has a central role. It specifies most of the other concepts. For detailed specifications expressed by the numbered circles see Appendix B.2.

Besides suitable materials and the most important concepts, selection of a suitable development environment is very important. We suggest that the students for their first programming steps use BlueJ due to the reasons mentioned by Bergin et al. (2005). BlueJ provides a simple sample project that uses geometric figures such as squares, circles, or triangles. The students are asked to explore the object-oriented concepts by working with these objects. After their first steps in object orientation, they are asked to build a model of their small project and implement it in BlueJ. Towards the end of the course, the students have a choice to switch over to Eclipse. When doing so, they are advised to use the main-method of Java to run the project. Students who complete their assignments in a short time are given an extra task by the tutoring student. The participants are encouraged to build a simple graphical user-interface for their project. Again, the tutors are asked to not provide direct instruction, but to give advice where the relevant information can be found.

As we wanted to investigate the development of novice programmers' knowledge and abilities we decided to let the participants work on their own. Hanks (2007) conducted a study on pairing novice programmers with the result that while "pair programming has been shown to provide many pedagogical benefits to students who are learning to program, it appears that pairing students still struggle with the same types of problems as students working by themselves" (Hanks 2007, p. 162). Nevertheless, students who work together get stuck less than students who work alone. For this reason, we utilized the tutoring system to help students over their first serious hurdles at the beginning of the course. With this approach we were also able to gather data representing the ability and knowledge development of each student.

[22]http://ssw.jku.at/JavaBuch/#InOut - last access 10.12.2014

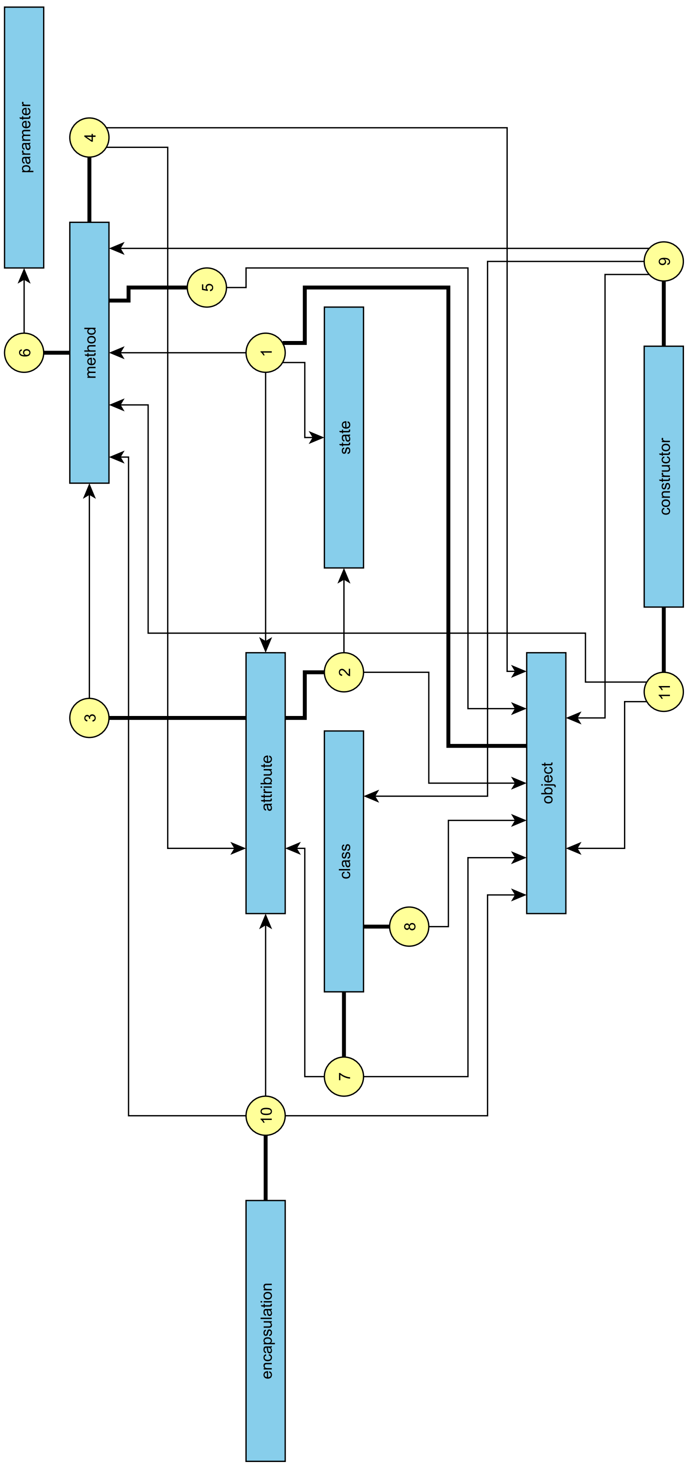


Figure 7.4: Content specification map for the basic object-oriented concepts of the worksheets presented during the course

7.4 Data Gathering

During the four runs of the preprojects we gathered a lot of data from the participants. This section shows what data was collected and how. Data gathering can be divided into three parts. Data was first collected during the enrollment procedure for the course. The second part of the data was collected through a survey at the end of each course. The third type of data was gathered through the evaluation of the "products" that the participants produced during the course; for example, concept maps or program code.

Before the course started, students enrolled and had to complete a short survey on their personal data. Because the students of the course were divided into homogeneous groups with regard to their prior programming experience, they had to self-assess their programming skills when they registered for the course, as described in the preceding section. In addition to the previous-knowledge level, the students were asked for information about their study path. The information about the knowledge level was mandatory, whereas indicating their study path was voluntary.

The second survey, which the students had to fill in, took place after each run of the course. The survey was completed with pen and paper. In each run, except the run in 2008, the results of the survey could be connected to the other results by identifying each participant with a number. The questionnaire was divided into five sets of questions.

The first questions covers personal data. This includes questions on gender (**male**/ **female**) and age (**under 20**, **20-25**, and **over 25 years**). Furthermore, we once again asked for the study path giving the options **computer science**, **economical computer-science**, **biological computer-science**, **computer science for teaching**, or **others**. With the next questions we assigned the students to a specific group by asking them for the time slot (**1-4**) of their course and for a self-assessment of their previous knowledge (**1-3**, as described in Section 7.3.2). The last two questions of this set covers their previous computer science education and the origin of the participant. The possible answers for the education question included **intensive course**, **basic course**, **compulsory subject**, **elective subject**, or **none**. The answers for the last question can be the abbreviations of the German federal states or an open answer when it's a foreign country.

The second set of questions asked for attitude towards the organization of the course. We asked about the time of announcement, the organization in general, and the schedule of the course. In the 2011 winter term we added a question about satisfaction with the size of the group. Each question was answered on a scale of one to five. The questions around organizational purposes are excluded in further research and were only for evaluation of the courses.

The third set of questions covers satisfaction of the worksheets. On a scale of five, the following questions had to be answered. The first question covers the amount of information on the worksheet (high to low), the second question gathers the understandability on the information of the worksheets (good to bad). The last question referred to how detailed the worksheets were (not detailed to too detailed). The fourth question set was related to the peer tutor. The first question gathers information about the helpfulness of the tutor. The answers were given on a scale from very helpful to not helpful at all. Another question asked whether the help from the peer tutor was understandable. Again, question sets three and four were not used in the investigations. The third question set was posed only for evaluation reasons. Contrary, the results of the fourth set are not surprising. The tutor was assessed as being helpful and the help was also understandable. This was reported by almost all participants. Thus, the answers provided no additional information on the population.

The last and most interesting category covers the self-assessment of the knowledge the participants had gained during the course. The first question asked if the participants had learned something in general during the course. The scale ranged from "a lot" to "very little". The second question investigates the self-assessment of their knowledge of Java after the preprojects. The participants assessed their abilities on a scale from "not present" to "program on my own". The last question gathers the understanding of the concepts of object orientation. The students assessed themselves on a scale of "understand" to "not understand at all". In general, the participants could answer on a rating scale from 1 to 5 with the ranges described above. Although, the labels of the steps between the endings are not presented in the survey, they are assumed to be clear for all participants. Resulting from this, the answers to the questions can be assumed to be interval scaled, although they are originally ordinal scaled. In contrast to the first two questions, the answers to the last question were coded from 5 to 1 to have the same direction of low to high in the comparative analysis and graphical representation.

In a small section at the end of the survey the students were asked for a small open feedback on the course. A sample questionnaire of the 2011 winter term can be seen in Appendix B.3.

Besides the survey, we collected all the program codes that the participants produced during the course (for examples, see Appendix B.6). The size of the code samples range from only a few lines of code to several classes with code for a GUI. For gathering the source lines of code (SLOC), a toolkit[23] from the University of Southern California was used. It is based on a measurement framework that was published by Park et al. (1992). More precisely, only the logical SLOC is applied. Logical SLOC intend to measure statements. Furthermore, they are not sensitive to format and style conventions. On the other side, they are dependent on the language used. The counting rules for Java can be found in Table 7.3. The SLOC were counted for each file produced by the participants of the course. Afterwards, if there were different versions of the same project, only the most complete version was included in the investigation. The other versions were dropped out. In a last step, the total number of logical SLOC was counted for each participant.

[23]http://csse.usc.edu/ucc_wp/ - last access 10.12.2014

No.	Structure	Order of prec.	Logical SLOC rules	Comments
R01	"for," "while," "foreach," or "if" statement	1	Count once	"While" is an independent statement.
R02	do {...} while (...); statement	2	Count once	Braces {...} and semicolon ; used with this statement are not counted.
R03	Statements ending with a semicolon	3	Count once per statement, including empty statements	Semicolons within "for" statements are not counted. Semicolons used with R01 and R02 are not counted.
R04	Block delimiters, braces {...}	4	Count once per pair of braces {..}, except where a closing brace is followed by a semicolon, i.e. }; or an opening brace comes after a keyword "else".	Braces used with R01 and R02 are not counted. Function definition is counted once since it is followed by {...}.
R05	Compiler directive	5	Count once per directive	

Table 7.3: Logical SLOC counting rules (Park et al. 1992)

For the analysis of the code we examined a list of concepts for the concept maps described below. In a first run on the data of the 2009 winter term, we simply gathered the concepts mentioned on the worksheets. The implementation of the concept descriptions of the worksheets was the main focus. To find the concepts in the code, several "code items" were examined. If an item could be found in the code, the code was 1-rated for this concept. This first item set can be seen in the following list:

C1) **Order of attributes, constructors, and methods in class definitions:** On the second sheet we presented a code sample that showed the correct order of attributes, constructors, and methods in a class definition.

C2) **Initialization of attributes with default values in the constructor:** All attributes were set to a default value; arrays were initialized.

C3) **Constructors with parameters:** In the section about constructors on the worksheet, the possibility of overloading a constructor was shown.

C4) **Initialization of attributes with default values:** We stressed the initialization of all attributes before they were used.

C5) **Return values of methods:** Did the students use the "return" statement to pass values from methods or did they only operate on global variables?

C6) **Parameters of methods:** Did the students use input parameters or operate on global variables or attributes?

C7) **Access modifiers of methods:** Not all methods have to be public. Did the students mark some of them as private?

C8) **Arrays:** Did the students use arrays?

C9) **Self reference by "this":** Using the same name for attributes and parameters in methods can improve the readability of the code. Did the students use "this" in constructors or methods?

C10) **Main-method:** At the beginning of the preprojects the students used BlueJ. This IDE allows the direct interactive call of methods without implementing a "main" method. At the end of their work, the students were asked to execute their programs without the usage of BlueJ, so they had to implement a "main"-method.

C11) **Conditional statement:** The students should use the "if" construct with "else" when necessary (instead of two separated "if" constructs).

C12) **"While-do" repetition:** "While-do" is the first form of repetition that was explained on the worksheets.

C13) **"Do-while" repetition:** The "Do-while" repetition was presented on the worksheets after "While-do".

C14) **"For" repetition:** The "for" repetition was the last form of repetition on the worksheet. The students should use this type if the total amount of repeats is already known before the first one starts.

As these items are related to the "implementation" of the worksheets and not with implementation of programming concepts in general, a revision of the concept list and corresponding "code items" was completed after the first run. Of course, there are many ways in which program codes can be analyzed. As we are interested in the relation to the collected concept maps, we tried to conceive a systematic approach to help us identify whether or not a concept is included and actually applied in the program code of a given project. Problems that could occur are expressed in Section 7.1.5.

In addition to the concepts extracted from the worksheets, we included the list of the "quarks of object-oriented development" presented by (Armstrong 2006). Besides the study of Armstrong (2006) and the list presented here, there are several other studies presenting lists of concepts or topics related to object orientation. Goldman et al. (2008) list three main topics (programming fundamentals, discrete math, and logic design) that are built from a Delphi process. They interviewed experts for concepts that are important for an introductory course. The experts rated their concepts with regard to importance and difficulty. As the final list mainly contains topics that are to be taught in an introductory course, the list is not applicable to this investigation since a list of concepts is needed. Nevertheless, some topics can be either mapped directly (for example, *inheritance*) or indirectly (for example, *object* and *class*).

Tew and Guzdial's study 2010 on a validated assessment of programming concepts tried to define a test for fundamental concepts based on multiple choice questions. The selection of the concepts is based on a document analysis of different types of texts including textbooks and the ACM/IEEE curriculum (see description in Section 5.3.1). Tew and Guzdial (2010) finally introduced a list of ten concepts related to an introductory computer-science course that are not dependent on a specific language. If this list is compared to the one underlying our investigation, exact matches such as *arrays, selection statement* and *parameters* are found. Other concepts are given more detail than we in our study, such as the *loop statements* that were divided into definite and indefinite loops. Contrarily, others are summarized to a fundamental top category such as *fundamentals* including *variables* or *assignments*. An additional general category, *object-oriented basics*, contains *class*, *methods*, and similar concepts.

The selected concepts have to fulfill two preconditions. First, they have to be expressible in the externalization of cognitive knowledge. So, there has to be a theoretical basis for the concept that has an interdependency on other concepts. In contrast, the concepts have to be observable in the program code. This is why some "quarks" have been changed; for example, *polymorphism* to *overloading*. Others such as *object orientation* are kept or split up in a "cognitive" and "observable" instance, such as it was done with *data encapsulation* and *access modifier*. The complete list of the concepts can be seen in the following Table 7.4, while the complete questionnaire for the assessment of the concept maps can be found in Appendix B.4.

Abbr.	Concept	Abbr.	Concept	Abbr.	Concept
AM	access modifier	CO	constructor	ME	method
AR	arrays	DE	data encapsulation	OB	object
AG	assignment	DT	data type	OO	object orientation
AC	association	IN	inheritance	OP	operators
AT	attribute	IS	initialization	OV	overloading
CL	class	IT	instance	PA	parameter
CS	conditional statement	LO	loop statement	ST	state

Table 7.4: List of the identified programming concepts

In the present study no classical knowledge test could be formulated to evaluate more than the conceptual knowledge of the participants; that is, the concepts that the participants really applied in their programming. To evaluate the program code of the participants, a methodology had to be found to perform a code analysis based on concepts related to programming. Because of that, items are formulated that can be rated if the concept is implemented. However, as a concept can be implemented in different ways, a complete match of all ways how the concepts can be implemented in the program code must first be found.

For the code analysis, possible applications of the concepts are investigated first. For example, the concept *constructor* can be present in the form of **using a constructor** (i.e., by creating an object) or in the form of **defining a constructor**. According to this procedure, the properties for every concept that is observable in the code are identified and it is shown that the respective concept is applied. Afterwards, these properties are categorized and each category represented by an item that can be rated with yes/no (1/0) while analyzing the programming code.

Some of the concepts have to be excluded from the beginning due to different reasons: *object orientation*, *class*, *data type* and *instance*. The first exclusion is because in the context of Java, object orientation is present by design. The next two exclusions occur because the use of Eclipse makes it impossible to distinguish between "implementation by the student" and "implementation forced by the IDE". Finally, the last is excluded because it is included in the concept *object*.

In the end there are 39 code items. Some turned out to be trivial, as they are 1-rated in nearly all datasets. For example, the concept *method* consists of the two categories *using a method* and *defining a method*. However, nearly every project used a method, because they were all using the input method to read values from the console.

In Table 7.5 all the items are shown that, from here on, are called "code items". The abbreviations indicate which concept the item relates to, which is shown in Table 7.4. Additionally, for every item the criteria for the 1-rating of the code is shown. Table 7.5 represents a variation of the table presented in (Berges et al. 2012).

ID	Code item	Criteria for 1-rating
	The code contains...	
IN1	...inheritance from existing classes	The keyword `extend` can be found in the code combined with an existing class such as *JFrame*.
IN2	...a manually created inheritance hierarchy	The keyword `extend` can be found in the code combined with a class that is defined within the project.
ME1	...a method call	There is a method call in code, except the standard output to the console.
ME2	...a method declaration	There is any method declaration in the code. Static methods are excluded.
ME3	...a return value in a method	The keyword `return` can be found in the code.
AG1	...an assignment	There is an assignment of a value to an attribute or variable.

Table 7.5: List of code items with the criteria for 1-rating - continued on next page

ID	Code item	Criteria for 1-rating
	The code contains...	
CO1	...a declaration of a new constructor	There is a declaration of a constructor that is not the empty standard constructor provided by Java itself.
CO2	...a call of a constructor	The keyword `new` in combination with a class name can be found in the code.
ST1	...a possibility to save the state of an object	There are attributes defined in the code.
ST2	...a possibility to change the state of an object	There are methods declared in the code.
ST3	...a possibility to use the state of an object	The attributes' values of an object are accessed either directly or with the help of methods
AC1	...an association between classes	There are attributes or variables in the code that have a non-primitive data type.
AC2	...any use of associations between classes	The values of attributes or variables with non-primitive data types are used in the code.
DE1	...attributes with a visibility other than public or default	There are the keywords `private` or `protected` in the code.
OP1	...an assignment operator	There is a single = in the code.
OP2	...any logical operators	There is one of the following operators in the code: &, &&, |, || or !.
OP3	...any other operators, apart from the assignment or logical operators	There are operators such as +, -, % in the code.
AR1	...an array with pre-initialization	There is an array declaration with predefined values in the code.
AR2	...an array without pre-initialization	There is an array definition without initialization in the code.
AR3	...any access of the elements of an array	There is a variable or attribute name followed by [#] (# is a number) in the code.
AR4	...an array initialization with *new*	The keyword `new` combined with a data type and [#] (# is a number) can be found in the code.
AR5	...methods of the class `Arrays`	The word `Arrays` can be found in the code.

Table 7.5: (contd.) List of code items with the criteria for 1-rating - continued on next page

ID	Code item The code contains...	Criteria for 1-rating
IS1	...an explicit initialization of the attributes	To all the attributes an initial value is assigned either directly in the declaration or within the constructor.
PA1	...a method call with parameters	A method with parameters is called in the code, except the standard output of Java.
PA2	...any method declarations with parameters	There is a method declaration in the code that contains parameters.
PA3	...a method where the parameters are used in the method body	There are methods where the parameters are used within the method body. For example, their values are assigned to attributes.
AT1	...attributes	There is a declaration of attributes in the code.
AT2	...an access to attributes of other classes	Either the attributes of an object of another class or methods of an object of another class that access foreign attributes are used in the code.
AT3	...an access to the attributes of their own class	The values of the attributes are used within the code.
CS1	...an if-statement without else	There is the keyword `if` in the code and no `else`.
CS2	...an if-statement with else	There are the keywords `if` and `else` in the code.
CS3	...a switch-statement	There is the keyword `switch` in the code.
OB1	...a declaration of any object attribute or variable	There are attributes of variables with a non-primitive data type in the code.
OB2	...any use of a declared object	The attributes or variables with non-primitive data types are used in the code.
OB3	...a reference to its own object using *this*	There is the keyword 7`this` in the code.
OV1	...a declaration of an overloaded method	A method is declared at least twice. The method can either be one of the participant's or one of the predefined Java methods.
OV2	...any use of an overloaded method	There is a call of a method in the code that is overloaded. The standard output is excluded.

Table 7.5: (contd.) List of code items with the criteria for 1-rating - continued on next page

ID	Code item	Criteria for 1-rating
	The code contains...	
LO1	...loops	There are the keywords `for` or `while` in the code.
AM1	...the access modifiers *public*, *private*, or *protected*	There are the keywords `public`, `private`, or `protected` in the code.

Table 7.5: (contd.) List of code items with the criteria for 1-rating

The list of the code items provides a straight-forward mapping of concepts to code items. However, a concept may be typically related to more than one item, but any item belongs to exactly one concept. To get an average score for the application of a concept, which is represented by a set of code items, suitable formulas have to be found. Basically, there are five ways how a concept is represented by several items. Because of that, we could derive five different formulas for the combination of the item scores.

(1) The concept is assumed to be applied if at least one of the corresponding code items is scored with 1. The scores of the code items are combined by a Boolean OR-function.

(2) The concept can be divided into several components and each component is represented by one code item. The score for the concept is calculated as the average over the scores of the code item.

(3) There is a clear hierarchy between two items (e.g., to use a parameter of a method, you have to declare a method with parameters first). It turned out that in these cases the most suitable score for the concept is also the average over both items.

(4) Again, there is a hierarchy between the items of a concept. In contrast to (3), one item is so important for the implementation of the concept that the concept gets the value 1 if the item is implemented. The second item has a lower priority and the concept is rated with 0.5 if only this item is implemented.

(5) There is a very strong relationship between items of a specific concept. In this case all items must be fulfilled. In this case, the scores of the items are combined by a logical AND operation.

By using these five methods (in rare cases a sequential combination of them), we finally derived a mapping from the scores of the code items belonging to a single concept to a value between 0 and 1. This can be regarded as a score that indicates the implementation of this concept in the program code. The complete list of the formulas can be seen in Table 7.6.

Concept	Formula
AM	AM1
AR	**OR**(AR1,AR2,AR4)/2+**OR**(AR3,AR5)
AG	AG1
AC	(AC1+AC2)/2
AT	(AT2+(AT1+AT3)/2)/2
CS	**OR**(CS1,CS2,CS3)
CO	**IF**(CO1,1,**IF**(CO2,0.5,0))
DE	DE1
IN	**OR**(IN1,IN2)
IS	IS1
LO	LO1
ME	(ME1+ME2+M3)/3
OB	(OB1+**OR**(OB2,OB3))/2
OP	(OP1+OP2+OP3)/3
OV	**IF**(OV1,1,**IF**(OV2,0.5,0))
PA	(PA1+(PA2+PA3)/2)/2
ST	ST1/2+**OR**(ST2,ST3)/2

Table 7.6: List of the formulas to calculate the score of the programming concepts

In addition to the raw programming abilities that the freshmen have or acquire during the course, we wanted to explore the differences between knowing and doing in programming. Therefore, we gained concept maps to externalize the conceptual knowledge. The concept maps – which are based on a list of pre-defined concepts –are drawn on paper and then digitalized with the yEd-Editor.

The questionnaires are, again, numbered so that all results can in the end be combined. In the first step the students are asked to choose whether they know a concept or not. The aim of this is to make them remember and reflect on the concepts in order that they avoid simply guessing of associations. If they know a concept, they should give an answer on how they gained the knowledge thereof. The possible answers are: “known before,” “known by the tutor,” “known by own inquiry,” “known from any other source”. The questionnaires for the pre- and the post tests are the same. On the second page the freshmen were given an example map, because we were unable to teach the students how to draw a concept map as required in theory (see Section 4.1).

After digitalizing the concept maps, a list of associations was formed –represented by an edge and the two connected concepts. A score was also assigned to each concept association triplet according to the following scheme:

- If the association is a correct statement, it is scored with 2.
- If the association forms a statement that is clearly wrong or the meaning of the statement could not be understood, it is scored with 0.
- If none of the above two conditions apply, the association is scored with 1.

Edges with no label were excluded from the analysis altogether. The grading scheme was validated by having three experts grade a randomly chosen subset of the edges after having explained the grading scheme to them. For intercoder reliability, Krippendorff's alpha coefficient (cf. Krippendorff 2004, 2011) is calculated. The value is 0.62 for the reliability measured between all raters. A further analysis shows that the most deviations occur between the grades 2 and 1. For that reason, edges with a grade of 1 are skipped in the following analysis.

In theory, our data for each participant should consist of four parts. We collected two concept maps (one drawn before and one after the course) and a questionnaire with basic personal data and the prior knowledge. Additionally, we have the source code of the programs that were written by the students during the course.

Nevertheless, the gathered data based on the minimally invasive programming courses is quite different over the years. Therefore, not all datasets are comparable to each other. An overview of the different data gathered from each student during the courses is shown in Table 7.7

Year	Personal Data	Program Code	Pre Concept Map	Post Concept Map	Student Questions
2008	Y	Y	N	N	N
2009	Y	Y	N	N	N
2010	Y	Y	Y	Y	Y
2011	Y	Y	Y	N	N

Table 7.7: Overview of the data gathered from each student over the years

In the first year we gathered the personal data independently from the program code. Therefore, no linking between the results is possible. The code is evaluated only for testing purposes. The same is done with the personal data. In all later runs the results are combined by a numerical identifier. The information we examined from each run is taken to improve data gathering in later runs. Linking of the results is done after the first run. Identification of the tutor for each participant is done after the second run when we realized that the tutor could have a strong influence on the results.

For the various investigations that were done on the minimally invasive programming courses, different sets of data are needed. Table 7.8 presents the number of possible datasets. In the first column the number of datasets, where the program code could be mapped to the personal data in the survey are shown – data needed in Section

7.5.1. The second column presents the students who have a concept map for previous knowledge, a concept map drawn after the course, and personal data – data needed in Sections 7.5.2 and 7.5.3. The third column shows the number of datasets where a concept map drawn at the end of the course, the personal data, and the program code are present – data needed in Section 7.5.4. The last column contains the datasets with program code – data needed in Section 7.6.

Year	Code of Novices	Knowledge of Novices	Differences Knowing/Doing	Code Evaluation
2008	0	0	0	0
2009	92	0	0	93
2010	103	82	89	112
2011	128	0	0	145

Table 7.8: Number of possible datasets for the different investigations over the years

One of the major problems in the investigation of the first two runs is the fact that the effect the peer tutor had on the results could not be reliably determined. Therefore, each tutor participated in at least two groups: one consisting of students with previous knowledge and one of students without. To find the influence of the tutor in the results, each tutor was assigned a color. The questionnaires of the students in each group was then in the color of the corresponding tutor.

Besides the results of the participants themselves, data about the complete groups were gathered. In the first run in the 2008 winter term, some groups were filmed. This was done to view the process of a participant in programming a project like the given one. Four groups with different previous knowledge levels were filmed. The group was filmed with a wide scope since there was only one camera. When the video sequence was analyzed, it was quite difficult to see what a single student was doing. Because of this, video filming of the groups was not pursued.

Another investigation performed on the whole group focused on the questions that the participants asked the tutors. Therefore, the tutors were requested to write a report of all questions they were asked. To map the questions to previous knowledge, the tutors wrote down the identifier of the student posing the question. Additionally, they had to write down their type of response. The options were “answering by a hint,” “answering by introducing a part of the code,” or “answering by recommending further information like the Java reference”. For the last answer type three options were given: “Java ist auch eine Insel (book),” “Java documentation/reference,” and “Help in the Java-IDE (Eclipse)”. The report form can be seen in Appendix B.5.

In the 2010 winter term, 392 questions in total were collected. They are all digitalized with the number of the asking student, the question, and a coding of the type of answer. All answers, including the three options of type three are coded with 0 or 1. The complete vector of types is treated as a binary number and converted to a decimal number in the range of 0 to 31.

In the next step the questions are rated on their relation to the project/worksheets (1) or to programming aspects (2). The extraction of the 0-rated (kind of nonsense) and 1-rated questions lead to a list of 182 questions (see Appendix B.8). In addition to the categories that are on the report form, the previous knowledge of the asking participant has been added. Furthermore, each question is coded with the main concept (see Table 7.4) the question relates to.

7.5 Analysis of Novice Programmers’ Knowledge and Abilities

The next subsections describe the results of different aspects related to novice programmers’ knowledge and abilities. Besides the investigation of program codes, externalizations of cognitive knowledge through concept maps build the basis for this. More precisely, differences in the program codes of novice programmers are investigated (Section 7.5.1). Furthermore, development of knowledge (Section 7.5.2) and misconceptions (Section 7.5.3) during a course, as well as the differences between knowledge of a concept and its application in the program code (Section 7.5.4), are investigated. Additionally, in the next section, the evaluation of the program code with the psychometric methodology of the item response theory is introduced (Section 7.6). All these experiments provide an answer to the research questions of how to evaluate novice programmers’ knowledge and abilities (**RQ6, RQ7,** and **RQ8**).

7.5.1 Differences in the Program Code

During the introductory courses that were based on the idea of minimally invasive programming courses, participants produced program code. As these code fragments have only a little influence from instruction – due to the course design – they are predestined for evaluating differences in novice programmers’ coding abilities. The presented research provides an approach for this, based on cluster analytic methodologies (see Section 4.2). The first analysis shown in (Hubwieser and Berges 2011) was improved one year later (Berges et al. 2012).

The first investigation of the program codes of 2009 that was published in (Hubwieser and Berges 2011) only focused on the implementation of concepts of the course materials. In contrast, the investigation of the program code for this thesis focusses on the concepts gathered during the 2010 and 2011 courses. As mentioned in Section 7.4 the score for the implementation of a concept is calculated out of the code items. The investigation described in this section concentrates on the novice programmers. Therefore, only the datasets of those participants without any previous knowledge are included. Starting with 323 datasets with programming code and personal data from the survey, all datasets are eliminated in which the previous knowledge is not in group one (“No previous programming knowledge”). The resulting 141 datasets still include those who have studied computer science in school. With the aim of investigating the students with absolutely no previous knowledge, the datasets with any contact to computer science in school are eliminated. This leads to the final 108 datasets analyzed in this thesis. The number of projects spreads over the years: 40 projects in 2009, 41 projects in 2010, and 27 projects in 2011.

In a first step the code items are investigated on their own. For each of the code items (see Table 7.4), the relative frequency of application in the program code is calculated. This calculation is first conducted on all datasets together and then separately for the years 2009 to 2011. The resulting values range from 0 to 1 with an average on all items of 0.57. The increasing order of the items in Figure 7.5 is only chosen to improve readability.

First, the mean scores of the different code items are analyzed. Looking at those items that have a mean value of less than 0.1 reveals that the two items of the concepts *inheritance* and *overloading* are below this line. This is not surprising as these concepts are not introduced on the worksheets and the investigated population has no previous knowledge. Another item below this line is **CS3**. Again, this is not a surprise, because the `switch-statement` that is the topic of the item is neither introduced on the sheet nor is it useful in the "Mastermind" project. The last two items that have a mean value below 0.1 are those related to the concept *array*. The first one is **AR1**, which covers the pre-initialization of an array. As we only see the code in its final state, it is possible that the pre-initialization was removed when implementing a constructor or a main-method. Over two thirds (68%) of the participants that have not applied any pre-initialization of arrays have defined their own constructor in the code. The last item is **AR5**, which covers the pre-defined class Arrays to handle operations on arrays. This is quite a useful but difficult way to operate on arrays. Nevertheless, there were some participants who looked up the needed sources to apply the class, for example, for sorting the array.

On the other hand, there are code items that have a mean value of more than 0.9. Obviously, there are again items that are trivial because they are described exactly on the worksheets. The first trivial item corresponds to the standard output of Java. Although the method *System.out.println()* is static, it is a *method call with parameters*. Because of that, the item **PA1** is implemented in almost every project and can be seen as trivial in this context. The second trivial item is the method call itself. Because the standard output is given as an idiom on the worksheet, it is excluded from the analysis of the method calls. Other methods that are applied are those for the input. As nearly every project contains an input of values, there is a relative frequency of 0.92 for **ME1**. Another set of quite trivial items covers the concept of operators. The assignment operator is used in almost every project (**OP1** and **AG1**). Other operators (**OP3**) such as arithmetic operators are also used in most projects. The most interesting item that has a mean value above 0.9 is the item that is related to the conditional statement (**CS1**). The simplest form without an alternative (**CS1**) is applied in 94% of the projects. The conditional statement with an alternative (**CS2**) is only applied in 78%, although it is convenient to use it in the code. Obviously, the correct implementation of an alternative is more complicated than the serial application of the conditional statement, as it was done in most of the cases when no alternative was implemented.

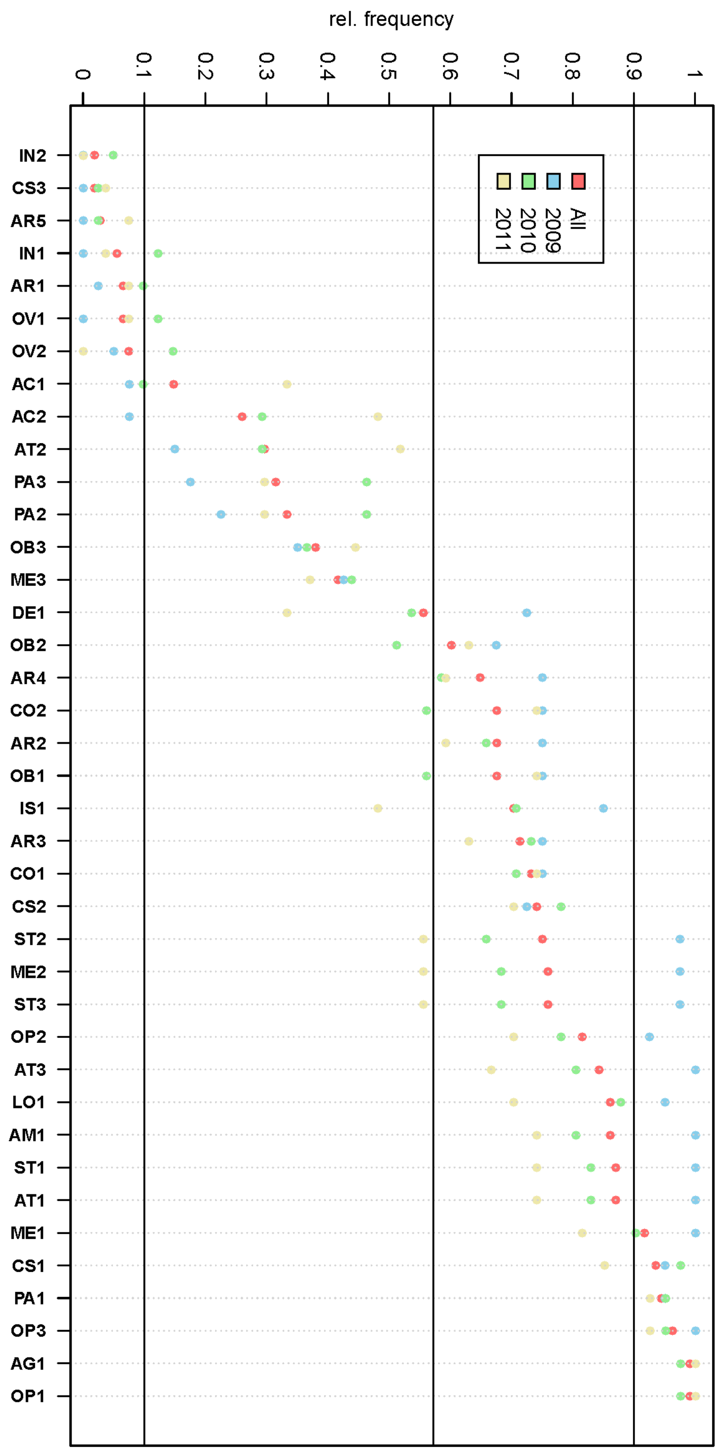


Figure 7.5: Relative frequencies of the code items over all student projects (red) and those from the years 2009 (blue), 2010 (green), and 2011 (nude)

Next, the items are investigated that have a mean value above the total mean value of 0.57. There are 24 items above and 15 below the mean value line. If we divide the items on their belonging to the programming paradigms, there are 11 items for the procedural paradigm (**AG1, OP1, OP2, OP3, AR2, AR3, AR4, IS1, CS1, CS2, LO1**), nine items for the object-oriented paradigm (**ME1, ME2, CO1, CO2, AT1, AT3, OB1, OB2, AM1**), and four items for both paradigms (**ST1, ST2, ST3, PA1**). Looking at the relative frequencies, 79% of the items are related to the procedural paradigm, 44% of the items for the object-oriented paradigm, and 71% of the mixture-items are above the mean. Concentrating on the "pure" items, it is obvious that the procedural paradigm is implemented more often. This is clearly caused by the course design. Although the participants were encouraged to design a model first, most of them started coding from scratch. In most cases, they either implemented one class with one method in BlueJ or, if they started with Eclipse from the beginning, they implemented only a main-method including all the code.

After analyzing the code items for all years, some differences among the participants within the years became evident. In 2009 we have more items over or under the quantiles of 90% or 10% (see Figure 7.5). In contrast to the analysis on all years, there are more method declarations in the code (**ME2**). Also, the items related to the state of an object (**ST1, ST2, ST3**) could be 1-rated in more than 90% of the projects. The logical operators are used quite often in the code (**OP2**). Furthermore, the access (**AT3**) and declaration (**AT1**) of attributes is more common than in the overall mean. Last, there are more loops (**LO1**) and access modifiers (**AM1**) in the code. Obviously, the participants are more extreme in their application of programming concepts as they have more items with a mean value below 0.1. In particular, there are fewer associations between classes (**AC1, AC2**). The participants seem to apply more concepts than the average, which is emphasized by the higher total mean value of 0.61. In the dataset of 2009, there are again 79% of the procedural items and 67% of the mixed items, but only 53% of the object-oriented items above the mean. Nevertheless, the concepts related to object orientation are applied more often. Comparing the implementation within the years, the concepts above the overall mean are implemented more often, while the concepts below the overall mean are implemented less.

In 2010, the same items have a mean value over 0.9 than in the overall case. Contrary to this case, slightly more inheritance is used in the code, so the item **IN1** is above the 0.1 line. The concept of overloading is applied more often, so the two corresponding items are above the last 10% as well. In contrast to the overall case, there are fewer associations in the code (**AC1**). The total mean value is nearly the same as it is in the overall case (56%). In general, fewer items are over the mean line; 79% of the procedural items and 67% of the mixture items. For the object-oriented items, only 31% are over the total mean. Although there are fewer items in the quantiles, the object orientation of the code in this turn is worse than the average of all turns, based on the concepts implemented. To summarize, object-oriented concepts are applied less, but, on the other hand, the concepts are more advanced.

While the top 10% of the items only contain four concepts, the last 10% of the items equal the overall case. In 2011 there are fewer projects with a method call and fewer application a conditional statement. The total mean value is only 53%, which means that in an average case the participants without any previous knowledge and with no education in computer science at school implement fewer concepts than in the previous two turns.

When comparing the years the application of the concepts become more equal. While in 2009 most concepts are above 90% or below 10%, in 2011 most items are distributed around the overall mean. In general, the advanced object-oriented items are located below 10% for all three years. Furthermore, the items related to object orientation are used less than those related to the procedural paradigm. Regarding the project descriptions, the assignments do not encourage the use of object orientation. Additionally, the use of BlueJ with its capability to execute each method on their own discourages the distribution of the functionality on several methods, and because of that, the notion of object orientation in general.

To emphasize the findings of the code items a comparison to the self-assessment of the novice programmers is conducted. In Figure 7.6 the mean values of the self-assessment on the learning gain are presented. Figure 7.7 shows the average assessment of Java knowledge, whereas Figure 7.8 shows the assessed knowledge of object orientation. Each of the figures is a bar chart with a set of bars for each year and one bar for all years. Each set is divided into three parts for the two groups of the cluster analysis presented below and one for all participants. The scale of the y-axis reaches from one to five with a mean value of 3 instead of 2.5, because the answers to the questions could not have a value below one.

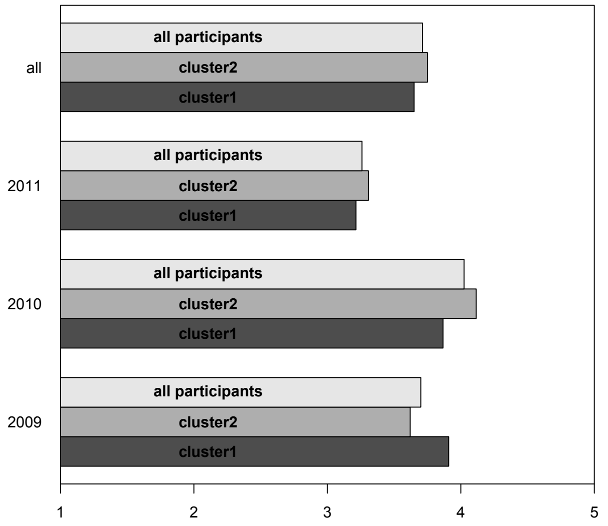


Figure 7.6: Average values of the novice programmers' self-assessment of the Learning Gain

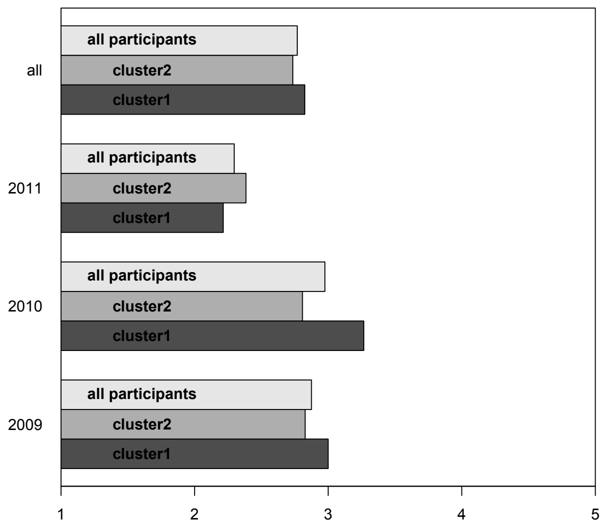


Figure 7.7: Average values of the novice programmers' self-assessment of the Java knowledge

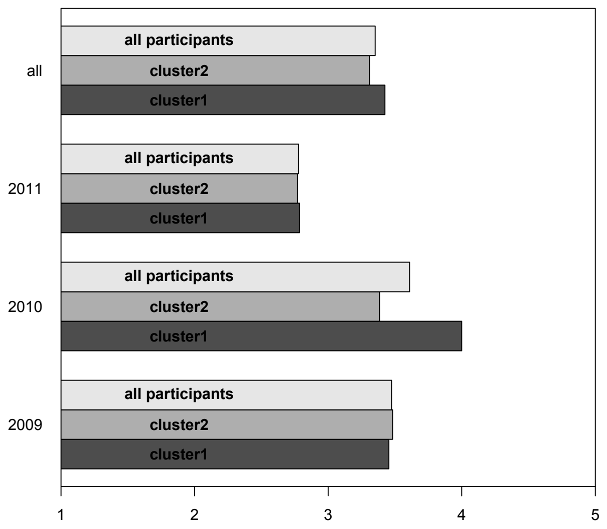


Figure 7.8: Average values of the novice programmers' self-assessment of the OO knowledge

In addition to the average values of the implemented concepts that are lower in the turn of 2011, the assessment of the object-oriented knowledge is lower than the average (see all participants in Figure 7.8). Also, the assessment of the learning gain and the Java knowledge is lower in 2011 (see all participants in Figures 7.6 and 7.7). The average value of the years 2009 and 2010 are nearly the same and vary around the total average, while the value of 2011 is lower than the other two years and lower than the average (see all participant in Figures 7.6-7.8).

After evaluating the code items on their own, focus is changed to the values for the concepts calculated by the formulas listed in Table 7.6. As described in Section 7.4, the items of each concept are put together and the result normalized to fit the range of 0 to 1. Some of the concepts have a strictly binary value of 0 or 1, while others have specific values in between. Nevertheless, the scale level of all categories is ordinal given that 1 is better than 0. By applying a cluster analysis, the novice programmers' abilities are investigated based on their implementation of the programming concepts.

For this purpose, two homogeneous groups are searched in the results based on the methodologies described in Section 4.2. First, the standard k-means method of GNU R is calculated. The results are skipped, however, because the cluster sizes between the two groups are very different. The same occurs when applying the k-medoids method "Partitioning Around Medoids (PAM)". Because the hierarchical methods have the advantage that the separation of the clusters can be done after the analysis, the groups are split on the basis of these methodologies.

As two clusters are needed with a distribution of the items of almost 1:2, the standard method of GNU R for the hierarchical analysis named *hclust* is chosen. Here, the linkage methods of Section 4.2 were all applied. The complete linkage method provided the most suitable cluster sizes. The resulting dendrogram (hierarchy tree) can be seen in Figure 7.9. Resulting from the hierarchical analysis and a separation on the highest distance level, the 108 datasets are divided into two groups. The first group includes 40 datasets and the second group 68.

For each group and category (concept) the mean value is calculated. As all items for one concept add up to 1, the mean values again range from 0 to 1. The results are shown in Figure 7.10. On the x-axis the concepts are listed by the abbreviations given in Table 7.4. For the sake of clarity, they are reordered into algorithmic and object-oriented groups. First, there are the concepts that belong to algorithm thinking, which are followed by the object-oriented concepts. The y-axis shows the mean values of the given range.

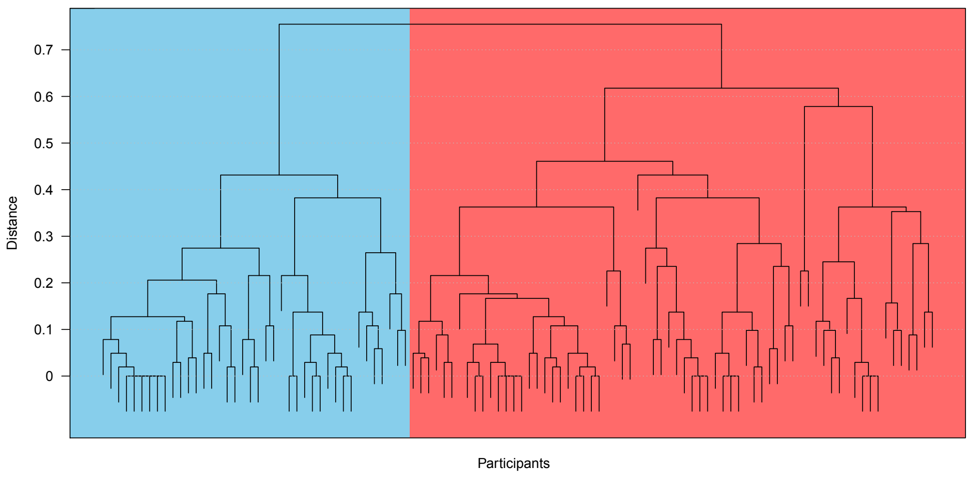


Figure 7.9: Dendrogram of the cluster analysis with a hierarchical clustering algorithm resulting in two clusters (red and blue)

To examine the differences between the groups, the code items belonging to a specific concept are investigated. Therefore, the concepts that have different mean values are sought. For some of the concepts this is obvious; for example, **array (AR)** or **object (OB)**. Nevertheless, for some concepts it is not that obvious. For this reason the p-values of a Welch two sample t-test for the mean values of each concept are calculated to obtain a statistically proven list of concepts to investigate. The hypotheses for the test are:

$H_0 : \mu_1 = \mu_2$: The mean values of the two groups are equal for a specific concept.
$H_1 : \mu_1 \neq \mu_2$: The mean values of the two groups are not equal for a specific concept.

In Table 7.9 the differences between the mean values and the corresponding p-values are shown. All concepts with a p-value smaller than 0.01(**) are considered significant, excluding the concepts with only one code item: **inheritance (IN), method (ME), state (ST), association (AC), array (AR), parameter (PA), attribute (AT), object (OB)**. Obviously, the largest differences are found for the concepts of object orientation. The differences for the procedural concepts are much lower and in most cases the differences are not even significant.

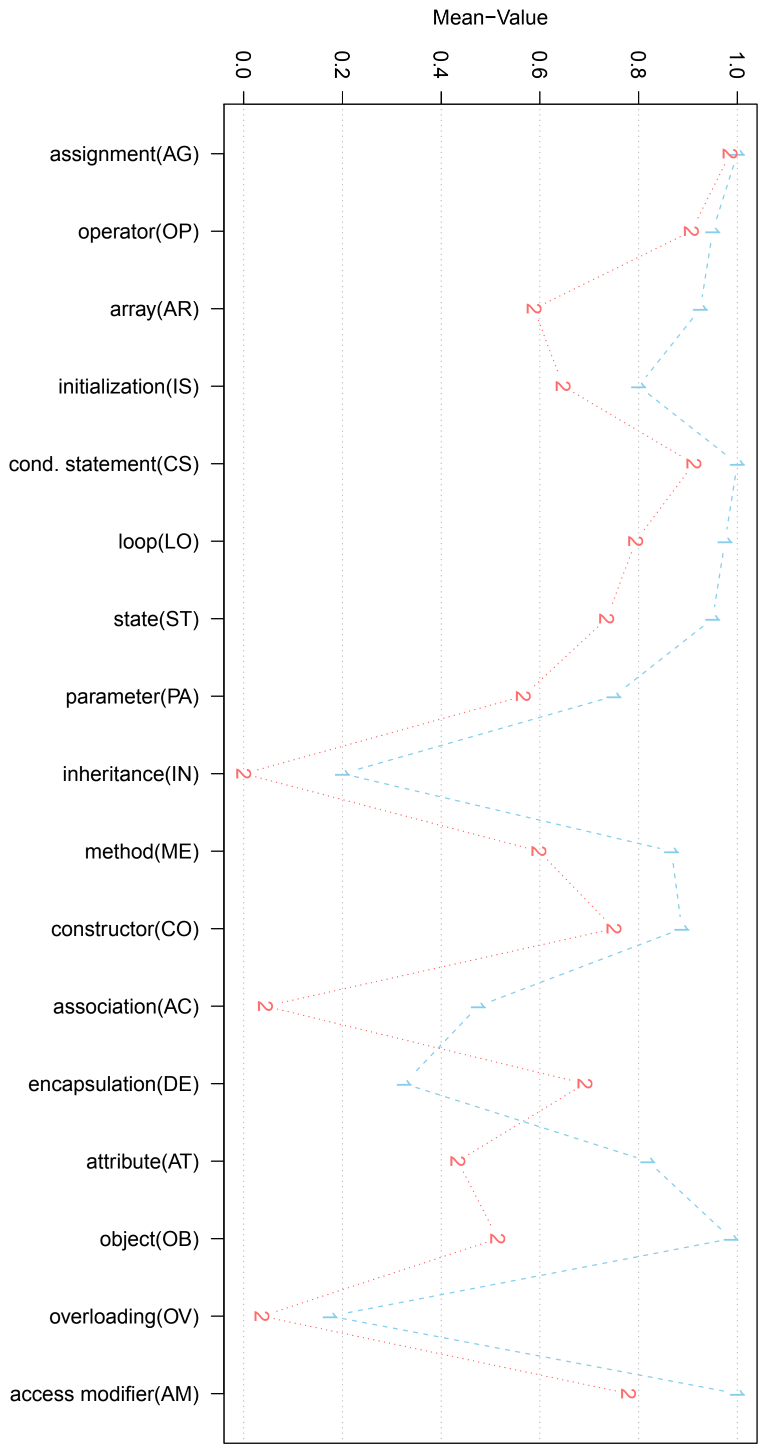


Figure 7.10: Different groups (more implemented concepts (1) vs. less implemented concepts (2)) of novice programmers

Concept	Difference		Concept	Difference	
	$\mu_1 - \mu_2$	p		$\mu_1 - \mu_2$	p
AG	0.01	0.32	ME	0.27**	1.13e-06
OP	0.04	0.17	CO	0.14*	0.02
AR	0.34**	1-25e-05	AC	0.43**	1.59e-07
IS	0.15	0.08	DE	-0.37**	1.97e-04
CS	0.09*	0.01	AT	0.38**	5.51e-12
LO	0.18**	1.51e-03	OB	0.47**	6.05e-12
ST	0.21**	1.86e-04	OV	0.14*	0.03
PA	0.18**	3.74e-04	AM	0.22**	4.67e-05
IN	0.2**	3.37e-03			

Table 7.9: Differences between the mean values of the two clusters and the corresponding p-values ($* < 0.05, ** < 0.01$) of the Welch two sample t-test on $\mu_1 = \mu_2$

Each of these remaining concepts is split up into its code items. In Figures 7.11 to 7.14 the occurrence of the items in the code is displayed for each cluster. Additionally, the frequency of participants that did not implement any item of the given concept is presented. On the x-axis the code items are displayed for both clusters, while the y-axis shows the relative frequency of the occurrence in the code.

To find differences in the items, the independence is calculated for two items each, in comparison with the two clusters. If there is a significant difference that is not resulting from the two clusters, there should be no correlation between the items. For this purpose, an χ^2-test is conducted on all item pairs in a specific concept. Here, all differences are presented that are significant to a p-value of 0.05(*). Furthermore, a Welch two-sample t-test is provided to find significant differences between the clusters, restricted to those participants that did not implement any of the items of a specific concept. The results of the tests are presented in the following list (t-test) and tables (χ^2-test).

Inheritance
mean value difference for "no items": 0.2** (p-value: 0.002)

no implementations of any item in cluster 2

Method
mean value difference for "no items": 0.03
(p-value: 0.188)

	ME1	ME2	ME3
ME1	1	0.452	0.002**
ME2		1	0.014
ME3			1

State
mean value difference for "no items":
0.21** (p-value: 4.51e-05)

	ST1	ST2	ST3
ST1	1	0.801	0.857
ST2		1	0.944
ST3			1

Association
mean value difference for "no items":
0.55** (p-value: 1.12e-08)

	AC1	AC2
AC1	1	0.455
AC2		1

Array
mean value difference for "no items":
0.34** (p-value: 6.26e-06)

	AR1	AR2	AR3	AR4	AR5
AR1	1	0.235	0.236	0.279	0.26
AR2		1	0.99	0.806	0.619
AR3			1	0.813	0.616
AR4				1	0.572
AR5					1

Parameter
mean value difference for "no items":
0.04* (p-value: 0.042)

	PA1	PA2	PA3
PA1	1	0.089	0.046*
PA2		1	0.782
PA3			1

Attribute
mean value difference for "no items":
0.19** (p-value: 8.61e-05)

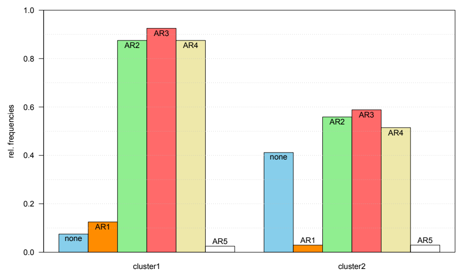
	AT1	AT2	AT3
AT1	1	1.532e-04**	0.967
AT2		1	1.824e-04**
AT3			1

Object
mean value difference for "no items":
0.41** (p-value: 1.41e-09)

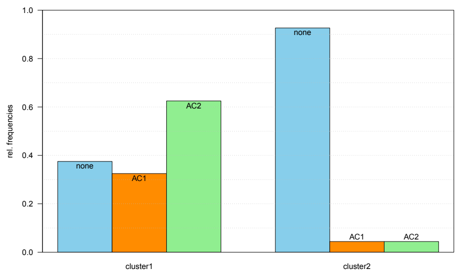
	OB1	OB2	OB3
OB1	1	0.552	0.0981
OB2		1	0.627
OB3			1

First, there are concepts that only differ in the value of any implemented items. The pairwise comparison of the items show no significant difference, except for the fact that the participants of cluster 2 implemented the items less often. All representatives of this group are shown in Figure 7.11. The underlying concepts are **attribute (AT), association (AC), state (ST), object (OB)**, and **inheritance (IN)**.

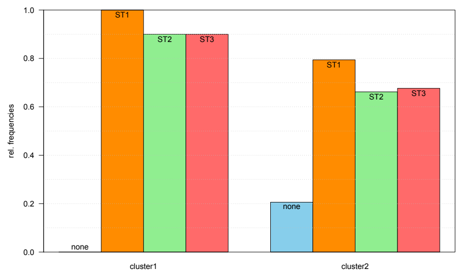
Interestingly, **inheritance (IN)** was only implemented by participants of cluster 1. For this reason no χ^2-test could be applied. However, the differences between the clusters are obvious (see Figure 7.11(d)). Concerning **object (OB)** and **state (ST)**, only in cluster 2 were there participants who did not implement any of the underlying items. The pairwise comparison shows no significant difference (seen in Figures 7.11(c) and (e)). The other two concepts, **array (AR)** and **association (AC)**, have participants who did not implement any of the items in both clusters. However, the mean values differ significantly. The χ^2-test is hardly applicable for either concept since the frequencies for some items are very low. This is the reason why a pairwise comparison is difficult to interpret (see Figures 7.11(a) and (b)).



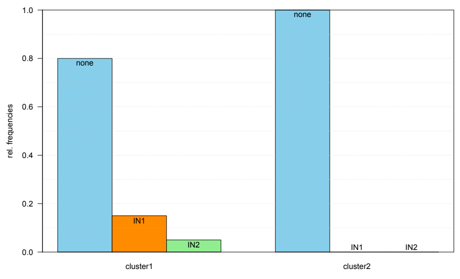
(a) array (AR)



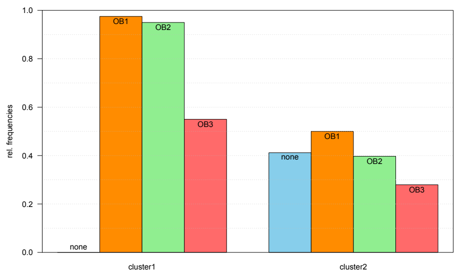
(b) association (AC)



(c) state (ST)



(d) inheritance (IN)



(e) object (OB)

Figure 7.11: Relative frequencies of the code items separated by the two clusters (cluster 1, cluster 2) only differing in the participants not implementing any item

Additionally, the underlying items of the concept **method (ME)** have significant differences between the two clusters concerning the pairwise comparison of the items. In contrast, the mean values of the participants not implementing any item do not differ significantly. In general, only a few participants implemented none of the items. Concerning the pairwise comparison, cluster 1 differs in the use of a return value in the methods (**ME3**). The call (**ME1**) and the declaration (**ME2**) of methods are used in the same way in both clusters (see Figure 7.12).

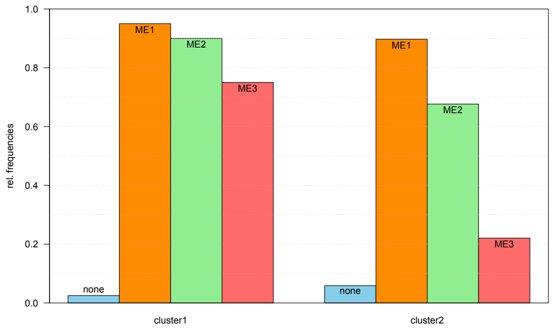


Figure 7.12: Relative frequency of the 1-rated code items for the concept *method* (ME) in relation to the participants of each cluster (cluster 1, cluster 2)

Last, the concepts **parameter (PA)** and **attribute (AT)** have significant differences in both tests. In both clusters, attributes are declared (**AT1**) and accessed in their own class (**AT3**) in the same way. But, the access of attributes of other classes through direct access or methods (**AT2**) differs, obviously (see Figure 7.13). Additionally, cluster 1 contains no participants who did not implement any of the underlying items.

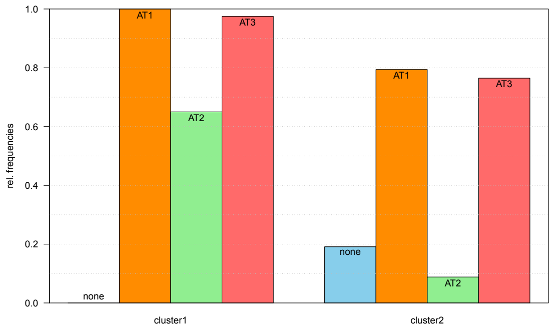


Figure 7.13: Relative frequency of the 1-rated code items for the concept *attribute* (AT) in relation to the participants of each cluster (cluster 1, cluster 2)

Similarly, the method call (**PA1**) with parameters is implemented in both groups. But, the method declaration with parameters (**PA2**) and the use of the declared parameters in the method body (**PA3**) is implemented less in cluster 2 compared to **PA1** (see Figure 7.14).

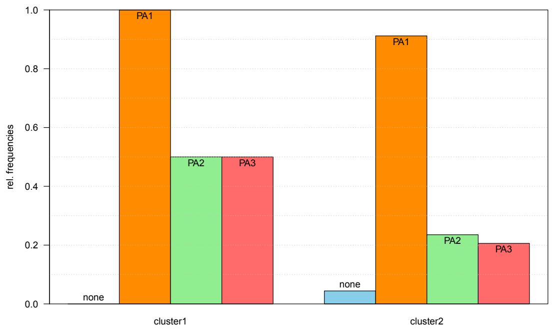


Figure 7.14: Relative frequency of the 1-rated code-items for the concept *parameter* (PA) in relation to the participants of each cluster (cluster1, cluster2)

Concerning the comparison of the concepts related to object orientation (for example, *object, method*, or *attribute*) with those related to the procedural paradigm (for example, *array, loop*, or *conditional statement*), it can be stated that the first cluster is more familiar with the concepts of object orientation. This corresponds to the results of the self-assessment on the understanding of object orientation (see Figure 7.8). For all runs the value of the self-assessment of cluster 1 is higher (but not significant (p=0.27)) than cluster 2. In all turns between 2009 and 2011 there is a difference, but not significant. Nevertheless, the data gives strong advice that the members of cluster 1 assess themselves better in the understanding of object orientation and implement more of the concepts in a more detailed way than the members of cluster 2.

Regarding the self-assessment (see Figure 7.7) of the general programming abilities in Java, we can see that there is no significant (p=0.3) difference between the two clusters in the question on the knowledge of Java. This corresponds mainly to the general programming concepts that belong to the procedural paradigm.

Despite the differences shown in the code, in the self-assessment there are no significant differences regarding the learning gain (see Figure 7.6).

7.5.2 Development of Knowledge

A second investigation on the novice programmers is related to the development of their knowledge about object orientation and programming (see Section 7.1.4 for related work). Therefore, the results of the investigation of the concept maps drawn in the 2010 term are compared. As there was only a previous test in 2011 – due to organizational reasons – these results cannot be included.

Before the start of the courses in the 2010 winter term, the participants were asked to draw concept maps on their knowledge of object orientation and programming. The concept list, as well as the analysis methodology, are described in Section 7.4. To investigate the development of the knowledge on object orientation and programming of novice programmers, we start with a list of 595 associations drawn by all participants in the pre-test. In a first step all associations that are not rated as completely correct (2) are eliminated. Since the interest for this thesis is only in evaluating the knowledge of novice programmers, all datasets that did not fit the pattern "no previous knowledge in programming (1)" and "no prior education in computer science (5)" are excluded. This results in a list of 30 associations (out of 16 concept maps) that are shown in Appendix B.7.1.

For comparing the pre- and post-tests of novice programmers, those datasets that contain results of both will be used. Again, all those associations that are drawn by students without any previous knowledge are considered. The resulting list of 221 valid (2-rated) associations can be found in Appendix B.7.3.

7.5.2.1 Previous Programming Knowledge

Before analyzing the knowledge development of novice programmers, the previous understanding of those concepts are looked at. As the edges of the maps have no direction, each item of the list in Appendix B.7.1 is investigated twice (60 edges in total).

First, the concepts that are associated in more than 10% of the 30 edges are investigated. These are mainly the basic concepts of object orientation. Besides *class, method, attribute*, and *object*, there is only the concept *data type*. In Table 7.10 these five concepts are listed with their associations. The number in parenthesis shows how much association between the given concepts is drawn in the maps.

Concept	Associations
class	object(3), association(1)
method	object(2), attribute(1), operator(1), conditional statement(1)
data type	object(2), attribute(1), parameter(1)
attribute	object(6), inheritance(1), assignment(1), method(1), data type(1)
object	attribute(6), class(3), method(2), data type (2), state(2), association(1), object orientation(1), constructor(1)

Table 7.10: Concepts with more than three associations drawn by novice programmers in the pre-test and their associated concepts

The connection between *object* and *attribute* seems to be intuitive, even for novice programmers. The second most drawn association is between object and class. Again, there seems to be an intuitive understanding of this association. Nearly all of the associations that are drawn more than once in the 16 maps connect the concept of *object* in a specific way. In the knowledge structure of the novice programmers, the concept of *object* seems to be very important. Obviously, this results from the intuitive understanding of object orientation, which is investigated in detail below.

The edges of the concept maps can be grouped together. Again, the direction of the edge is not of interest. Two edges are assumed to be in the same group when they combine the same concepts. Figure 7.15 shows the grouped map. All the concepts that are part of the concept maps of the novice programmers are included in the map. The edges are weighted by the number of their appearances. The weight of the edges is represented by the thickness of the lines. Singularities are left out as the concept mapping technique is not applicable for individual analysis (see Section 4.1)

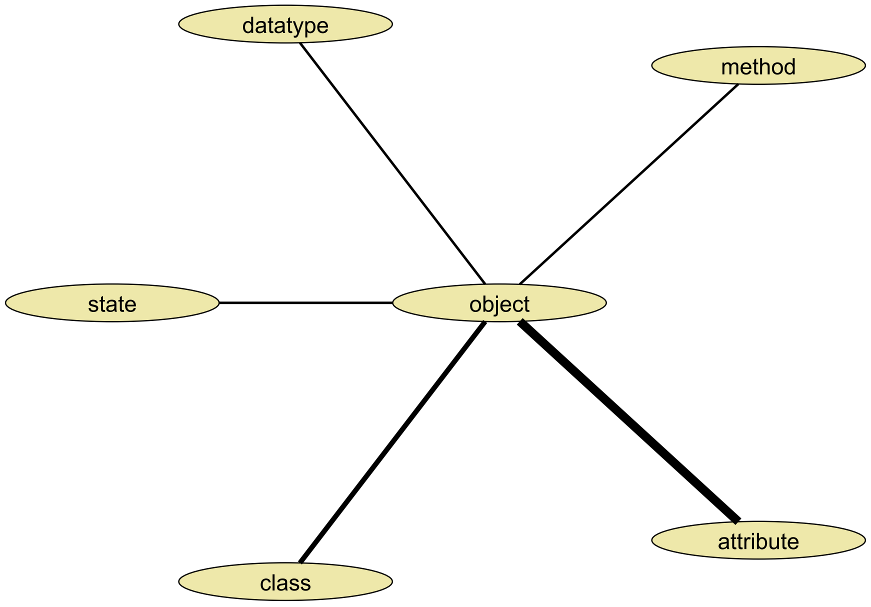


Figure 7.15: Combined graph of the 2-rated associations of the novice programmers in the pre-test – the thicker the line, the more connections have been found between the concepts

Obviously, there are only edges connected to the concept of *object* that are not a singularity. The method of drawing concept maps before the start of a course can give a good impression on how to start into object-oriented programming. Hence, the method can support the selection of suitable didactic methods, as mentioned in Section 5.1.1. Nevertheless, there were only sparse concept maps drawn by the novice programmers. The small number of maps that could be investigated has to be considered. The results only provide a clue for intuitive understanding of object-oriented notions.

In Table 7.11, all edge groups are shown that appear in more than one concept map. In addition to the concepts, the normalized edge labels are shown.

Concept 1	Concept 2	Edge label
object	class	assigns, belongs to, defines
object	method	works on, consists of
object	state	has, is in
object	data type	is, has
object	attribute	has, owns, defines

Table 7.11: Mostly drawn edges by novice programmers in the pre-test

As mentioned in the above paragraph, all edges have a connection to the concept *object*. The intuitive understanding that was expressed by the participating novice programmers is interesting:

Object belongs to class

Class defines object

Method works on object

Object consists of method

Object has state

Object is data type

Object owns attribute

If these items are compared with the basic concepts of Section 2.2, all items except inheritance and polymorphism are contained in the list. Some of the participants even have an idea of the concept of the state of an object.

A last investigation on the knowledge examined in the pre-test examines the self-assessment. Starting with the assumption that we have three categories, the participants had to self-assess their previous programming knowledge. The participants are clustered by the partitioning cluster analysis algorithm "Partitioning Around Medoids (PAM)" (see Section 4.2). The data basis for the clustering is once again a table with 1 for a concept and participant if there is a valid connection to this concept in the corresponding concept map of the pre-test. Figure 7.16 shows which partition of the cluster corresponds to which knowledge category. For example, 40% of cluster 2 is in category 1, 40% in category 2, and 20% in category 3. In general, assigning clusters to the self-assessment category is clear for clusters 1 and 3, as it can be expressed by a correlation of 0.71 (Spearman's rank correlation). If the second cluster is included in the correlation analysis, the coefficient decreases to 0.54.

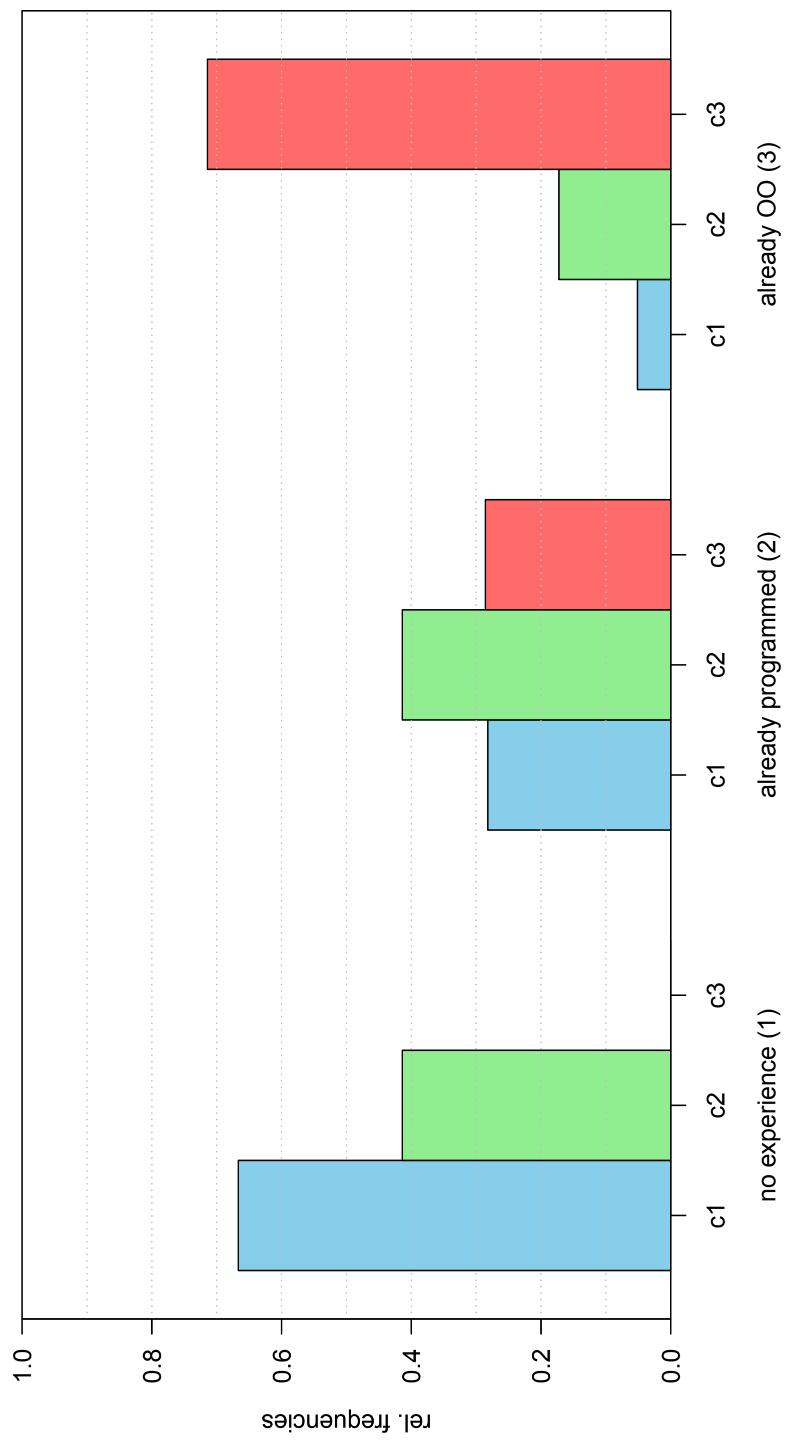


Figure 7.16: Partitioning of clusters from the pre-test on levels of previous knowledge

Because clustering itself only provides a partitioning of a given dataset, the characteristics of the clusters found in the data have to be examined. In Figure 7.17 the relative frequency of the appearance of a concept in the pre-test is drawn for each group. The groups without any pre-knowledge (1) and with experience in object-oriented programming (3) can be clearly separated. In general, group 3 – which assessed themselves as experienced – can externalize more correct rated edges in a concept map than those without any experience. The externalization of the knowledge related to object orientation is interesting. Cluster 3 has high values (> 0.75) for the concepts *instance, class, method, parameter, object, inheritance*, and *constructor*.

As described above, the classification of cluster 2 is more difficult than clusters 1 and 3. This can be justified. For most concepts the relative frequencies of the appearance of a concept in the concept maps are located between the values of the other two clusters. Nevertheless, there are exchanges in the order for the concepts *state, association, attribute, assignment*, and *object*. But, only the two concepts *attribute* and *object* have a significant difference. Thus, in general, clustering of the externalization represents the self-assessment in the correct way.

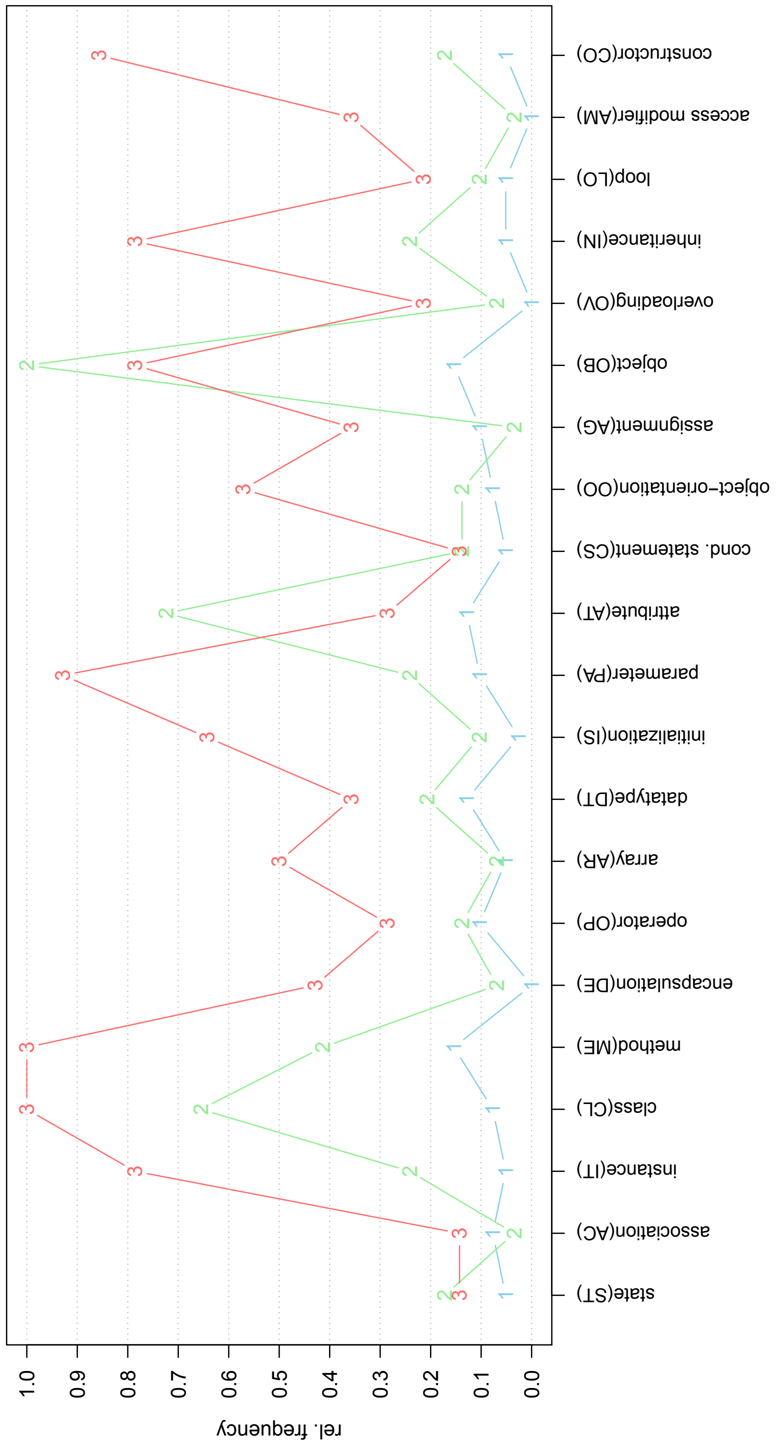


Figure 7.17: Relative frequency of included concepts in the pre-test for each cluster (**c1** to **c3**)

7.5.2.2 Posterior Programming Knowledge

In contrast to the previous test, the grouped concept map resulting from the post-test contains more concepts that are associated in several ways. Again, the concept *inheritance* is missing. According to the previous test, the most edges associate *object, class*, and *attribute*. The most associated concept is *object* followed by *class*. An overview of the combined concept map is presented in Figure 7.18. Here, all associations among the concepts that are not singular are shown. The thickness of the lines represents the number of edges between two concepts in relation to the number of concept maps. Thus, the thicker the line the more concept maps contain an edge between the given concepts.

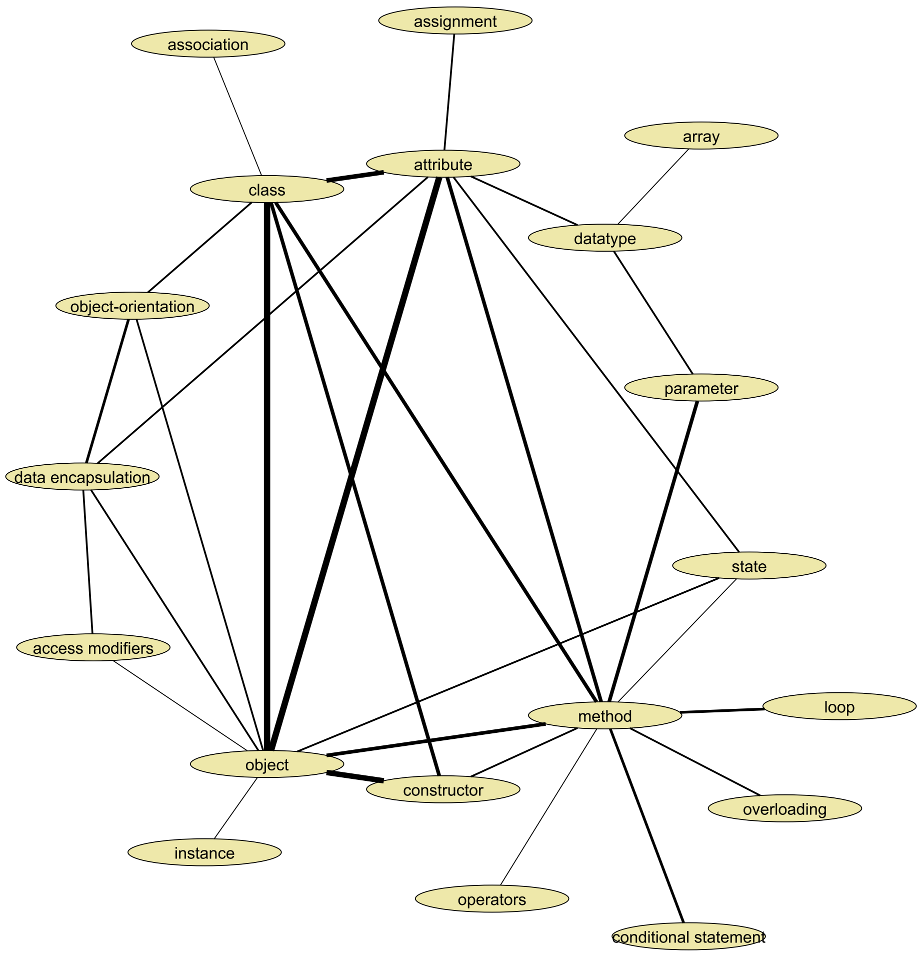


Figure 7.18: Grouped concept map of the 2-rated association of the novice programmers in the post-test – the thicker the line, the more connections have been found between the concepts

The comparison of Figure 7.15 and Figure 7.18 shows that there is an increase in the represented knowledge. An interesting difference is that the edges between *object* and *data type* are not present anymore. The development of the novice programmers' knowledge is investigated in more detail in the next subsection.

After illustrating the novice programmers' expressed knowledge in the post-test, the comparison of this knowledge to the presented learning materials is of interest. Here, two graphical representations are compared. In Figure 7.4, a concept specification map (CSM) of the worksheets is shown. As the concept maps have, in general, no direction, the connections between the concepts in the CSM and the grouped concept map presented in Figure 7.18 are compared. All the connections except *method* and *encapsulation* can be found in the post-test of the novice programmers. The numerous connections of the concept *object* in the CSM are represented in the concept maps as well. Furthermore, the CSM shows strong connections among the object-oriented concepts *object, class*, and *attribute*. Again, this is found in the knowledge representation. Thus, the concepts of the learning materials and their interdependencies can be found in the concept maps of the participants without any previous knowledge.

Another interesting fact is the difference in knowledge after the course between the novice programmers and the other groups. The average scoring of the novice programmers are compared with the other participants in the post-test. Obviously, the two lines in Figure 7.19 are quite similar. The goal of the pre-courses – which is to make participants of a programming introductory course more homogeneous with respect to their programming knowledge – is achieved. More precisely, there are only a few concepts with a significant difference in the representation of the knowledge on programming concepts. The only significant differences are in the concepts where no knowledge gain between the pre- and post-tests of the novice programmers took place (Figure 7.20 in the next subsection). The concepts with a p-value (in parentheses) lower than 0.01 by a two-sided Welch t-test include *instance* (0.002), *initialization* (0.003), and *inheritance* (0.0006).

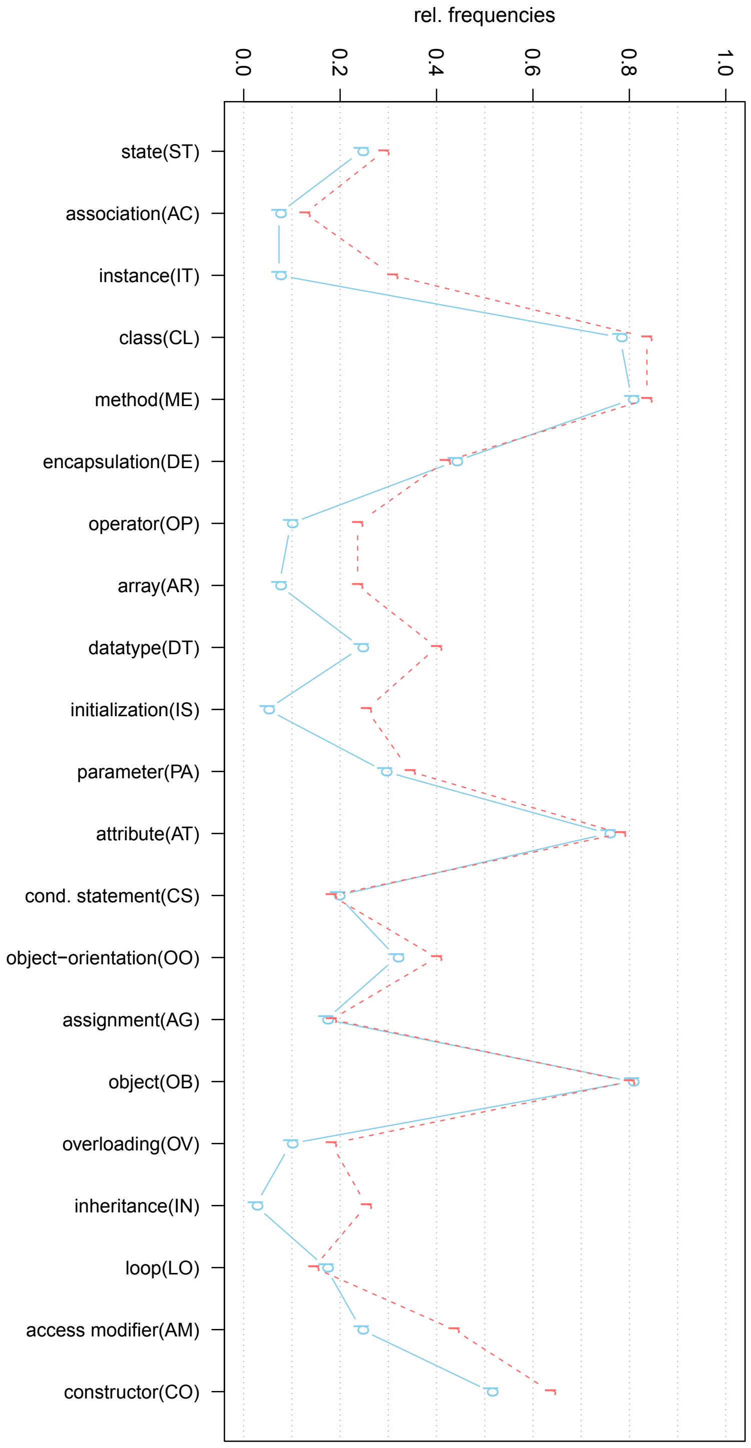


Figure 7.19: Mean values for each concept in the post-test separated by the novice **(p)**rogrammers and the **(r)**est

7.5.2.3 Knowledge Development

As the theory underlying concept maps (see Section 4.1) does not allow conclusions that are related to missing associations or for an individual, the mean values are calculated for each concept over all participants without any previous knowledge and with no background in computer science education in school. The mean values can be seen in Figure 7.20 where the b-line shows the values of the pre-test and the e-line those of the post-test. In this case, the mean values express how many novice programmers have appropriate associations of programming concepts in mind.

For a further investigation of the knowledge on specific concepts, the list of concepts is reduced. For this reason, only those concepts that have a significant development between the pre- and the post-test are included. For this purpose a Welch t-test is applied to the values. The resulting differences in the mean values and the corresponding p-values) are shown in Table 7.12. To gather the concepts that have a significant difference, only those with a p-value less than 0.01(**) are considered. These concepts include *class, method, data encapsulation, attribute, object, access modifier*, and *constructor*.

Concept	Difference		Concept	Difference	
	$\mu_1 - \mu_2$	p		$\mu_1 - \mu_2$	p
ST	0.17	0.07	AT	0.4**	1.54e-03
AC	0.03	0.65	CS	0.13	0.09
IT	0	1	OO	0.23*	0.02
CL	0.6**	3.58e-07	AG	0.07	0.46
ME	0.63**	3.39e-08	OB	0.47**	1.27e-04
DE	0.4**	1.35e-04	OV	0.13*	0.04
OP	0.03	0.65	IN	-0.03	0.56
AR	0.07	0.31	LO	0.2*	0.01
DT	0.13	0.17	AM	0.27**	2.93e-03
IS	0	1	CO	0.43**	9.05e-05
PA	0.23*	0.01			

Table 7.12: Differences between the mean values of the concepts expressed in the pre- and the post-test by the novice programmers and the corresponding p-values ($* < 0.05, ** < 0.01$) of the Welch two sample t-test on $\mu_1 = \mu_2$

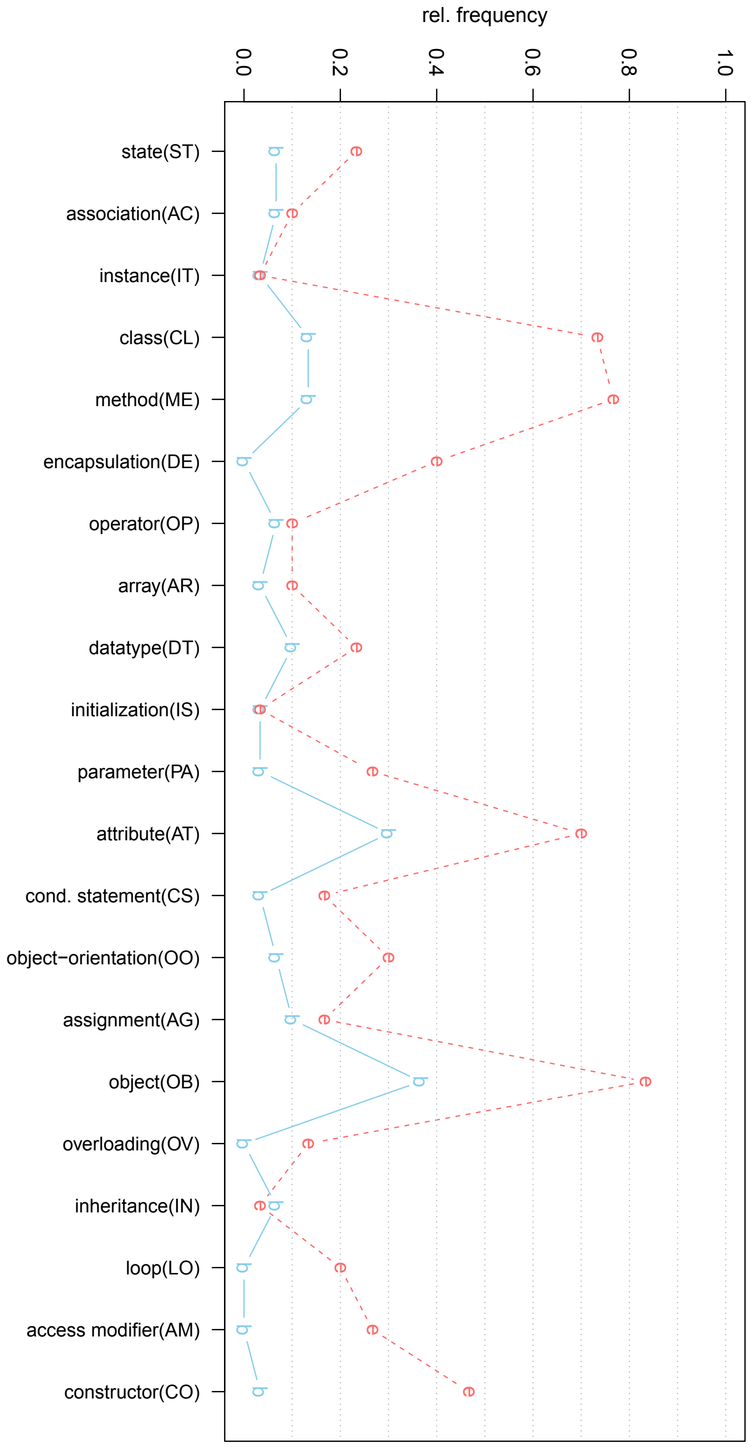


Figure 7.20: Mean values for each concept in the test at the **(b)**eginning and at the **(e)**nd of the course concentrating on the novice programmers

First, the concepts with the biggest knowledge gain are those that are closely connected to object orientation. Seven of the 12 concepts related to object orientation have a significant difference in the mean values, while none of the mixed or procedural concepts are significantly different. Besides contributing to the first programming experiences, the course has the goal of providing first insights into the object-oriented programming paradigm. Obviously, this goal is achieved as there is a major knowledge change (according to the theory of conceptual change (Section 3.5)) in the population of the novice programmers. Interestingly, the knowledge about *conditional statement* and *loop* did not increase much, although these concepts had to be applied a lot during the programming task.

In addition to the development of knowledge, a closer inspection on the differences between the two tests can be done. In Figure 7.21 there are bars that show the differences between the two tests. The value of each concept and participant of the pre-test is subtracted from the corresponding value in the post-test. An increase in knowledge about a concept between the pre- and the post-tests leads to the number of 1. A concept is 0-rated if there is no change, and rated with -1 if there is a decrease in knowledge. The green bars show the average increase, while the red bars show the mean values of the decrease for each concept.

The main aspect is that there are more edges drawn to new concepts than concepts that are not connected anymore. In general, very few edges are removed. This could be due to different reasons. One reason could be the very low previous knowledge level – there were not that many edges that could be removed. On the other hand, the major gain in knowledge representation lies in the concepts explained in detail on the worksheets. The concepts that remain connected or unconnected are *method, data encapsulation, overloading, loop, access modifier*, and *constructor*. The concepts of the pre-test (line b) in Figure 7.20 nearly all have mean values below 0.1. Only the concept *method* has a higher mean value (0.13).

Looking at the added edges, the differences between the concepts are much clearer. There are concepts where only a few valid edges are added (below 20% of the maps): *association*, *instance*, *operators*, *array*, *initialization*, *conditional statement*, *assignment*,*overloading*, and *inheritance*. These concepts are either from the procedural paradigm or are the advanced object-oriented concepts. Again, this matches the impression of the comparison of the pre- and post-tests. The concepts where valid edges are added in more than 50% are *class* and *method*. If the range is widened to 40%, the concepts *attribute, object*, and *constructor* are included. Thus, the added edges are mainly in relation to the object-oriented paradigm.

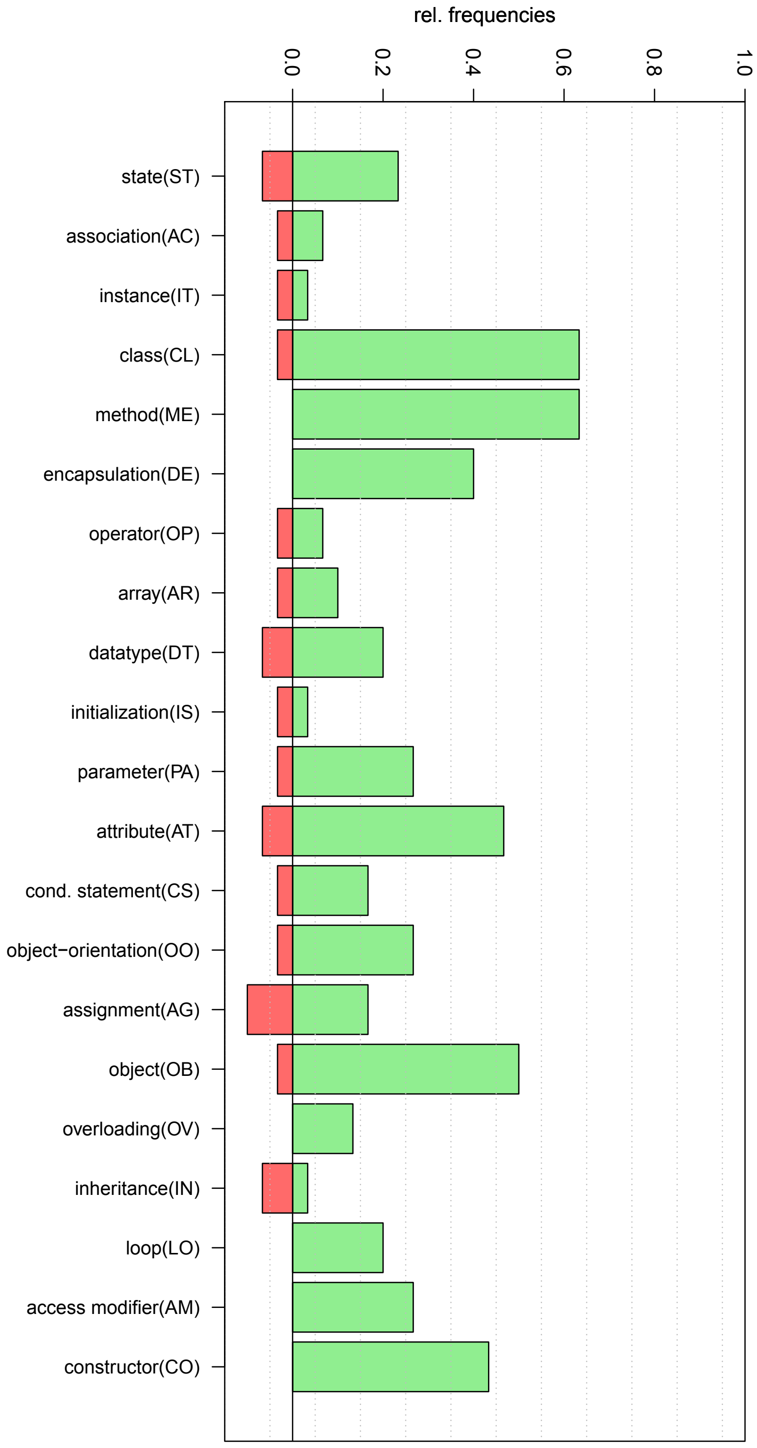


Figure 7.21: Relative frequencies of differences between the pre- and the post-test for each concept drawn by the novice programmers (green: Increase – red: Decrease)

7.5.3 Misconceptions

According to the investigation on the development of knowledge, the examination of the representations of misconception follows the same methodology. Again, the results of the concept maps drawn by the participants of the preprojects (see Section 7.3) are taken for analysis. For the misconceptions, the associations rated with 0 are taken into account. If a participant drew a 0-rated connection between two concepts the map is rated with 1 for this concept; otherwise, it is rated 0. The gained matrix forms the basis for the following analysis.

To compare the mean values from concepts of the pre- and the post-tests, we only take those datasets into account that contain a pre-test as well as a post-test (30 datasets). As only the misconceptions of the novice programmers without any previous knowledge and with no computer science education in school are of interest, we eliminated all datasets that do not fit these criteria. The resulting list contains 156 associations; 58 for the pre-test and 98 for the post-test. The complete list of the 0-rated associations of the novice programmers can be seen in Appendix B.7.2 for the pre-test and Appendix B.7.4 for the post-test.

For examination of the misconceptions, the mean values are calculated for all concepts. These values express how many of the participants have a misconception of a specific concept. Figure 7.22 shows the mean values for the concepts on the y-axis. On the x-axis the concepts are shown with their abbreviations. The line marked with **b** presents the pre-test and the line marked with **e** shows the values of the post-test. A significant increase (p-value smaller than 0.05) based on the pre-test can be found for the following concepts with the p-value in parenthesis: *method (0.01), initialization (0.04), attribute (0.03), access modifier (0.04)*, and *constructor (0.04)*. The same is done for the decreasing concept *assignment (0.01)*.

The associations connecting concepts with misconceptions in more than 20% of the maps in the pre-test are categorized to find common misconceptions. These concepts are: *class, operator, data type, parameter, assignment, object*, and *loop*. Obviously, there are more misconceptions related to the procedural paradigm than those related to the object-oriented paradigm. According to the results found in Section 7.5.2, there is an intuitive understanding of object orientation that the novice programmers can express.

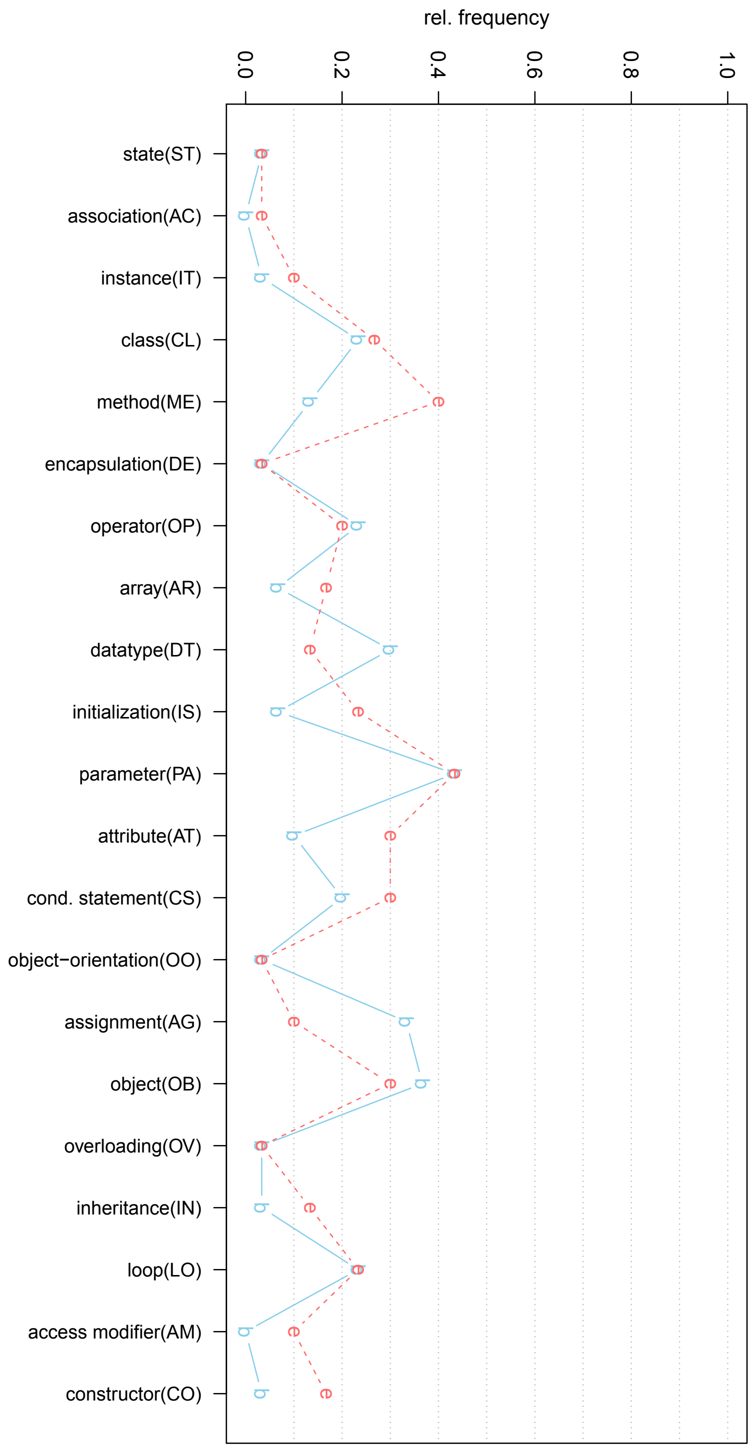


Figure 7.22: Development of the misconceptions in the concept maps drawn by novice programmers at the **(b)**eginning and at the **(e)**nd of the course

Another interesting group of concepts are those that have misconceptions in more than 20% of the maps in the post-test. These concepts are: *class, method, initialization, parameter, attribute, conditional statement, object*, and *loop*. In contrast to the pre-test, misconceptions related to object orientation increased. Basically, there are two different explanations. First, the novice programmers get in touch with the object-oriented concepts for the first time during the course. Before the course their knowledge of object orientation is only very basic and intuitive. After the course the new concepts related to *object* and *class* are not always associated in a proper way. Second, growth of knowledge occurs with an increase in misconceptions, as described below.

Although the worksheets concentrate on the object-oriented concepts, all the concepts that have a significant increase in misconceptions are strongly related to object orientation; whereas the only significant decrease in misconceptions is for a procedural concept.

A closer look on the 0-rated associations to *class* and *object* show that in the pre-test there are five misconceptions (number of occurrences in parentheses): parameter and attribute are mixed up (5), assignment and initialization are mixed up (3), object is an element of class (2), operator and method are mixed up (1), and object and class are mixed up (2).

In contrast, in the post-test there are four misconceptions (again with the number of occurrences in parentheses): object is element of class (8), object and class are mixed up (4), attribute and attribute value are mixed up (2), and parameter and attribute are mixed up (3).

Regarding these two lists a development of the misconceptions can be found. Fundamentally, there is an increase in the misconception that an object is contained in a class. Furthermore, the concepts of *object* and *class* are mixed up. Instead, the mixture of parameter and attribute decreases.

Besides *class* and *object*, the most misleading concept for students who are new to programming seems to be *parameter* followed by *attribute* and *method*. Regarding this group, mixing of the concepts *attribute* and *parameter* is the misconception that is expressed the most in the investigated maps. This can be observed in both the pre-test and the post-test.

Figures 7.23 and 7.24 show the comparison of the concepts connected with 2-rated edges and the concepts with 0-rated edges for the pre- or post-tests.

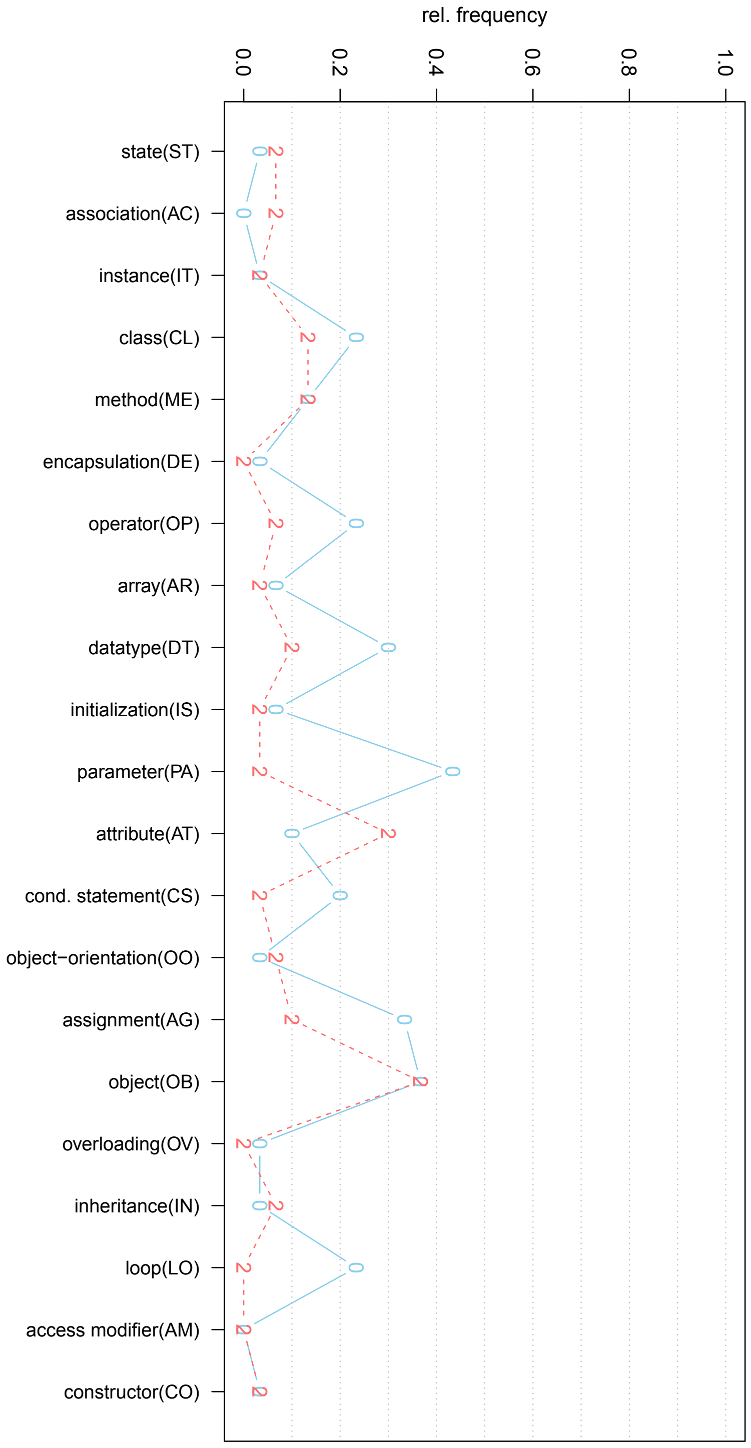


Figure 7.23: Comparison of the concepts connected with **2**-rated edges and concepts with **0**-rated edges in the pre-test

The differences between *knowledge* and *misconception* in the pre-test are of a wider range (see Figure 7.23). With a significance level of 0.05 there are six concepts with a significantly lower level of knowledge than the level of misconceptions: *operators* (0.04), *data type* (0.03), *parameter* (0.01e-02), *object orientation* (0.02), *object* (0.01), and *loop* (0.03e-01). It is noteworthy that for the concept *attribute* the significant difference is in the other direction. Here, there are more maps with a "correct" (2-rated) connection than maps with a "wrong" (0-rated) connection. There is obviously an intuitive understanding of the concept. A closer look at the associations related to *attribute* show that almost all associations are connections to *object* with the normalized label **has**. On the other hand, there is the misconception of mixing up *attribute* and *parameter* in only a few maps.

Generally, in the post-test the distribution of misconceptions follows the distribution of knowledge (see Figure 7.24). So, whenever students know more about certain concepts, they also make more mistakes in that area. This may partially be caused by prominent concepts in the course material that students feel obligated to include in their concept map, but the trend is rather obvious. Interestingly, the concepts with the biggest differences are all related to the object-oriented paradigm. In particular, they all have fewer representations of misconceptions than of knowledge (*class, method, data encapsulation, attribute, object orientation, object*, and *constructor*). For the other concepts, the differences are either small (for example, *assignment, data type*, or *state*) or the concepts have even more representations of misconceptions than of knowledge (for example, *conditional statement, loop*, or *operator*). As the materials of the course give an introduction into the object-oriented paradigm, the increase in knowledge is not surprising. Nevertheless, the differences between the concepts that have a theoretical basis on the worksheets are represented with more knowledge in the concept maps.

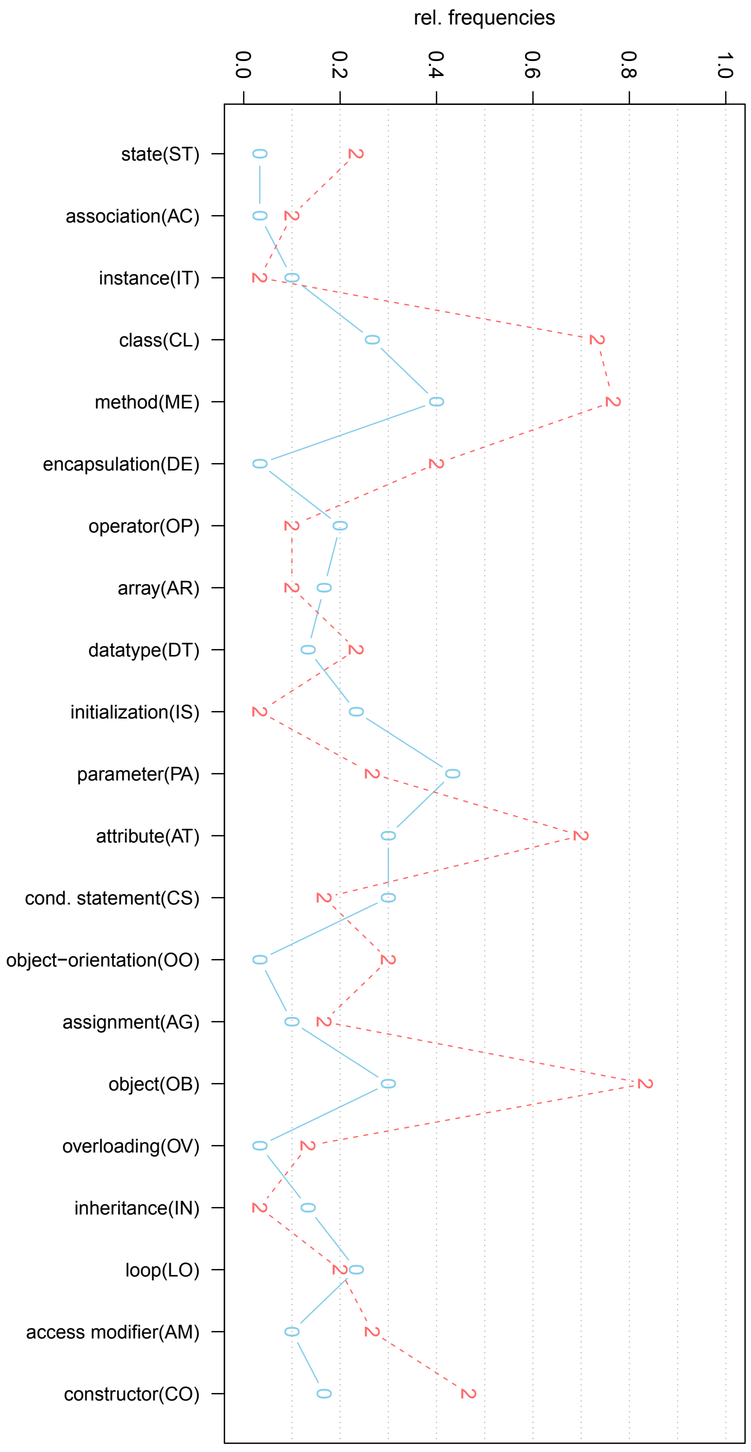


Figure 7.24: Comparison of the concepts connected with **2**-rated edges and concepts with **0**-rated edges in the post-test

7.5.4 Difference Between Knowing and Doing

The maps provide an externalization of the students' knowledge related to the topics of the course, whereas the code items provide a measurement of the students' programming abilities. The main focus of this subsection, is with finding a connection between students' knowledge and the program code they produced, as well as investigating what influence any prior knowledge has. For combining the results of the knowledge analysis with the analysis of the program code, a measure indicating whether or not a student "knows" something about a concept is needed. As described in Section 7.4, the following basic measure was derived for this. If the concept map of a student, for a given concept owns an incident edge with a score of 2, it is assumed that the student "knows" something "correct" about the concept and the concept is scored with 1; otherwise with 0.

This is clearly a very basic measure and one might expect that the percentage of maps fulfilling this criterion for a given concept is very large. However, as the concept maps are rather small and sparse, it works well in practice. Using this measure, two characteristic vectors for each concept map is achieved; for each concept a 1/0 score indicates whether or not the map fulfills the criterion for that concept. Thus, averaging the components of those vectors for each concept over all maps gives an estimator for the probability that the students know something about that concept or that they have a misconception about it.

By applying the scoring scheme to all concepts (see Section 7.4) on all code items of the concepts used in the concept maps, it is possible to compare the program code that indicates the application of the concepts to the concepts in the maps; this indicates how well a concept is understood by the students. The results of this comparison are shown in the Figures 7.25 and 7.26. For the post-test we plot the mean values of how well the students understood a concept (**k**) in contrast to the score obtained from their implementation of this concept (**a**).

The results in Figure 7.25 for students without prior experience fall into three categories:

(1) The two values are high and close together. This indicates concepts that, on average, are understood well and implemented well. The core concepts of object-oriented programming (*attribute, method*, and *object*) fall into this category.

(2) The two values are low and close together. This is the opposite of category 1. *Association*, *overloading*, and *inheritance* belong to this category, which are the more advanced object-oriented programming-related concepts.

(3) There is a gap between the two values. This includes all the concepts that are related to procedural programming, as well as the more "technical" concepts such as *arrays* and *access modifiers*.

Students with prior knowledge (see Figure 7.26) show almost exactly the same distribution along the concepts; however, the gap is not as pronounced in all of the cases.

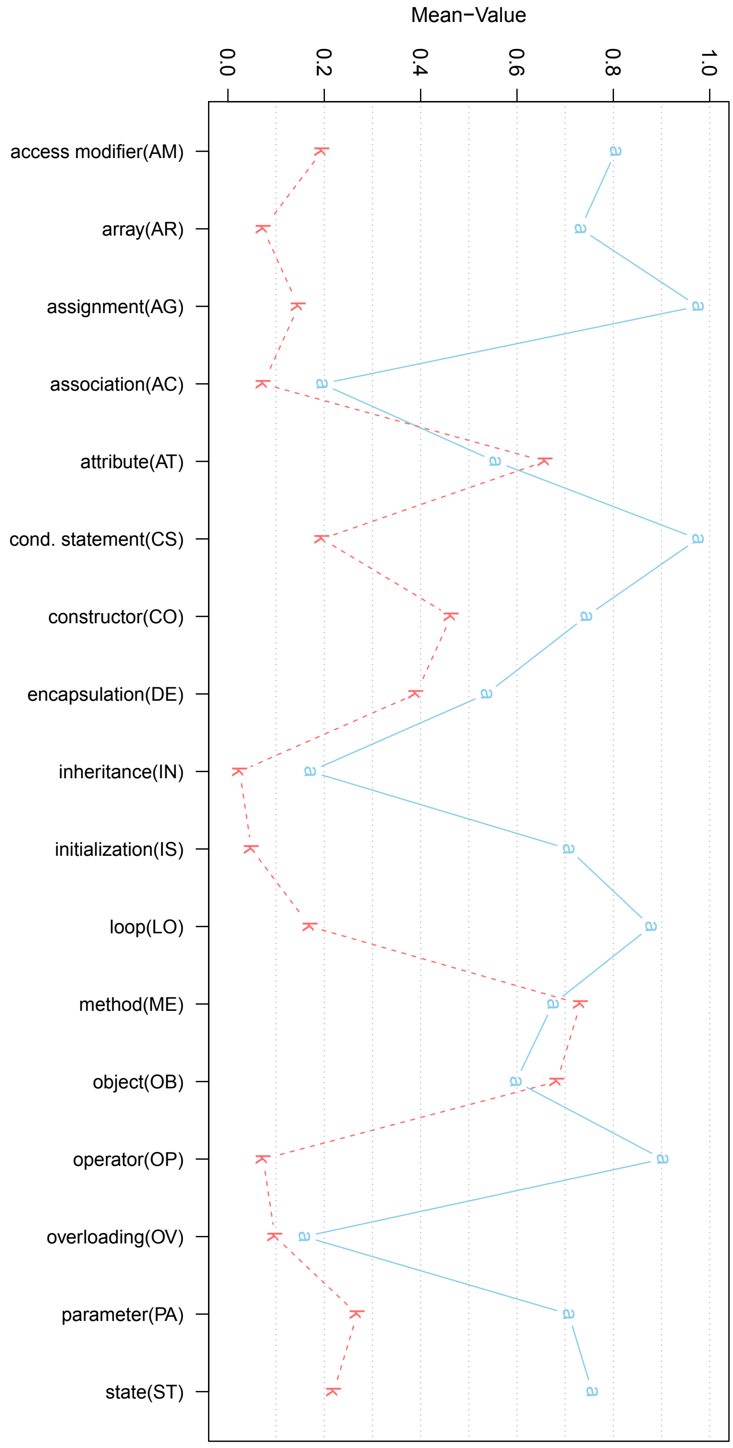


Figure 7.25: Comparison of the code items and the concept maps in the post-test of the students without prior knowledge. Line **a** shows the mean values of the concepts in our code analysis. Line **k** shows the mean values of the same concepts in the concept maps of the post-test.

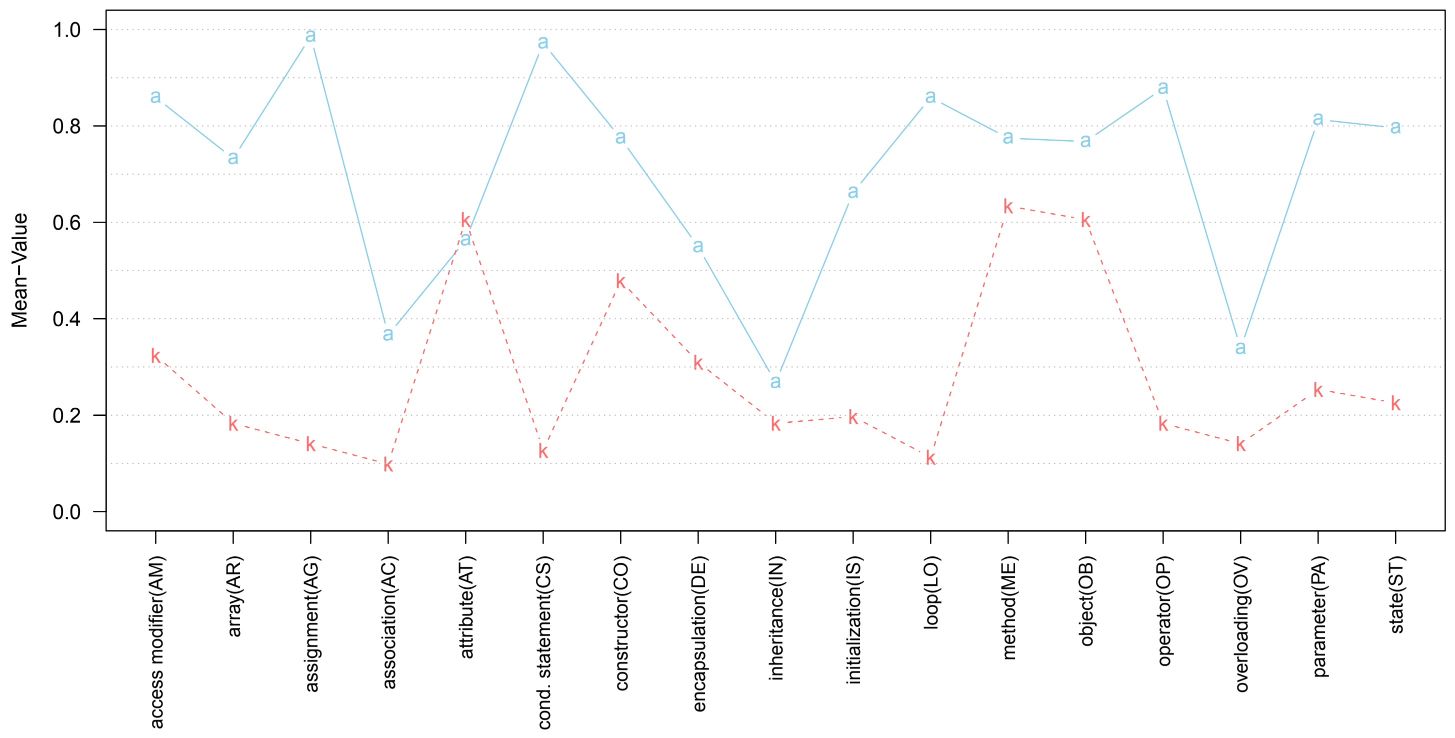


Figure 7.26: Comparison of the code-items and the concept-maps in the post-test of the students with prior knowledge. Line **a** shows the mean-values of the concepts in our code analysis. Line **k** shows the mean-values of the same concepts in the concept maps of the post-test.

It is interesting to note that there are concepts of object orientation in both of the first groups. This clearly shows that several object-oriented concepts (like *inheritance*) are seemingly harder to grasp than the rest. The students actually are able to understand and apply the basic concepts of object orientation after the course. However, this has to be seen in relation to the actual programming project that the students were asked to do: While an experienced programmer would certainly use *inheritance* for the task at hand, it can also be solved using very little or no *inheritance*. This is reflected in the fact that the students with prior knowledge were also not good at implementing *inheritance*, but knew “something” about it.

Next, we take a closer look at the serious difference between the representation of knowledge and its implementation in the code of the third group of concepts such as *conditional statement* and *loop*. These concepts do not fit directly into the paradigm of object orientation. Both concepts are implemented quite often in the code, but are not often correctly integrated in the concept maps. As mentioned above, the majority of the concepts have their origin in the object-oriented paradigm. Thus, it is quite difficult to find the correct association for them, which may explain the low value in the concept maps. However, it may just as well show that “understanding” concepts is not a trivial task and that it takes considerably more time than learning to apply them. This is supported by the fact that there are relatively more misconceptions for those concepts in the post-test (see Section 7.5.3).

This indicates the inherent difficulty in learning object-oriented programming. There are several groups of concepts that are all needed for the creation of an object-oriented program, but those groups of concepts are showing radically different results even for those students who had prior programming experience. There is a group of concepts that is understood and applied well, a group that is only applied well, and a group that is neither understood nor applied well.

Finally, there is an interesting observation on two very similar concepts; *state*, on the one hand, and *attribute* on the other. As the state of an object is defined by its attributes, nearly identical values are expected. This basically holds true for the implementation. But, looking at the results of the knowledge post-test, a big difference in the values can be seen. Here we clearly have the case of a concept that is being implemented by the students, but not understood. It is not surprising that the students do not fully grasp the concept of state transitions of objects after this particular course.

7.5.5 Common Questions

During the 2010 term the tutors were asked to record the questions they were asked by the participants. The questions are categorized by their content, their relation to programming, and the type of answer the tutor gave. To analyze the common questions that the participants asked the tutors during the preprojects, the questions gathered using the method described in Section 7.4 are first grouped by the concepts provided in Table 7.4, as well as two additional categories for syntax errors (*syntax*) and other relations such as graphical user interfaces (*other*).

In addition to categorizing the questions by concepts, the questions are grouped by the type of answer that was given by the tutors. Because of the binary coding of the answers, 32 categories evolved for the five possible choices since all permutations are valid. The tutors could have given answers that belong to multiple types.

After the general categorization of the questions described above, the questions from the novice programmers without any previous knowledge or computer science education in school are investigated. From the 182 questions we gathered during the course, 89 questions are from novice programmers. To analyze the questions, the representatives of the categories were counted. Table 7.13 shows the relative frequencies for all participants and for the novice programmers by category.

Abbr.	Category	Rel. frequency		Abbr.	Category	Rel. frequency	
		All	Novices			All	Novices
AM	access modifier	0.022	0.011	IT	instance	0	0
AR	array	0.104	0.045	LO	loop statement	0.066	0.045
AG	assignment	0.005	0	ME	method	0.088	0.124
AC	association	0.049	0.045	OB	object	0.093	0.124
AT	attribute	0.033	0.034	OO	object orientation	0	0
CL	class	0.044	0.034	OP	operators	0.033	0.045
CS	conditional statement	0.049	0.079	OV	overloading	0	0
CO	constructor	0.049	0.056	PA	parameter	0.027	0.034
DE	data encapsulation	0	0	ST	state	0	0
DT	data type	0.071	0.045	SY	syntax	0.049	0.056
IN	inheritance	0.005	0	OT	other	0.164	0.157
IS	initialization	0.049	0.067				

Table 7.13: Relative frequencies of the questions by category for all participants and for the novices group only

While the questions in general mainly cover the concepts of *method, object*, and *array*, the novice programmers did not ask many questions on the concept *array*. The category with the highest relative frequencies in both groups is *other*.

The distribution of the questions of all participants differs only slightly from the distribution of the novice programmers. There is a non-significant difference between the basic concepts of object orientation (such as *method* or *object* – which are contained in more questions from the novices) and more complex concepts (such as *array* – which are contained in more questions from all the participants).

Comparing the misconceptions from Section 7.5.3, a correlation is shown between the relative number of questions posted to the tutor and the relative frequency of misconceptions in the concept maps. For all participants the correlation is 0.58 and for the novices alone it is 0.64. This could be an indicator that the tutors' answers were partly confusing. The correlation is weaker (0.35) between the questions and the expressed knowledge (see Section 7.5.2) in the concept maps.

The report form of the questions the participants posed to the tutors give a clue of what was happening during the course. According to the fact that the answer could be given using different types of answers, the relative frequency add up to more than 1. As Table 7.14 shows, for all participants two thirds of the questions were answered by a hint, while almost all of the remaining questions were answered by simply showing the appropriate code. On the other hand, the novices' questions were answered in equal parts by showing the appropriate code or presenting a hint. The simplest form of help by pointing to the reference or the IDE's integrate help is barely used. Additionally, rarely is help provided by recommending the proper chapter in the online book. The Java reference, which was only used in the groups with previous knowledge, was recommended a few times.

Answer type	Rel. frequencies	
	All	Novices
hint	0.615	0.584
code	0.423	0.517
rec. book	0.016	0.011
rec. reference	0.06	0.022
rec. IDE-doc	0.11	0

Table 7.14: Relative frequencies of the questions by answer type for all participants and the novice group only

7.6 Evaluation of Program Code using Psychometric Models

As mentioned in Section 4.3, the item response theory is a common methodology that is applied in human sciences for measurements. Usually, a test with several evaluated items is conducted on a specific population. Afterwards, the items are checked and parameters are calculated for the items and for the individuals of the test. As programming, in general, is a complex task and refers to several cognitive (and, therefore, latent) processes, the item response theory is applicable. In this section a new view on the evaluation of program codes is presented. Generally, small programming tasks on a specific concept or code element are provided to the participants of a programming or coding assessment. But, if the coding ability – in the sense of a competency – must be evaluated, the tasks have to be more complex. The interdependencies of different structure elements within a program code are of interest.

The code items defined in Section 7.4 can be seen as a partition of the observable abilities needed for programming. During the preprojects the students were asked to implement a small project on the basis of an assignment that did not include explicit questions on programming. Nevertheless, the resulting program code contains the responses to these questions. This is the reason why the code items can be assumed to be the questions posed to the participants, even though they were not. The items are adhered to, although they do not cover all concepts that would be assigned to the programming ability, because the items are based on the concepts extracted from the worksheets that are, again, the basis for the programming tasks.

All items are rated with either "yes" (1) or "no" (0), which leads to a dichotomous vector for each participant. The resulting matrix is the basis for a latent trait model. Generally, there are two major goals related to latent trait analysis. The first goal that has to be achieved is the finding of a homogeneous itemset. So, either the number of latent constructs is examined, or the itemset is reduced to fit a previous defined number of latent constructs. In the latter case, this is usually only one construct if the number of constructs of the complete itemset is unknown. The second goal is then to examine a valid test framework that fits the assumed latent trait model. In this investigation, for example, the item list is fitted to a Rasch model. The model is chosen because it is the simplest model, if the preconditions are fulfilled. As mentioned below in Section 7.7, the programming ability of a person is not uni-dimensional. Nevertheless, the investigation is concentrated on the code items that were defined for the novice programmers' investigation.

> "Whether the questions, for example, use a multiple-choice, open-ended, true-false, forced-choice, or filled-in-the-blank response format is irrelevant. All that matters is that the data analyzed are dichotomous and that the assumptions are tenable. The appropriateness of the model to the data is a fit issue." (Ayala 2009, p. 22)

After defining the code items and examining the program code, there are 350 datasets. As some of the model-fitting tests (see Section 4.3.2) need separation criteria, only the datasets with a survey are included in the investigation. This results in 321 final datasets.

In a first step, all items that are related to the same structural element as others are eliminated. For example, both `ST1` and `AT1` need a variable declaration in the code to be 1-rated. More precisely, this affects the items `AG1, ST1, ST2, ST3`, and `OB1`. So, for each structural element there is unique set of items.

As all the tests that rely on dividing the population or item set into two parts need differing items in both parts of the population, the trivial items are removed in advance. An item is said to be trivial if either almost all or almost none of the participants are rated with (1). For that reason a limitation level of 0.01 is defined for this study. In particular, this affects only the item `OP1`, which covers the use of an assignment operator.

Due to the large number of items at the beginning, the calculation of an exact test on homogeneity is not possible (for example, the Martin-Löf test described in Section 4.3.2.1 or test statistics provided by Bartholomew (2008) (see Section 4.4)). Because of that, the quasi-exact tests of Section 4.3.3 are applied. First, the item set is reduced to those items that are homogeneous. More precisely, the resulting items have to be uni-dimensional. According to Section 4.3.3, the modified test statistic **T1m** is applied on the data.

Starting with the 33 items after the exclusion process, all items that violate the homogeneity criterion are eliminated. Therefore, the **T1m** statistic is iteratively applied to the item set. This is necessary as the nonparametric tests operate on a set of simulated matrices that are dependent on the result matrix (see Section 4.3.3). After the elimination of several items, the result matrix changed.

In the first run the items `DE1, OP2, IS1, AR5, CS2, CO1, AR4, CS3, OV1, AR1, PA2, PA3, ME2`, and `IN1` violate the homogeneity presumption. The selection criteria for which items are eliminated is the frequency of the violating items. Thus, `DE1` has the most violations, while `IN1` has the least. In general, all pairs of violating items are listed and the items are removed from that list one by one until the list is empty; then, no violations are left. Afterwards, a new set of simulated matrices is calculated based on the new item set. The second run results in elimination of `AR3, AR2, LO1, OB3, ME1, AC2, AT1, AT3, ME3, PA1`, and `AC1`. Once again, the items are ordered by their number of violations. After a third and fourth elimination run, the items `AM1` and `CS1` are removed from the item set. In the end the remaining items are homogeneous.

Now, as the item set has been reduced to six items, the exact tests of Section 4.3.2 can be applied for justifying the nonparametric tests. The Martin-Löf test (Section 4.3.2.1) is conducted on the resulting item set. Here, the p-value is 0.66. Thus, the items are assumed to be homogeneous. After that, the two test statistics presented in Section 4.4 are calculated. For the general goodness-of-fit test statistic G^2, a value of 62.1 is the result. Additionally, the χ^2 test statistic X^2 results in a value of 139.4. Both are not significant for 13 degrees of freedom to a level of 0.05 in the χ^2-distribution. Again, the H_0-hypothesis is rejected and the items are assumed to be homogeneous.

Following the test on homogeneity, the test statistic **T1** is applied to find the items that violate local stochastic independence in the way of being too similar in their results. Actually, only the item `OB2` is dependent on another item and is, therefore, eliminated from the results set. Last, the learning aspect of the local stochastic independence is tested with the test statistic **T1l**. Here, no item violates the presumption.

The resulting item set with the items `IN2, CO2, OP3, AT2`, and `OV2` is valid with regard to the presumption of the Rasch model. After validating the presumptions of the Rasch model, fitting of the data and a valid model are calculated for the given data.

The likelihood-ratio test (see Section 4.3.2.3) has a p-value of 0.50 for the first pre-knowledge splitting criterion (pre-knowledge level 1 vs. pre-knowledge levels 2&3).

The test is not significant and, because of that, the model is assumed to be valid. In other words, the data fits the model. A look at the Wald test (see Section 4.3.2.4) for the items also shows no significant model violations (p<0.05). For both tests we have to assume a vector that splits the population into two parts. As mentioned in Section 4.3.2, theoretically all possible splittings have to be tested. In reality, this is not applicable. For this reason students' self-assessments of the previous knowledge (pk) concerning programming is chosen as a separator and, additionally, gender is chosen to find differences. Regarding the students' previous knowledge, to get two groups two levels are put together and compared with a third level. Table 7.15 shows all the p-values for the different splitting criteria. Gender as a separation criterion is not applicable as only 19% of the participants were female. As mentioned by Koller and Hatzinger (2013, p. 99), the two groups should be almost of equal size.

Item	Previous knowledge			Gender
	Level 1 vs. 2&3	Level 2 vs. 1&3	Level 3 vs. 1&2	
`IN2`	0.72	0.84	0.9	n.a
`CO2`	0.55	0.91	0.29	n.a
`OP3`	0.3	0.73	0.12	n.a
`AT2`	0.56	0.78	0.62	n.a
`OV2`	0.13	0.35	0.37	n.a

Table 7.15: The p-values of the Wald-test for different splitting criteria

Regarding the gender aspect, a closer look at the different items with separated participant groups show the violations in the model fit. For that reason, all items are split by gender. Afterwards for each group the items are tested if they are trivial. Due to the small group size of the female participants, a ratio of 0.01 is too small as it is less than one person. Because of that, the limit is set to 0.02, which means that at least one person has to have a different answer than the others. The problem occurs with the item `OP3`, where there is less than 0.02 different answers for the female group. Additionally, the item `IN2` has a value of only 0.03 for the female participants. Nevertheless, this is not critical for the test. All other items have a distribution between the answers of 0.3 and 0.7 (`CO2` and `OV2`) and 0.5 for both groups (`AT2`).

The graphical model check plot (see Section 4.3.2.2) shows the item parameters with separation of the two groups (see Figure 7.27). The separation criterion underlying this plot is the splitting of the previous knowledge between those without any and those with previous experience. Here, the two groups are almost of equal size with 180 to 141 participants in the corresponding groups. Again, the plot shows the difference between the easiest and the most difficult, on the one hand, and the other items, on the other hand. The confidence areas of those two items are much bigger than those of the other items. This is the result of triviality of the items (many 1-ratings and almost no 0-ratings or vice versa).

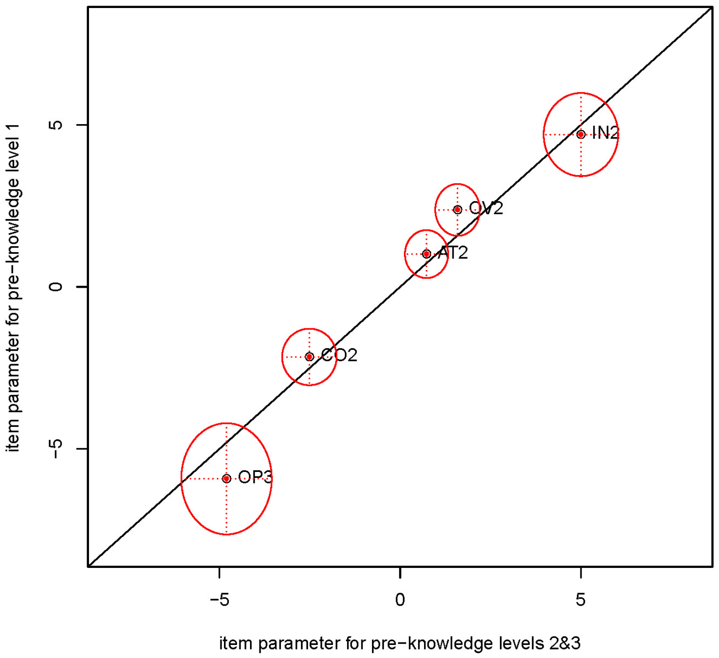


Figure 7.27: Graphical model check for all items in the Rasch model

As described in Section 4.3.1, the Rasch model is a one-parametric test model where the items only differ in their level of difficulty. To show that another model with more estimated parameters does not fit better, a two-parametric test model is calculated and both models compared. All model comparison coefficients such as Akaike information criterion (AIC) and Bayesian information criterion (BIC) (Burnham and Anderson 2002) have almost the same values. The two-parametric model provides no advantage by introducing an additional parameter. Nevertheless, Figure 7.28 shows that there are differences in the discrimination of the ICCs and that there are even two items that seem to intersect. `AT2` and `OV2` are very narrow for a high probability. The person parameters differ slightly between the two models, but the order of the items stays the same.

After fitting the dataset to a valid Rasch model, the next paragraphs present the results of the model. In Figure 7.29 the item characteristic curves for all items that are included in the model are shown. According to the definition of the Rasch model, they only differ in their level of difficulty. This is expressed in the figure by a shift on the x-axis, which shows the latent parameter on a scale of -10 to 10. All curves are parallel and only differ in the value of the latent parameter at the probability of 50% for rating a code item with "yes" (1). The probability that an individual with a specific value of the latent parameter has solved a specific item is drawn on the y-axis.

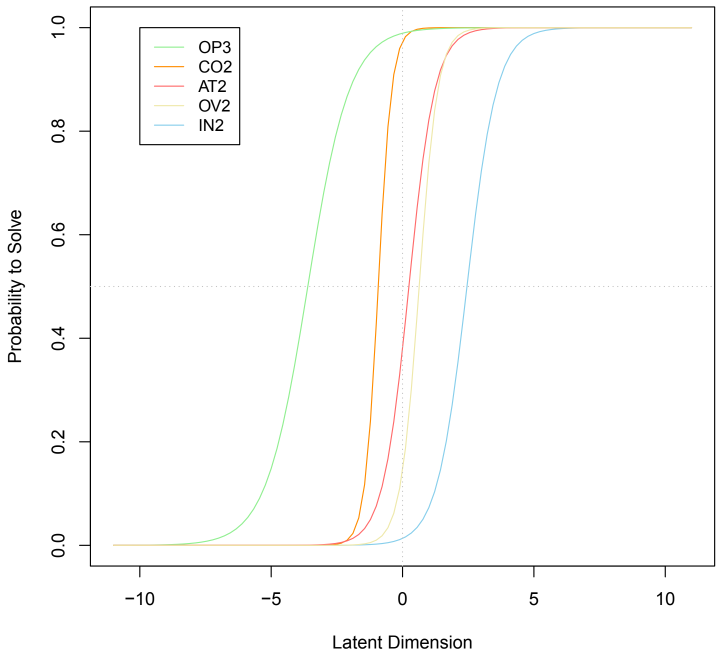


Figure 7.28: Item characteristic curves (ICC) of all items included in the 2PL model

By definition, the item parameters sum up to 0. The items `OP3` and `IN2` have values of -5.4 and 5.0, respectively. The item that is closest to the average of 0 for the investigated population is `AT2` (0.84). The use of attributes of other classes, either direct or by using a method, indicate participants with an average ability in coding, concerning the investigated items. Interestingly, all items except `AT2` and `OV2` have the same distance between each other. In general, Figure 7.29 presents a ranking of the items. The simplest item is `OP3`, which represents the use of arithmetic operators. The underlying concept is simple to code and all projects need calculations. As a result, the position within the items is not surprising. The next concept in the ranking is `CO2`, which indicates the use of a constructor or an initialization of an object. Again, the underlying concept is easy, but the basic object-oriented notions have to be implemented as well. Next, the items `AT2` and `OV2` indicate the use of interrelations between classes. As mentioned above, the first one represents the use of foreign methods and attributes. The second one represents the use of overloaded methods. Regarding the last item `IN2` (use of an own class hierarchy), these two items represent more advanced concepts of object orientation. Thus, the item set contains representatives of simple coding concepts that can be related to the procedural paradigm, as well as representatives of advanced object-oriented notions.

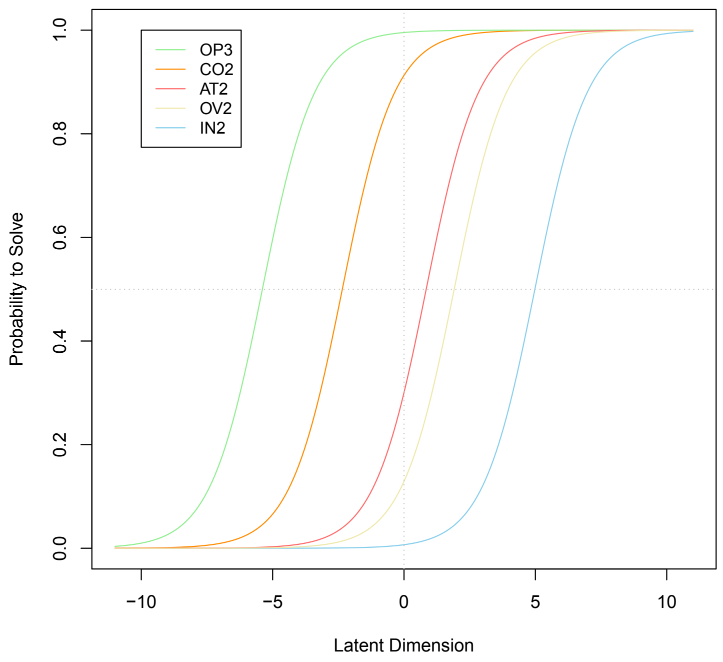


Figure 7.29: Item characteristic curves (ICC) of all items included in the Rasch model

In addition to the plot of the item characteristic curves, a ranking of the items by difficulty can be seen in detail in Figure 7.30. The figure is divided into two parts, sharing the x-axis where the latent parameter is shown on a scale from -5 to 5. The lower part of the figure shows the item parameter for each item. The items are placed in ascending order by the value of their item parameter. The upper part of the figure shows the distribution of the person parameters on the latent dimension. The mean value of the person parameters is 0.11; the median is -0.92.

In general, if the Rasch model is valid, the marginals of the underlying dataset are a sufficient statistic. Because of that, each possible person score is related to a person parameter (see Figure 7.31). For mathematical reasons the parameters for the margins 0 and 5 cannot be estimated, but have to be interpolated. An overview of the person parameters and their distribution can be seen in Table 7.16.

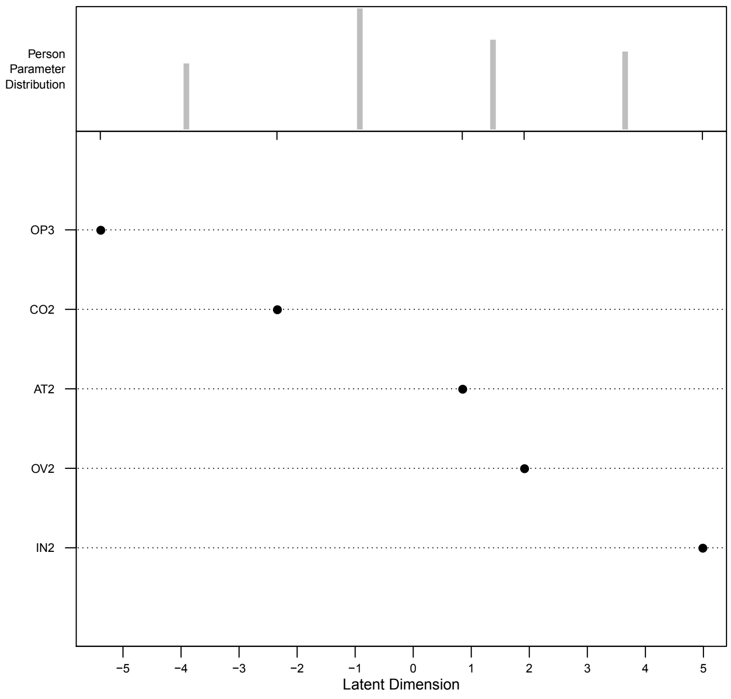


Figure 7.30: Item parameters ordered by difficulty and the distribution of person parameters

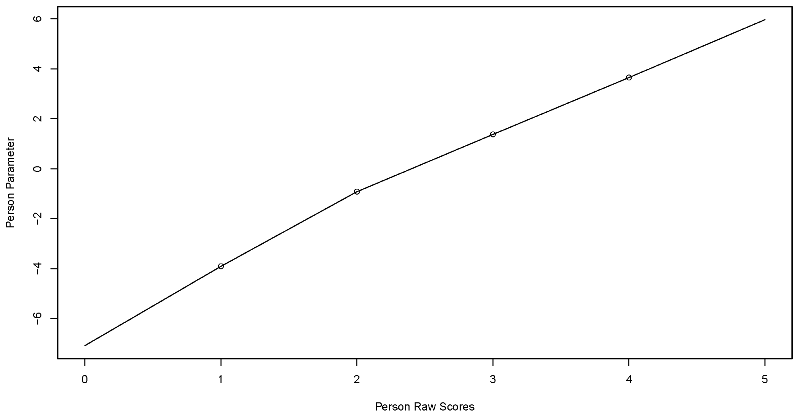


Figure 7.31: Correlation of the person parameters to the raw score

Marginal score	Person parameter	Frequency		
		All	Pre-knowledge level 1	Pre-knowledge level 2&3
0	-7.08	0.01	0.03	0
1	-3.91	0.18	0.28	0.09
2	-0.92	0.33	0.35	0.32
3	1.37	0.24	0.26	0.23
4	3.65	0.21	0.08	0.31
5	5.97	0.03	0.01	0.04

Table 7.16: Frequencies of the values of person parameters using the Rasch model, as well as the marginal scores

Actually, there is a medium correlation (0.42) between the self-assessment of the students' previous knowledge and their person parameters (p-value $\ll 0.01$). Regarding gender of the participants, females (-0.13) have a lower – but not significant – average person parameter than male students (0.26). On the other hand, the self-assessment has a significant difference (p-value $\ll 0.01$) in the person parameters. The students with previous knowledge have a mean value of 1.06, while those without any previous knowledge have a mean value of -0.93.

In addition to students' previous knowledge and their gender, lines of code are another measurement that can be conducted on the program code (see Section 7.4). In particular, the projects differed a lot in their complexity. There were projects with only a few lines of code (min. 6 LOC) and some with more than one thousand lines of code (max. 1330 LOC) containing a GUI and other features . The mean value of project size regarding the lines of code is 212.7 LOC. Figure 7.32 shows the distribution of the project sizes in box plots for all participants, with and without previous knowledge. For all participants, the median is 129 LOC, while the first quartile is 73 LOC and the third is 275 LOC. Furthermore, the projects developed by those with previous knowledge have significantly (p-value $\ll 0.01$) more lines of code. The mean value for those with pre-knowledge is 253.2 LOC versus 160.7 LOC for those without pre-knowledge.

Regarding the person parameters of the Rasch model, there is no correlation (0.07) to the lines of code. For that purpose the lines of code have been grouped together to form four groups at the quartiles of the distribution.

Validation of the Rasch model is quite difficult. In contrast to the formal validation that is described above, validating whether the items really measure the programming ability as the latent dimension is hard to proof. In fact, the items only cover a part of programming ability as some concepts that are not mentioned in the material for the preprojects are missing. Additionally, the facet of problem solving that is a huge part of the programming ability cannot be assessed by a simple structural analysis. There are more threats to the validity of the presented research, which form the content of the next section.

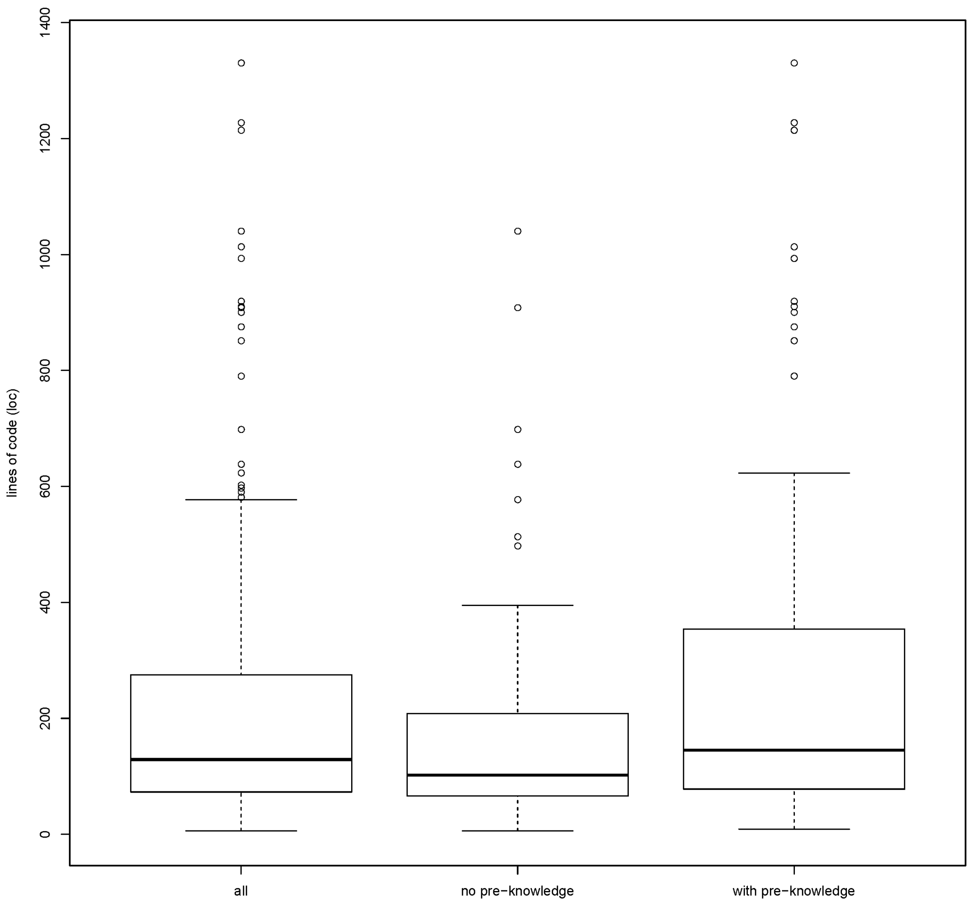


Figure 7.32: Box plots of the lines of code for the different previous-knowledge groups

7.7 Threats to Validity

Brooks (1980) published work on the problems of studying programmer behavior. The aspects mentioned by Brooks are now discussed for the presented investigations. The first issue mentioned includes choosing the participants of the study. On the one hand, the participants should be representative for the population that is being investigated. In the presented investigation this is not important since the results are not extended to generality as the experimental setting is very specific. On the other hand, the participants should be uniform with regard to their characteristics and abilities. This issue is fulfilled in that the participants are all starting with their first semester. Nevertheless, the participants did differ substantially in their previous knowledge and ability. However, as the study is conducted with different groups that are themselves homogeneous, and the fact that differences are found between those groups, there is no problem with the investigated population.

The second issue mentioned by Brooks (1980) is the choice of materials that are distributed during the experimental course. He mentions that the material should have a relation to the real-world. The program code produced during the courses have only a very limited number of lines with regard to real-world projects. However, as the study mainly focuses on novice programmers, this does not have to be taken into account. To address the problem of the large amount of available textbooks (see Section 6.4) with all their advantages and disadvantages, the students are provided with particular material that contains the required information. This possibly had an influence on the results found. But again, this chapter should only show options of evaluating novice programmers' knowledge and abilities without generalizing the results found in the investigated population. On the other hand, if the students had not received particular materials, it would not have been possible to control the input that the students received.

The last issue addressed in Brooks's paper (Brooks 1980) is the selection of an appropriate measure. In the presented results there are three different types of measure. The first one is the program code analysis. Here, the composition of simple code items to concepts is worth a discussion. As no literature shows similar ways of doing a code analysis, the composition had to be performed based on experts. A deductively composed concept list would have been better. A further problem is the code items themselves. Whether or not they really enclose all necessary aspects of programming constructs could not be proven. Nevertheless, the method is the main focus in this thesis. An interesting topic for future research is the validation of a general scoring system for program codes that is based on code items (see Section 8.2).

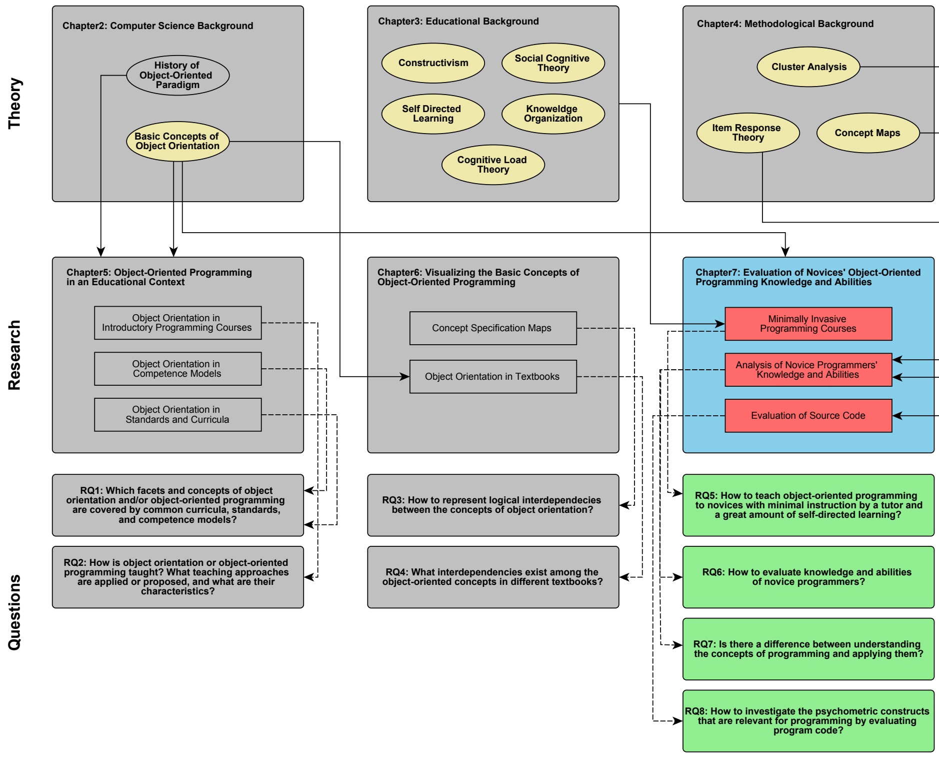
Besides the code items, concept maps are used to investigate the representation of cognitive knowledge of the participants. While concept maps are a proven tool, there are still some open questions about how to apply them in order to get the best possible results from students. The list of concepts have a major impact on the results. When asking students to draw concept maps, the introduction given to them has a major impact on the results. We provided the students with an exemplary concept map (not related to CS) that was a little different from the usual concept maps, because it had rather complex edge labels (almost complete sentences). To a great extent, the students tried to replicate this kind of concept map in their own map. Therefore, it would probably be best to provide the students with several examples that show a wide range of possible concept map applications, so that they are free to choose which way best expresses their knowledge. Additionally, it seems worthwhile to think about limiting the possible edge labels to a predefined set. Doing so would simplify using the edge labels as part of the analysis. As described by Hubwieser and Mühling (2011*c*), it seems like a set of 10-15 edge labels would suffice when taking into account what the most typical labels of the students have been.

The most interesting measure introduced in this thesis is the application of item response theory on the program code. The first and major problem is the one-dimensionality of the latent dimension. In a wider range we can state that all implementations of the concepts together form a part of the programming ability a novice programmer can have. However, looking at the different concepts, there are surely different aspects. There are concepts that are really difficult (such as arrays) and there are concepts that are less difficult, but the programmer has to recognize its benefit for a better programming code. Therefore, there is an obvious second dimension where its influence on the programming ability has to be investigated in future work (see Section 8.2).

Another problem we had when constructing the items after the code had been written and the courses had taken place, was the dependence of some concepts to the assignment that each participant had received. Some assignments forced specific concepts such as *inheritance*, while others did not. Due to the results of the tests we had to exclude some concepts that would have been interesting to investigate in a further way. Therefore, in a future run of an assessment of programming abilities applying item response theory, the assignments have to be chosen in a proper way so that no concept has to be excluded due to the abovementioned reasons.

7.8 Summary

The evaluation of novice programmers' knowledge and abilities is an important topic in computer science education. The investigations of this very special group of participants of programming courses were conducted in a special setting minimizing the influence of instruction on the participants, as well as on the results. The minimally invasive programming courses are a useful setting for investigating freshmen. By minimizing the influence of instruction and increasing the amount of self-directed learning, the novice programmers could be purely investigated (**RQ5**). As the results show, knowledge gain is high even when there is no direct instruction. Another important result from the presented investigations of novice programmers' knowledge and abilities is the gap between knowing and doing in programming. Cluster analysis (Sections 4.2) is a good method for finding homogeneous groups and differences in knowledge of novice programmers. Even if the programmers have the same previous knowledge, there are a lot of differences in the development of the programming ability in general (see Section 7.5.1), as well as differences in knowledge gained on concepts related to programming (see Section 7.5.2) or the misconceptions in that knowledge (see Section 7.5.3). Depending on the direction of the course that builds the setting for the investigation, there is a difference among the paradigms. However, no significant results could be found to support that object-oriented programming is more difficult than the rest. On the basis of the given material, it is not surprising that the concept maps of the post-test tend to have more associations with the concepts related to object orientation. The section on common questions (see Section 7.5.5) presents a methodology to gather the information about what really bothers the novice programmers and where the difficulties for this specific group are. All investigations addressing the knowledge and abilities of freshmen show a variety of methods that can be applied for this purpose and answer research question **RQ6**. According to research question **RQ7**, the differences between knowing and applying concepts related to object-oriented programming (Section 7.5.4) were investigated. It turned out that the novice programmers could apply concepts without being able to integrate them in a concept map (see Section 4.1). Nevertheless, there is one problem with programming at all; the rating of code is very difficult. The application of the psychometric methods of the item response theory on program code (see Section 7.6) provides an capability to rate the code as well as the concepts underlying it. With a valid Rasch model for a set of code items, it is possible to simply count the implemented items in the code. More precisely, the method enables the investigation of the psychometric constructs that are relevant for programming, especially, the coding ability (**RQ8**). In later research automatic testing and rating of code should be possible. Further ideas about this topic that are not part of this thesis are presented in Section 8.2.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

8 Conclusion

8.1 Summary

Chapter 5 shows that object orientation is present in all kinds of literature related to educational basics such as standards, curricula, and competency models. Contrary, object-oriented programming is often seen only as a tool and not as a topic itself. Because of this, the representation of programming aspects is rather low. As mentioned in the section on further research, the development of a competency model for programming in general and object-oriented programming in particular is one of the challenges that have to be faced in the next years (**RQ1**). The question on the proper methodology for introducing object orientation in an introductory course can be seen as answered. Based on the conceptual change theory, a paradigm change from the procedural to the object-oriented paradigm is very difficult. Because of that, the objects-early approach is the one that has the lowest difficulty level and can be seen as state of the art. Nevertheless, in literature the implementation of this approach is still discussed and there is no consensus on the first language and environment (**RQ2**).

The second aspect of this thesis was to find a graphical representation for the concepts of a specific topic and their interdependencies. The representation is especially useful for definitions and specifications that are spread over a textbook or course materials. In Chapter 6 the concept specification maps are introduced for that purpose (**RQ3**). The small investigation on the textbooks that are recommended for introductory courses around the world presented an application of the methodology. One result of this analysis is that the three main textbooks are to a large extent similar. But, in detail, small differences can be found, which can have an effect on the learning process; especially the relations among the main concepts of object orientation (*object, class, method*, and *attribute*). Some of the books associate the advanced concepts with *object, class*, or *method*. This can have a direct influence on the learning process (**RQ4**).

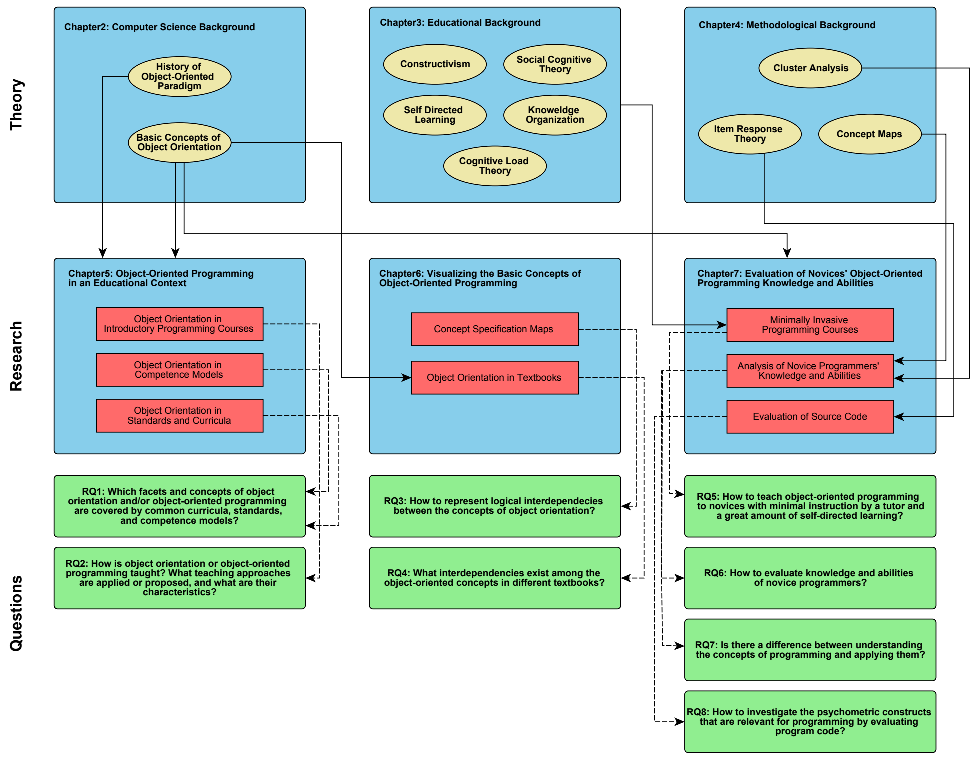
The minimally invasive programming courses (see Chapter 7) showed that object-oriented programming can be taught with little instruction. The courses are an excellent basis for research on novice programmers' knowledge and abilities as the course materials can be controlled in a better way than in a normal introductory course. Nevertheless, there are some difficulties in the course design; for example, influence of the tutors. Even with instruction not to instruct the participants, the exact influence was not traceable. The self-directed learning approach was successful concerning the self-assessment of the participants (**RQ5**). Additionally, the analysis of the produced program code showed several results. The items constructed out of the given material made it possible to conduct several methodologies on the code. First, the cluster analysis of the novice programmers' programming abilities is an applicable approach for finding differences within the population. For example, this can be used for providing different learning strategies to the examined groups. Additionally, faults in the instruction or course materials can be found if no group applied a specific concept that should have been used for the solution (**RQ6**).

The second analysis conducted on the programming results was the comparison of the knowledge of object-oriented programming concepts and the abilities expressed by the successful application of the code items. This analysis showed that there are several concepts that are applied, but cannot be included in a knowledge structure on object orientation. Furthermore, this combination of cognitive knowledge theory and code analysis provides a useful tool for finding and especially identifying misconceptions in the programming process (**RQ7**).

The most important result of this thesis is the proof that the application of the item response theory and especially the Rasch modeling on program code is applicable. After construction of implicit items out of the results produced by the participants of the minimally invasive programming courses, a suitable item set was calculated. After checking the homogeneity and the rejection of some items due to fitting reasons, a set of items could be included in a logistic model. Because the courses were not designed for that reason, the results cannot be generalized. However, the basic idea is a success. If the programming task is designed for that purpose and the groups can be compared with each other, and the whole model is proven by several populations, this work can provide a clue for the development of a competency model for object-oriented programming (**RQ8**).

Finally, the developed course design, the representation of concept structures in course materials and literature, and the basic notions of a proven evaluation tool will not solve the major problem of object-oriented programming. "Few computing educators of any experience would argue that students find learning to program easy. Most teachers will be accustomed to the struggles of their first year students as they battle in vain to come to grips with this most basic of skills and many will have seen students in later years carefully choosing options so as to minimize the risk of being asked to undertake any programming" (Jenkins 2002, p. 53).

Although the paradigm shift has reached computer science education, the problems remain in the missing compulsory subject in most countries. Generally, another problem cannot be solved. The absence of proven methodologies makes it hard to find generalizable results either for teaching or for evaluation reasons. Most research is still on methodology. This thesis gives an overview on the state of object orientation in education and provides a look at object orientation through the lens of computer science education with some new implications from other subjects like the humanities. Nevertheless, there are several options for further research that are described in the last section.


Theory
Research
Questions
Chapter2: Computer Science Background
History of Object-Oriented Paradigm
Basic Concepts of Object Orientation
Chapter3: Educational Background
Constructivism
Social Cognitive Theory
Self Directed Learning
Knoweldge Organization
Cognitive Load Theory
Chapter4: Methodological Background
Cluster Analysis
Item Response Theory
Concept Maps
Chapter5: Object-Oriented Programming in an Educational Context
Object Orientation in Introductory Programming Courses
Object Orientation in Competence Models
Object Orientation in Standards and Curricula
Chapter6: Visualizing the Basic Concepts of Object-Oriented Programming
Concept Specification Maps
Object Orientation in Textbooks
Chapter7: Evaluation of Novices' Object-Oriented Programming Knowledge and Abilities
Minimally Invasive Programming Courses
Analysis of Novice Programmers' Knowledge and Abilities
Evaluation of Source Code
RQ1: Which facets and concepts of object orientation and/or object-oriented programming are covered by common curricula, standards, and competence models?
RQ2: How is object orientation or object-oriented programming taught? What teaching approaches are applied or proposed, and what are their characteristics?
RQ3: How to represent logical interdependecies between the concepts of object orientation?
RQ4: What interdependencies exist among the object-oriented concepts in different textbooks?
RQ5: How to teach object-oriented programming to novices with minimal instruction by a tutor and a great amount of self-directed learning?
RQ6: How to evaluate knowledge and abilities of novice programmers?
RQ7: Is there a difference between understanding the concepts of programming and applying them?
RQ8: How to investigate the psychometric constructs that are relevant for programming by evaluating program code?

8.2 Further Research

8.2.1 Further Work on Concept Specification Maps

The concept specification maps were presented to provide an overview on the various concepts around object orientation. Additionally, they were applied to analyze the working materials of the minimally invasive programming courses.

For textbook or working material analysis, the maps have to be enhanced. The development of methodology to find intersections within the maps is of great interest. By applying such methodology, a quantitative analysis of materials could be done based on concept specifications. Generally, the quantitative analysis can be enhanced. This would propose some capabilities to compare courses or materials.

Another interesting aspect for future research on the concept specification maps is the combination of the specification in the maps with an ontology such as the one presented by Hubwieser and Bitzl (2010). If the ontology promotes time relations between the concepts, the resulting map can propose wrong orders of the material or difficulties during learning based on the materials given.

In addition to improving the maps, another aspect that is only a side effect of Chapter 6 is the systematic analysis of course materials proposed in introductory courses. Even the small investigation conducted in this thesis showed a broad variety in the recommended literature. An analysis of the textbooks and their fitting to the provided concepts during the course would be an interesting research area to improve the introductory courses and give an empirical founded recommendation of additional materials for a specific topic. These recommendations can be given on the level of one concept by using the concept specification maps.

8.2.2 Further Work on Evaluating Novice Programmers' knowledge and abilities

Several ideas for analyzing and categorizing novice programmers' knowledge and abilities and object-oriented program code in general are shown in Chapter 7. These ideas provide several possibilities for further research.

The code items that are the basis for the cluster analysis and the analysis based on the item response theory have to be investigated more closely. The concepts examined from the course materials were the basis for the items. But, there are further concepts related to programming in general, especially if other programming paradigms, such as the functional paradigm are taken into account. Furthermore, the creation of the items out of the concepts has to be validated.

The Rasch modeling of the program code shows interesting aspects. As shown in Section 7.6, the modeling does work, but, as expressed in Section 7.7, the selection of the tasks needs some improvement. Based on a more extensive list of items, creation of suitable tasks for a proven assessment tool should be possible. All the difficulties mentioned in Section 7.7 have to be taken into account for further research in this area. Nevertheless, for moving towards competency models such as those presented in Section 5.2, a tool to evaluate students' programming abilities is very important.

Additionally, there are several code items that had to be excluded from the dataset due to model violations. In future research it is important to find a way to either include the concepts by identifying other code items or extend the investigated population. Hubwieser and Mühling (2014) present an implementation of the latent trait analysis for finding homogeneous item sets in a pool of items. Nevertheless, as there are numerous combinations in this thesis' item list of 34 items, the approach has to be improved to handle all permutations. After that, an investigation of all homogeneous itemsets that are included in the item list is an interesting area for further research.

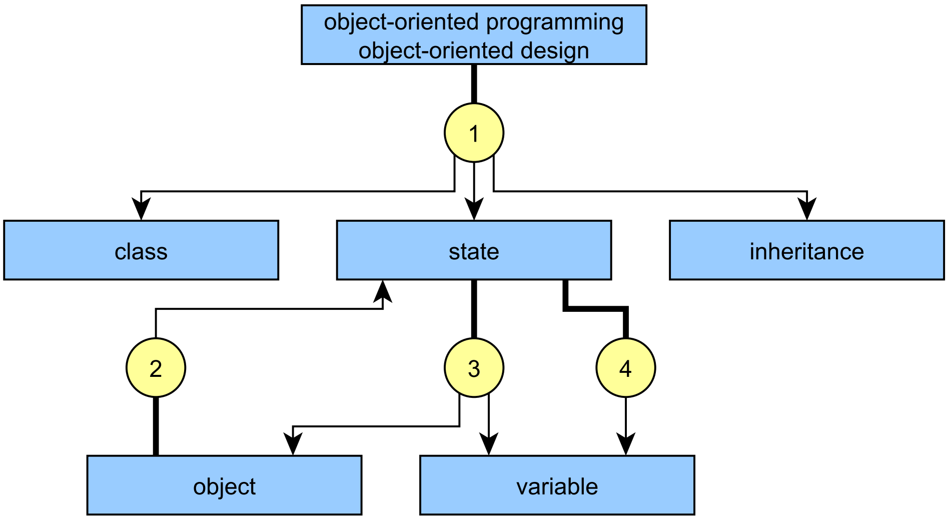
Regarding the programming abilities, the role of computational thinking and especially problem solving for the programming task, is of great interest. Here, investigating the involved cognitive processes and their modeling is a serious challenge for further research. More precisely, further research should aim to explore a multi-dimensional model displaying all facets of programming. For that purpose, the research presented in Section 7.6 conducts the basic approach.

Furthermore, the correlation of knowledge and abilities in programming purposes is another interesting facet of this thesis. The ability facets can be measured with an improved model similar to the one presented in Section 7.6. The aspect of knowledge assessment through concept maps was investigated in detail by Mühling (2014).

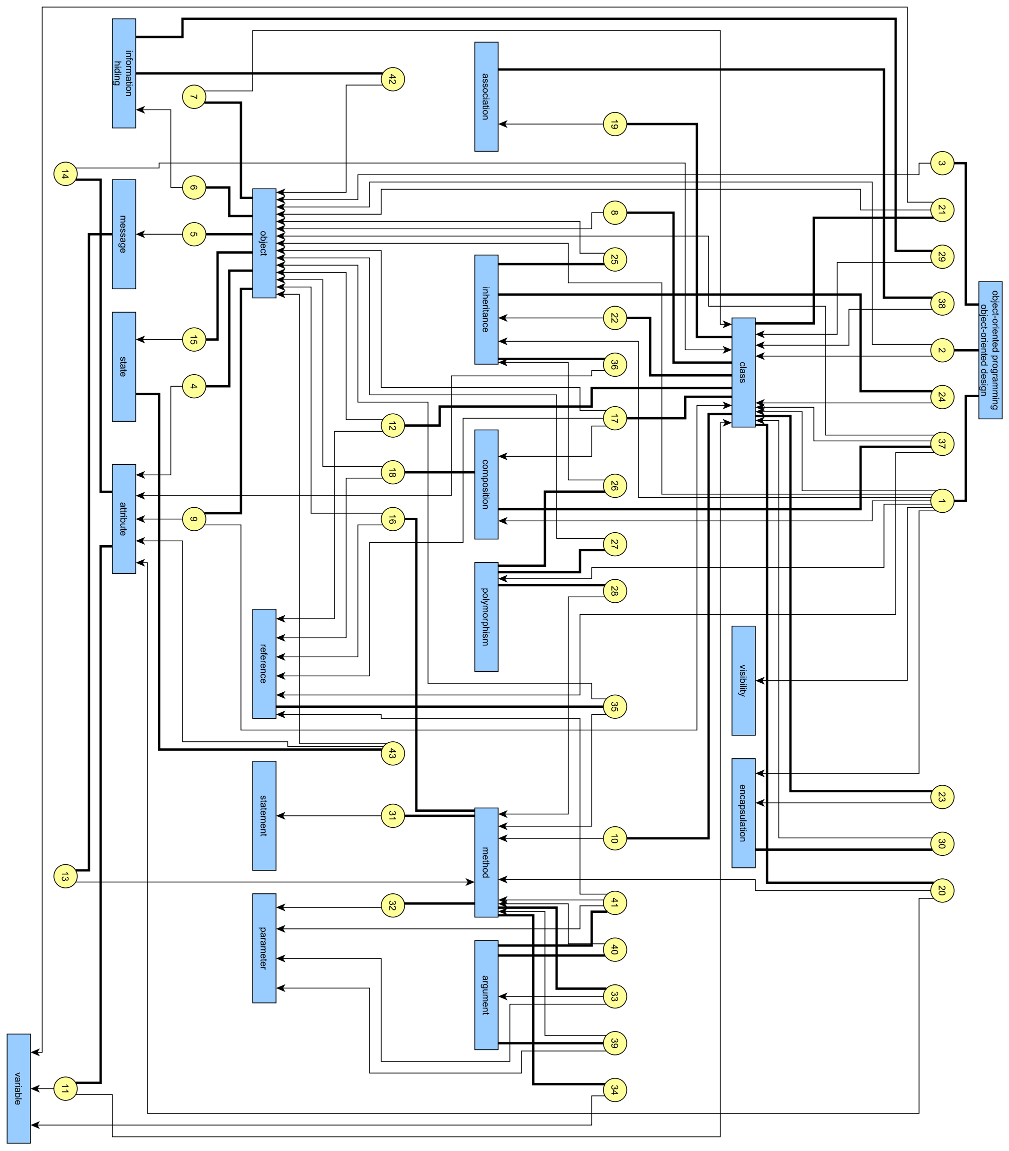
Another aspect of the comparison of knowledge and abilities is the relation to the taxonomy of Fuller et al. (2007) – as for some concepts, the ability to apply them and the ability to interpret them are very different. The taxonomy is based on the revised taxonomy of Bloom (Anderson and Krathwohl 2009). The cognitive process dimension is divided into two dimensions. The first dimension represents the abilities by "producing" cognitive processes; the other dimension represents the knowledge by "interpreting" cognitive processes. The two dimensions are thought to be orthogonal. Further research may be able to focus more on that taxonomy in order to obtain further evidence of its validity in object-oriented programming.

A Concept Specification Map (CMS)

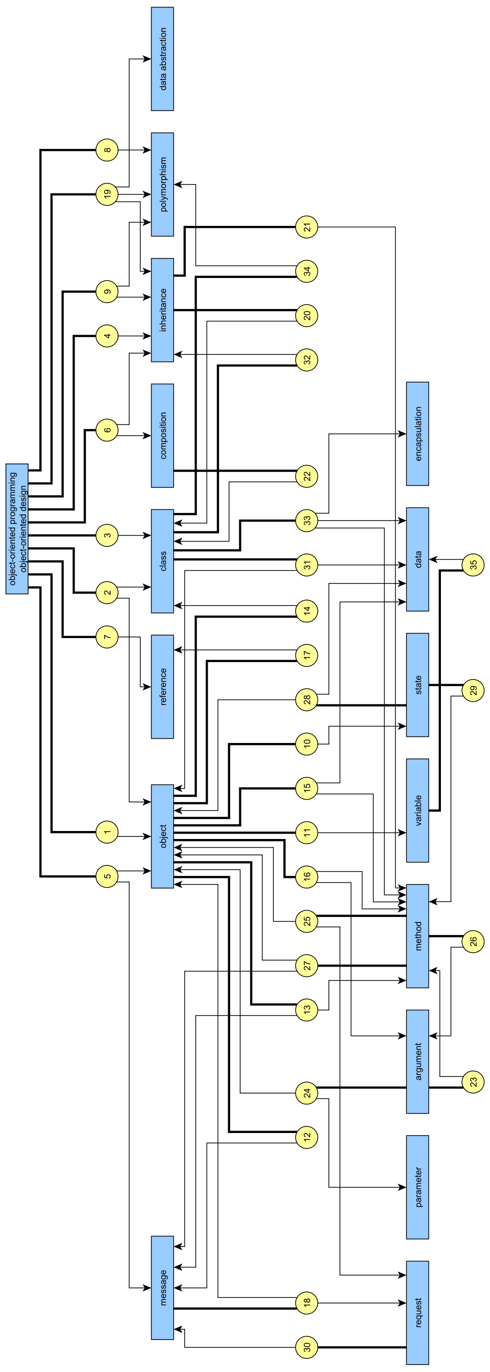
A.1 Abelson - Structure and Interpretation of Computer Programs



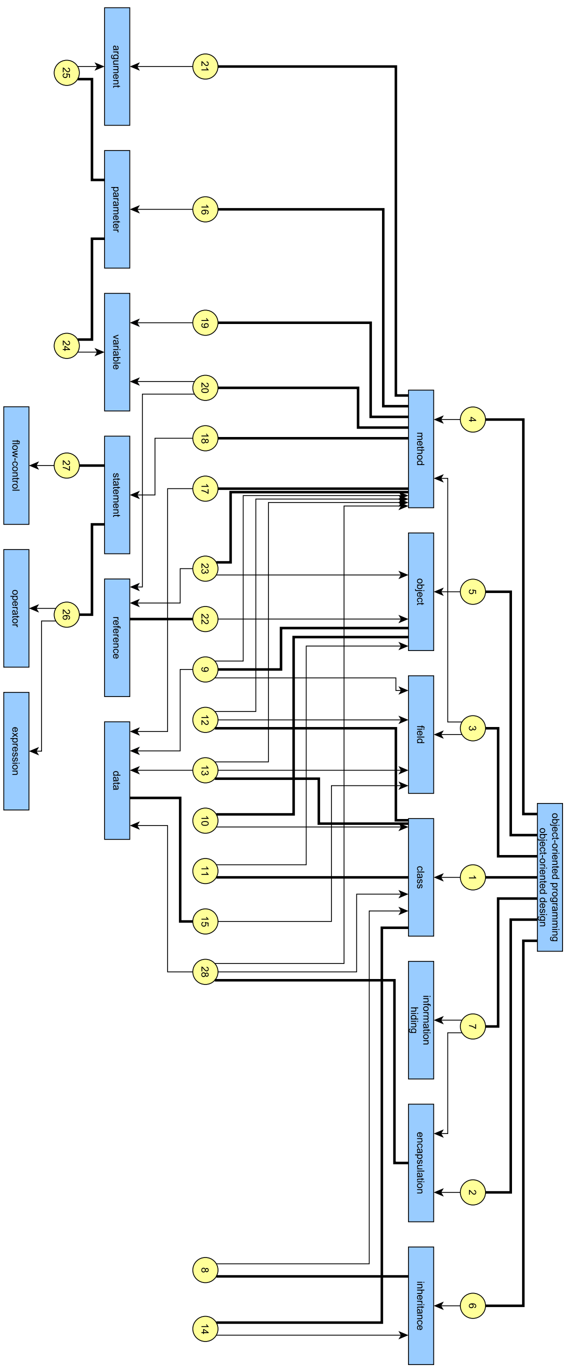
A.2 Deitel - How to Program Java



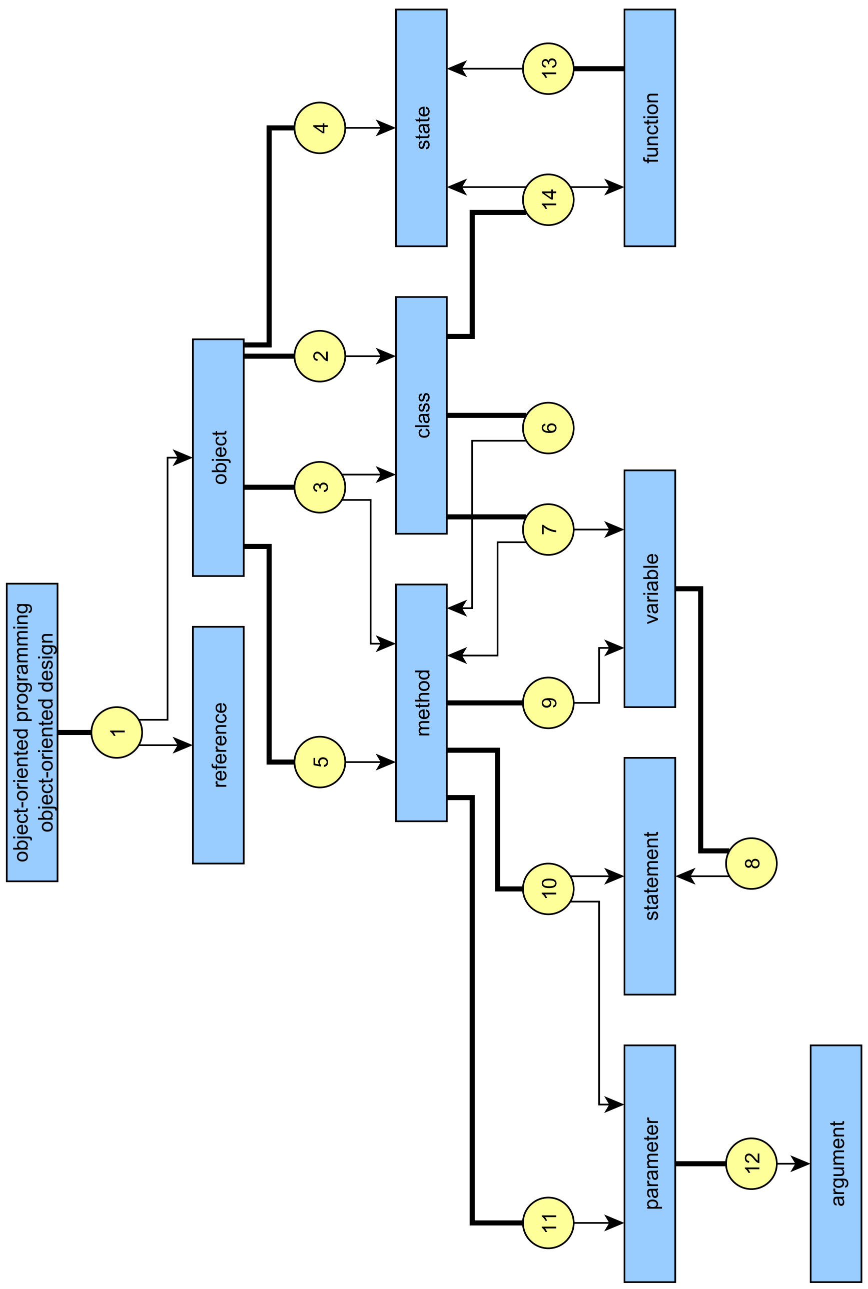
A.3 Eckel - Thinking in Java



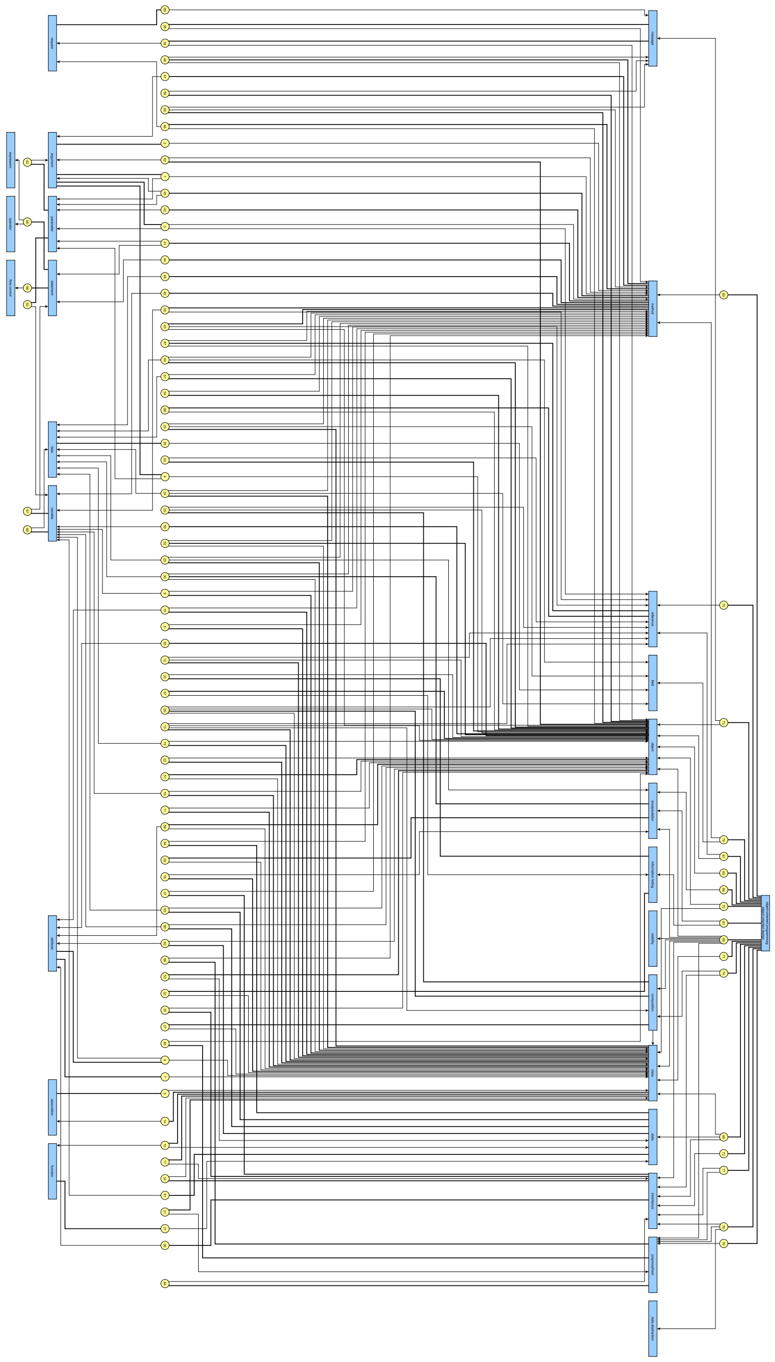
A.4 Flanagan - Java in a Nutshell



A.5 Sedgewick - Introduction to Programming in Java



A.6 CSM - All books



B Minimally Invasive Programming Courses (MIPC)

B.1 Worksheets MIPC

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Programmier-projekte

Prof. Dr. Peter Hubwieser
Marc Berges

Vorprojekte WS10/11

Blatt 1
Allgemeine Hinweise und Einführung in die Übungsplattformen

I. Hinweise

- Im Laufe des Projektes erhalten Sie **vier Übungsblätter**. Die Blätter enthalten jeweils die notwendigen theoretischen Grundlagen, sowie die Teilaufgaben zur Erstellung des Gesamtprojektes.
- Es werden drei verschiedene Projekte zur Auswahl gestellt. Diese unterscheiden sich im Umfang und im Schwierigkeitsgrad.
- Die Übungsblätter sollen einzeln in einem selbst gewählten Tempo bearbeitet werden.

II. Entwicklungswerkzeuge

Im Laufe des Projektes werden Sie verschiedene Programme benutzen müssen. Um den Einstieg zu erleichtern, sollen diese hier kurz vorgestellt werden. Im Aufgabenteil dieses Blattes können Sie sich dann mit den einzelnen Werkzeugen vertraut machen.

BlueJ:
Als Entwicklungsumgebung für Java wird in dieser Veranstaltung BlueJ verwendet. Der Vorteil dieser Entwicklungsumgebung liegt in der guten Übersicht über die verwendeten Klassen. Es werden nicht nur die Klassen an sich dargestellt, sondern auch die Assoziationen zwischen ihnen. Durch die Möglichkeit Objekte mit Mausclicks zu erzeugen eignet sich BlueJ vor allem für die ersten Erfahrungen mit Objektorientierung.

StarUML:
StarUML ist ein Modellierungswerkzeug zum Erstellen von Klassendiagrammen. Diese werden für die Modellierung des Projektes gebraucht. Mit Hilfe von Vorlagen und Drag&Drop lassen sich die Diagramme sehr einfach zusammenbauen.

ObjectDraw:
Mit ObjectDraw lassen sich einfache Grafiken auf Basis von Grafikobjekten erstellen. Diese Umgebung dient besonders dazu das Konzept von Objekten und Klassen zu verdeutlichen.

Eclipse:
Eclipse ist eine sehr umfangreiche Entwicklungsumgebung für Java und andere Programmiersprachen. Durch die Fülle an Funktionen ist sie als Einstiegsumgebung nicht geeignet. Nach einiger Programmiererfahrung lässt sich allerdings sehr komfortabel damit arbeiten.

III. Projektauswahl

Es werden folgende Projekte zur Auswahl gestellt:

1.	**Mastermind (leicht)** Ein Spieler legt zu Beginn einen vierstelligen geordneten Zahlencode fest, der aus den zehn Ziffern zusammengesetzt ist. Ein weiterer Spieler versucht den Code herauszufinden. Dazu setzt er einen gleichartigen Zahlencode als Frage. Auf jeden Zug hin bekommt der Ratende die Information, wie viele Elemente er in Ziffer und Position richtig gesetzt hat, bzw. wie viele zwar in der Ziffer, nicht aber in der Position übereingestimmt haben. Ziel des Spiels ist es, so schnell wie möglich, jedoch in höchstens zwölf Schritten, den Code zu erraten.
2.	**Ballsportmanager (mittel)** In diesem Projekt sollen die Ergebnisse einer Ballsportart angelegt und ausgewertet werden können. Fußball würde sich als Sportart anbieten, es ist aber auch jede andere Sportart mit mehreren Spielern und einem Punkteergebnis möglich. Im Ballsportmanager sollen die Spiele angelegt und eine Ergebnistabelle ausgegeben werden können. Ein Spiel wird immer von zwei Mannschaften bestritten und endet mit einem Punkteergebnis. Ob ein Spiel unentschieden enden kann, hängt von der Sportart ab. Die Mannschaften bestehen aus einer festen Anzahl an Spielern, zu denen auch die Reservespieler zählen.
3.	**Kniffel (schwer)** Das bekannte Spiel Kniffel oder Yahtzee ist eines der meisterverkauften Würfelspiele der Welt. Für das Spiel benötigt man fünf Würfel. In jeder Runde darf man bis zu drei Mal hintereinander würfeln. Dabei darf man Würfel, die „passen“ zur Seite legen und die restlichen neu werfen. Spätestens nach dem dritten Wurf muss man sich entscheiden, welches Feld man mit dem Ergebnis bewerten will. Es gibt zwei Blöcke, in die man das Ergebnis eintragen kann. Der erste Block ist der „Sammelblock“. Hier werden die Würfel mit der passenden Augenzahl zusammengezählt und in das entsprechende Feld eingetragen. Ist die Summe der Punkte aller sechs Felder größer als 63, wird ein Bonus von 35 Punkten hinzugezählt. Der zweite Block beinhaltet die Felder „Dreierpasch“, „Viererpasch“, „Full House“, „Kleine Straße“, „Große Straße“, „Kniffel“ und „Chance“. Beim „Dreier-„ und „Viererpasch“ zählen alle Augen. Für das „Full House“ gibt es 25 Punkte, für die „Kleine Straße“ 30 und für die „Große Straße“ 40 Punkte. Die höchste Punktzahl (50) gibt es für den „Kniffel“. In „Chance“ können alle Augen eingetragen werden. Gewonnen hat derjenige, der aus den beiden Blöcken und dem Bonus die meisten Punkte erzielt hat.

IV. Aufgaben

1. Starten Sie das Programm „ObjectDraw“ und machen Sie sich mit der Umgebung vertraut.
2. Mit Hilfe der Klassen Rechteck, Linie und Kreis lässt sich in ObjectDraw ein Fußballplatz mit Spielern zeichnen. Beobachten Sie die Veränderungen in den Objektkarten. Dazu muss das Analysatorfenster geöffnet sein.
3. Starten Sie die Entwicklungsumgebung BlueJ und laden Sie das Beispielprojekt „Shapes“. Dieses ist im Ordner „Examples“ im BlueJ-Verzeichnis zu finden.
4. Mit „Shapes“ lassen sich verschiedene grafische Objekte erzeugen und anzeigen. Legen Sie sich eine Reihe von Objekten an und experimentieren Sie mit diesen, indem Sie Farbe, Form und ähnliches ändern.

Prof. Dr. Peter Hubwieser
Marc Berges

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Vorprojekte WS10/11

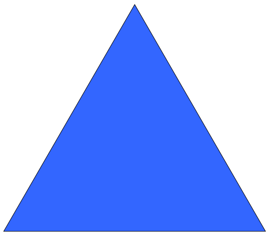
Programmier-projekte

Blatt 2
Klassen, Attribute, Methoden und Objekte

I. Theorie

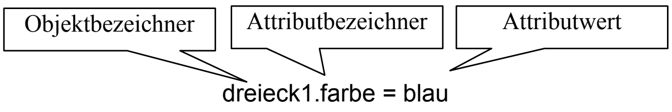
Auf dem ersten Blatt haben Sie bereits Objekte und Klassen kennen gelernt. Aber was steht hinter diesen beiden Begriffen? In diesem Abschnitt sollen die wichtigsten Aspekte der beiden Konzepte kurz erklärt werden. Im Anschluss daran beginnen Sie mit der Umsetzung des Projektes.

Objekt:
Unsere „reale Welt“ besteht aus Objekten. Alles was wir sehen kann als Objekt bezeichnet werden. Wenn diese Objekte elektronisch repräsentiert werden sollen müssen einige Dinge beachtet werden. Als Beispiel soll hier ein blaues Dreieck betrachtet werden. Um von einem konkreten Objekt sprechen zu können, müssen wir dem Computer mitteilen welches Objekt gemeint ist. Dafür ist ein eindeutiger Bezeichner notwendig. In unserem Beispiel lautet der Bezeichner „*Dreieck1*“. Ein Objekt hat einen bestimmten Zustand, der durch seine Eigenschaften bestimmt ist. Diese Eigenschaften werden Attribute genannt. Der Zustand eines Objektes lässt sich über Methoden ändern.

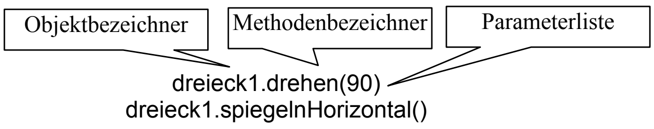


dreieck1 : DREIECK

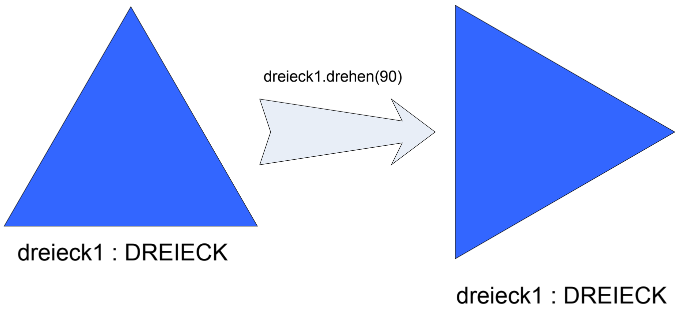
Attribut:
Attribute bezeichnen die Eigenschaften eines Objektes. Sie beschreiben den Zustand des Objektes. Dieser wird durch die Attributwerte festgelegt. In unserem Beispiel ist das Attribut *farbe* mit dem Attributwert **blau** belegt. Andere Attribute wären z.B. *seitenlaenge*, *winkel1*, *winkel2*, *winkel3*. Die Werte der Attribute lassen sich über Methoden ändern. Um klar zu machen, von welchem Attribut welches Objektes mit welchem Wert man spricht, gibt es eine kurze Schreibweise, die sog. Punktnotation:

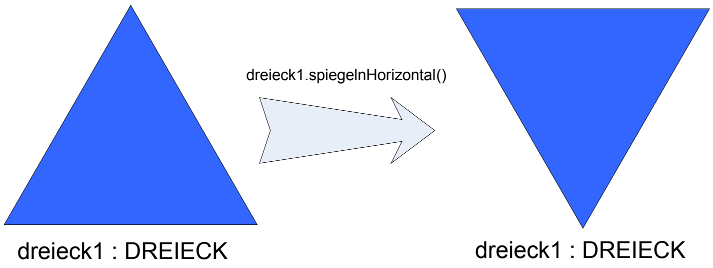


Methode:
Mit Hilfe von Methoden lassen sich die Attributwerte von Objekten verändern. Sie dienen aber auch dazu zwischen Objekten zu kommunizieren oder Operationen auszuführen. Um einer Methode Eingabewerte zu übergeben, können Argumente, sog. Parameter angegeben werden. Als Schreibweise wird wieder die Punktnotation verwendet. Dem Methodenname folgt dabei die Parameterliste in einer Klammer. Sind keine Parameter notwendig, bleibt die Klammer leer, muss aber mit angegeben werden, um den Unterschied zu den Attributen sicher zu stellen.

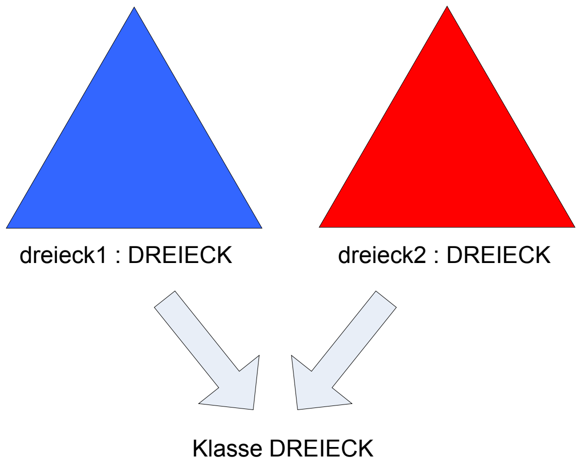


```
dreieck1.drehen(90)
dreieck1.spiegelnHorizontal()
```



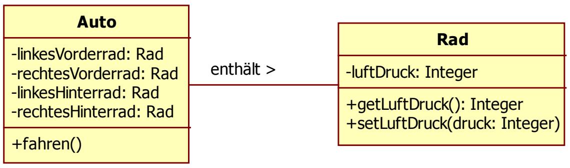


Klasse:
Um festzulegen, welche Attribute ein Objekt haben kann ist es notwendig, die Objekte mit gleichen Attributen (aber in der Regel unterschiedlichen Attributwerten) zusammenzufassen. Mehrere gleichartige Objekte werden in einer Klasse beschrieben. Im Beispiel mit dem blauen Dreieck ließen sich mehrere Dreiecke zu einer Klasse Dreieck zusammenfassen. Sie alle haben die gleichen Attribute, aber unterschiedliche Attributwerte. In der Klasse befindet sich also eine Art Konstruktionsplan für die entsprechenden Objekte. Die Klasse ist keine Menge von Objekten und kann daher auch ohne Objekte existieren. Auf Blatt1 haben Sie dies bereits erfahren können. Im Projekt „Shapes“ sind beim Öffnen die Klassen „Circle“, „Canvas“, „Square“ und „Triangle“ vorhanden. Über den Aufruf des Konstruktors wurden dann Objekte der jeweiligen Klasse erzeugt, die genau die vorgegebenen Attribute und Methoden der Klasse enthalten. Der Konstruktor ist eine spezielle vordefinierte Methode. Mit ihm wird ein Objekt einer Klasse erzeugt.



Das Prinzip der Datenkapselung:
Das zentrale Prinzip der Objektorientierung ist das Prinzip der Datenkapselung. Das heißt, dass die Attribute und die zugehörigen Werte eines Objektes von außen (also von anderen Objekten) nicht sichtbar sind. Die Attributwerte können nur mit Hilfe von Methoden geändert werden. Dadurch kann sichergestellt werden, dass Attributwerte nur auf geeignete Art und Weise geändert werden können. Objektorientierte Programmiersprachen stellen entsprechende Zugriffsmodifikatoren zur Verfügung, um private und öffentliche Attribute und Methoden zu kennzeichnen. Private Attribute/Methoden sind nur im eigenen Objekt sichtbar. Öffentliche Attribute/Methoden können von außen aufgerufen und gesehen werden. In Java werden private Attribute/Methoden mit *private* und öffentliche mit *public* gekennzeichnet.

Darstellung von Klassen:
Eine übersichtliche Darstellung von Klassen lässt sich mit einem Klassendiagramm realisieren. Angelehnt an die Sprache UML werden Klassen als Rechtecke dargestellt, die in drei Teile geteilt sind. Im oberen Teil steht der Bezeichner der Klasse. Darunter folgt die Liste der Attribute mit dem Datentyp, den das Attribut hat. Der Datentyp schränkt die Werte ein, die ein Attribut annehmen kann. Die Attributwerte haben in einer Klasse natürlich nichts zu suchen. Zusätzlich werden vor dem Attributbezeichner die Zugriffsrechte angegeben. Für *private* schreiben wir ein Minus, für *public* ein Plus. Im untersten Teil werden die Methoden mit Parameterliste und eventuellem Rückgabetyp angegeben. Auch hier werden die Zugriffsrechte durch ein Minus, bzw. Plus gekennzeichnet. Will man einen Zusammenhang zweier Klassen darstellen, werden die Klassen durch eine einfache Linie verbunden. Auf der Linie steht der Bezeichner der Assoziation. Die Richtung wird durch einen Pfeil (am Computer durch „größer“, bzw. „kleiner“) dargestellt.



II. AUFGABEN

1. In BlueJ gibt es die Möglichkeit auf erstellte Objekte mit Hilfe der Punktnotation Einfluss zu nehmen. Aktivieren Sie die Funktion „Code Pad“ im Menü „View“ und versuchen Sie die Farbe und Form ihrer Objekte zu verändern, indem sie die Methoden mit Hilfe der Punktnotation aufrufen.
2. Lesen Sie sich die Aufgabenstellung auf Blatt1 aufmerksam durch und versuchen Sie die beteiligten Objekte mit Ihren Attributen zu identifizieren. Erstellen Sie eine Liste mit den Objekten.
3. Versuchen Sie nur aus der Liste in Aufgabe2 die benötigten Klassen zu verallgemeinern.
4. Öffnen Sie ein neues Projekt in StarUML und zeichnen Sie Ihre ermittelten Klassen in das Diagramm.
5. Überlegen Sie sich, mit welchen Attributen Sie die Klassen charakterisieren wollen und fügen Sie sie in Ihr Klassendiagramm ein.
6. Im nächsten Schritt müssen die Assoziationen zwischen den Klassen dargestellt werden. Zeichnen Sie diese in Ihr Diagramm ein.
7. Zur Vervollständigung des Modells fehlen noch die Aktionen, die die Objekte ausführen können. Überlegen Sie sich welche Methoden die einzelnen Klassen enthalten sollen und vervollständigen Sie Ihr Modell.

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Programmier-projekte

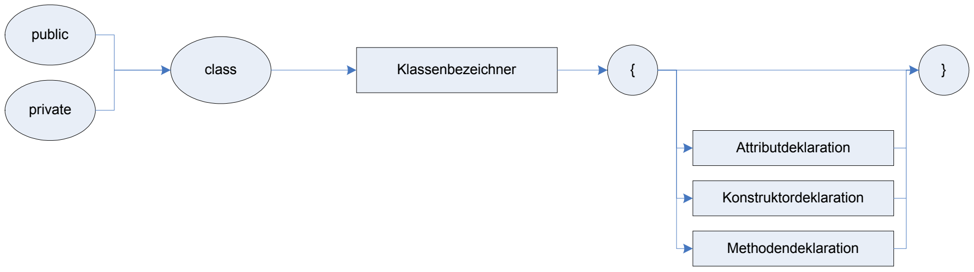
Prof. Dr. Peter Hubwieser
Marc Berges

Vorprojekte WS10/11

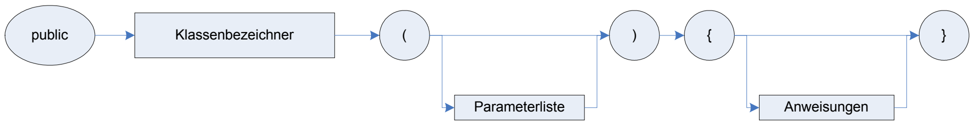
Blatt 3
Umsetzung in Java

I. Theorie

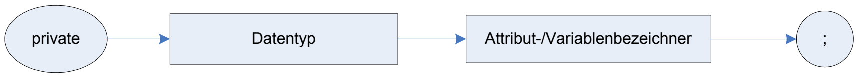
Klasse:
Eine Klasse wird mit dem Schlüsselwort *class* eingeleitet. Darauf folgt der Klassenbezeichner. Dieser wird üblicherweise groß geschrieben. In geschweiften Klammern eingeschlossen folgt der Klassenrumpf mit Attributen, Konstruktoren und Methoden.



Konstruktor:
Der Konstruktor dient zum Erstellen von Objekten. Er ist die erste Methode, die aufgerufen wird, wenn das Objekt erstellt wird. Als erstes steht der Klassenbezeichner. Darauf folgen die Parameterliste in runden Klammern und die Anweisungen in geschweiften Klammern. Jede Anweisung muss mit einem Strichpunkt von der nächsten getrennt werden.

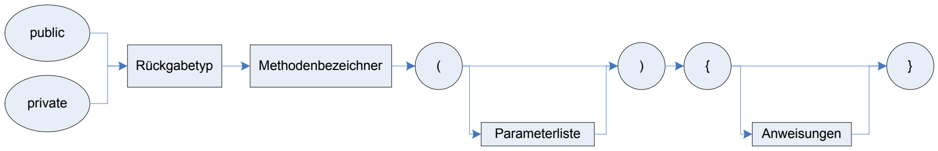


Attribute:
Attribute werden in Java in folgender Reihenfolge deklariert. Zuerst steht der Zugriffsmodifikator. Daran anschließend folgt der Datentyp. Als letztes folgt der Attributbezeichner, der üblicherweise mit einem kleinen Buchstaben beginnt. Um eine bessere Übersichtlichkeit zu gewährleisten, beginn jedes Teilwort mit einem Großbuchstaben. Will man dem Attribut einen Standardwert zuweisen, ist dies im Anschluss an den Attributbezeichner möglich.



Methoden:
Die Methodendeklaration in Java ist wie folgt aufgebaut. Soll die Methode einen Rückgabewert bereitstellen ist der Datentyp der Rückgabe zu Beginn der Deklaration

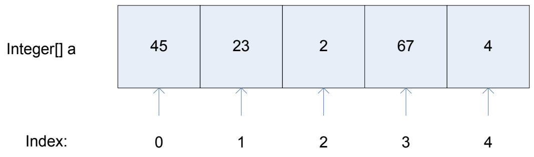
aufzuführen. Ansonsten wird mit dem Typ *void* angezeigt, dass keine Rückgabe zu erwarten ist. Danach folgt der Methodenbezeichner. Auch dieser wird wie die Attribute üblicherweise mit einem Kleinbuchstaben begonnen. Jedes Teilwort beginnt dann wieder mit einem Großbuchstaben. Nach dem Namen folgen die Parameterliste in runden und die Anweisungen der Methode in geschweiften Klammern. Ist im Methodenkopf ein Rückgabewert definiert worden, muss im Methodenrumpf eine explizite Rückgabe mit der return-Anweisung erfolgen. Mit dem Schlüsselwort *return*, gefolgt vom eigentlichen Rückgabewert, wird die Methode beendet. Nachfolgender Code wird nicht mehr berücksichtigt. Methoden können auch gleich heißen, müssen sich dann aber in der Parameterliste unterscheiden. Diesen Fall nennt man überladen. Dies ist auch bei den Konstruktoren möglich.



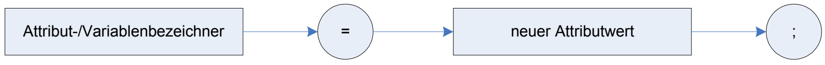
Zugriffsmodifikatoren:
In Java ist es möglich verschiedene Sichtbarkeitsbereiche für Attribute, Klassen und Methoden zu definieren. Die zwei wichtigsten sind *private* und *public*. Mit *private* versehene Attribute, Methoden oder Klassen sind nur innerhalb der Klasse sichtbar. Von außerhalb kann darauf nicht zugegriffen werden. Dieser Modifikator ist vor allem für die Kapselung in der Objektorientierung wichtig. Mit *public* werden die entsprechenden Elemente von außen sichtbar gemacht.

Datentypen:
In Java gibt es drei Arten von Datentypen. Zum einen die sog. primitiven Datentypen. Dazu zählen unter anderem die ganzen Zahlen (***int***), Fließkommazahlen (***double***), Zeichen (***char***) und Wahrheitswerte(***boolean***). Die zweite Gruppe leitet sich direkt von diesen Datentypen ab. Es sind die sog. Wrapper-Klassen. Dieses sind die objektorientierten Varianten der primitiven Datentypen. Bis auf *int* und *char* bleiben die Typbezeichner gleich, werden aber groß geschrieben. Für *int* lautet die entsprechende Wrapper-Klasse ***Integer***, für *char* ist es ***Charakter***. Die dritte Art sind die Objekttypen. Dabei kann jede Klasse als Datentyp verwendet werden. Der bekannteste Vertreter dieser Gruppe ist die Zeichenkette, also der Datentyp ***String***.

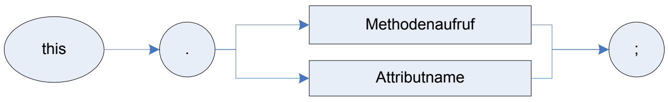
Felder:
Java bietet die Möglichkeit Felder mit bekannter Länge und von einem Datentyp anzulegen. Dabei ist es egal, ob es sich um primitive Datentypen oder Objekttypen handelt. Die Deklaration der sog. Arrays erfolgt über das Anfügen von eckigen Klammern nach dem Datentyp. Um das Feld nutzen zu können, muss die Länge definiert sein. Dies kann gleich bei der Variablendeklaration, oder auch später geschehen. Der Syntax dafür ist eine Zuweisung gefolgt vom Datentyp und eckigen Klammern, in die die Länge eingetragen wird. Die Elemente können mit dem Attributbezeichner gefolgt von eckigen Klammern, die den Index des Elements enthalten, aufgerufen werden. Der Index startet bei 0.



Wertzuweisung:
Für die Wertzuweisung schreibt man den Attributbezeichner gefolgt von = und dem Wert, der zugewiesen werden soll. Es ist auch möglich, einen weiteren Attributbezeichner oder eine Zelle eines Feldes anzugeben. Java verwendet dann automatisch den Attributwert.



Klassenbezug this:
Mit *this* kann auf die eigene Klasse Bezug genommen werden. Dies ist besonders dann nützlich, wenn in einer Methode die Parameter denselben Bezeichner haben, wie Attribute der Klasse. Um dann klar darzustellen, welcher Bezug wie hergestellt werden soll, muss *this* eingesetzt werden.



II. Beispielklasse

```
/**
 * Write a description of class TestClass here.
 *
 * @author (your name)
 * @version (a version number or a date)
 */

public class TestClass
{
    // Attribute

    private int beispielAttribut; //Der Datentyp ist int
    private int[] beispielArray;

    /*
     * Dies ist ein mehrzeiliger Kommentar vor dem Konstruktor
     */

    //Der Konstruktor trägt denselben Namen wie die Klasse

    public TestClass()
    {
      // Im Konstruktor können z.B. die Attribute mit Initialwerten
      //versehen werden
      //mit this wird auf das eigene Objekt verwiesen

        this.beispielAttribut = 0;
        this.beispielArray = int[5];
        this.beispielArray[0] = 2;
    }

    /**
     * Dies ist ein Kommentar im JavaDoc-Format
     *
     * @param  y   a sample parameter for a method
     * @return     the sum of x and y
     */
    public int beispielMethode(int y)
    {
        //die Methode liefert die Summe von beispielAttribut und dem
        // Produkt der ersten Zelle des Arrays und y zurück,
        //die vorher in eine lokale Variable gespeichert wurde

        int produkt = this.beispielArray[0] * y;
        int summe = this.beispielAttribut + produkt;
        return summe;
    }

    public int getBeispielAttribut()
    {
        return this.beispielAttribut;
    }
}
```

III. AUFGABEN

1. Öffnen Sie BlueJ und legen Sie ein neues Projekt an.
2. Mit der Schaltfläche „Neue Klasse“ lassen sich neue Klassen erzeugen. Legen Sie die Klassen aus Ihrem Modell an.
3. Im nächsten Schritt müssen die Attribute hinzugefügt werden. Öffnen Sie dazu die entsprechende Klasse mit einem Doppelklick und fügen Sie die Attribute vor dem leeren Konstruktor ein.
4. Als weiterer Schritt sollen nun die Methoden ergänzt werden. In einem ersten Schritt werden dazu die Methodenköpfe mit leeren Methodenrümpfen geschrieben. Bei den Methoden mit Rückgabewerten müssen Sie eine return-Anweisung mit angeben.
5. In einem letzten Schritt programmieren Sie bitte die get- und set-Methoden aus. Dazu schreiben Sie bitte in den get-Methoden den Rückgabewert und in den set-Methoden die Zuweisung der Attributwerte.

Prof. Dr. Peter Hubwieser
Marc Berges

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Vorprojekte WS10/11

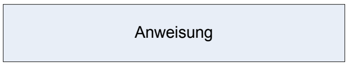
Programmier-projekte

Blatt 4
Algorithmen und Kommunikation

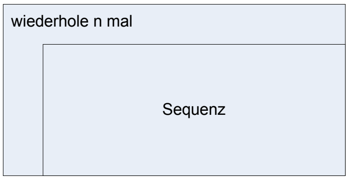
I. Theorie

Algorithmus:
Algorithmen beschreiben das eindeutige Vorgehen zum Lösen eines Problems. Dabei werden Eingaben in einer endlichen Zahl von Schritten in eine Ausgabe gewandelt. Das klassische Beispiel ist das Kochrezept, wo man die Zutaten als Eingabe und das fertige Gericht als Ausgabe hat. In der Informatik funktioniert das Definieren von Algorithmen auf ganz ähnliche Art und Weise. Zuerst muss das Problem klar dargestellt werden. Dann muss geklärt werden, was die Eingabegrößen sind. Dann wird das Vorgehen in kleinen Schritten beschrieben. Am Ende steht die Ausgabegröße. Um Algorithmen graphisch darzustellen, gibt es mehrere Möglichkeiten. Zwei davon sollen hier kurz dargestellt werden. Die erste Möglichkeit ist das Struktogramm. Dabei werden alle Elemente eines Algorithmus als Blöcke dargestellt, die ineinander verschachtelt werden können. Folgende Blöcke sind möglich:

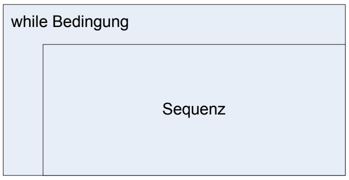
einfache Anweisung



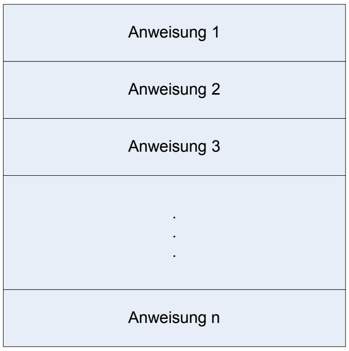
Wiederholung mit fester Anzahl



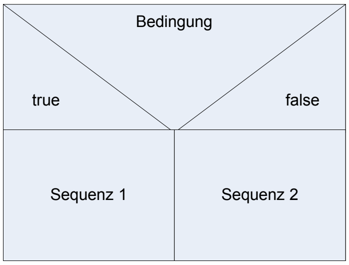
Wiederholung mit Bedingung



Sequenz

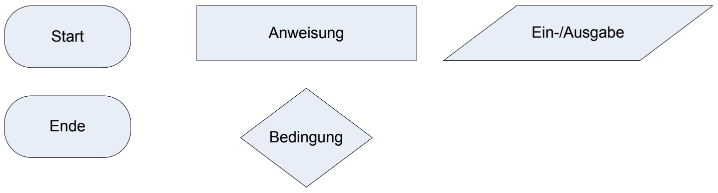


bedingte Anweisung

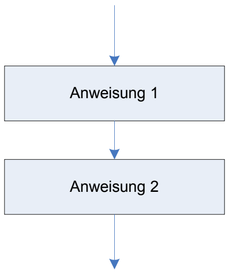


Die zweite Möglichkeit ist das Ablaufdiagramm. Hier sind folgende Elemente zu verwenden:

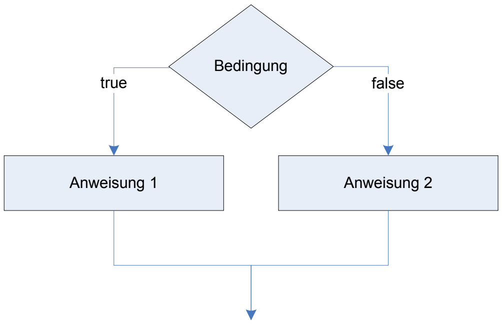
Standardblöcke



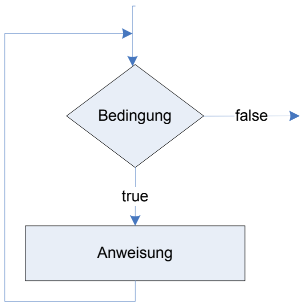
Folge von Anweisungen



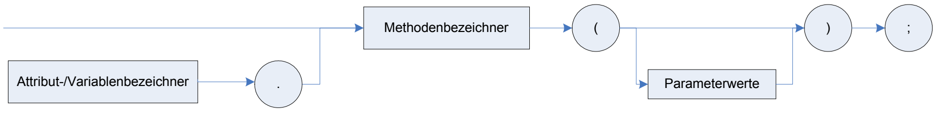
Anweisung mit Bedingung



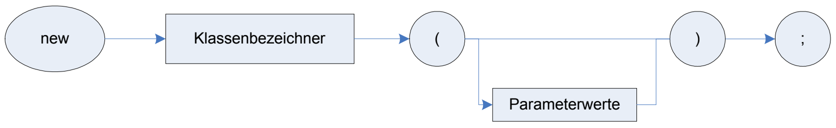
Anweisung mit Schleife



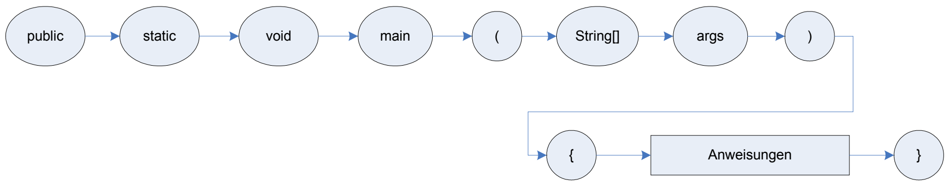
Methodenaufruf:
Beim Aufruf von Methoden wird die Punktnotation verwendet. Zuerst kommen der Attribut- bzw. Variablenbezeichner, dann der Methodenbezeichner und schließlich die Parameterwerte, die übergeben werden sollen. Diese müssen in der richtigen Reihenfolge angegeben werden. Soll eine Methode der eigenen Klasse aufgerufen werden, kann der Attribut- bzw. Variablenbezeichner weggelassen, oder durch **this** ersetzt werden.



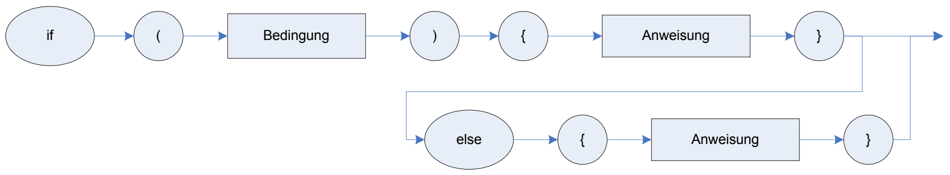
Erzeugen von Objekten:
Um mit Attributen oder Variablen von Objekttypen arbeiten zu können, müssen diese Objekte zuerst erzeugt werden. Dies geschieht, indem dem Attribut oder der Variablen eine neues Objekt zugewiesen wird, das mit dem Operator ***new***, gefolgt vom Klassennamen und den durch den Konstruktor geforderten Parameterwerten in runden Klammern, erzeugt wird. Eine Ausnahme bildet der Objekttyp String, bei dem kein Objekt explizit erzeugt werden muss. Dies geschieht automatisch, wenn dem Attribut bzw. der Variablen eine Zeichenkette zugewiesen wird.



main-Methode:
Um ein Programm außerhalb von BlueJ starten zu können ist es nötig, eine Methode zu definieren, die beim Start ausgeführt wird. Dies ist in Java die main-Methode. Ist diese in einer Klasse enthalten, kann die Klasse ausgeführt werden. Die main-Methode hat immer den gleichen Rumpf. Der Inhalt kann dann individuell angepasst werden. Unter anderem können hier die ersten Objekte erzeugt werden und der Ablauf des Programms definiert werden.



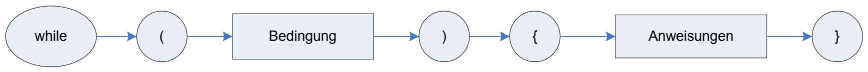
if-Anweisung:
In Java wird eine Fallunterscheidung durch das if-Konstrukt dargestellt. Es gibt zwei Varianten. Eine mit und eine ohne Alternative. Wenn keine Alternative vorgesehen ist, wird der entsprechende Teil einfach weggelassen. Die if-Anweisung startet mit dem Schlüsselwort ***if***. Darauf folgend steht die Bedingung in runden Klammern. Die Bedingung muss ein boolscher Ausdruck sein, d.h. er muss mit wahr oder falsch zu beantworten sein. Nach der Bedingung kommen die durchzuführenden Anweisungen in geschweiften Klammern. Ist eine Alternative gewünscht, steht jetzt das Schlüsselwort ***else*** gefolgt von den Anweisungen der Alternative wieder in geschweiften Klammern.



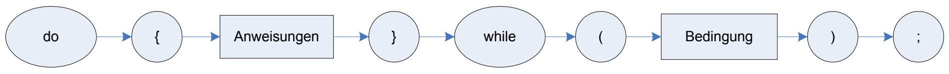
Schleifen:
Java bietet drei verschiedene Schleifenarten. Die erste ist die Wiederholung mit Anfangsbedingung. Bei dieser Schleifenart wird die angegebene Bedingung zu Beginn eines jeden Durchlaufs geprüft. Ist die Bedingung erfüllt, werden die Anweisungen der Schleife ausgeführt. Ist sie nicht erfüllt, ist die Schleife beendet. Die Wiederholung wird mit

dem Schlüsselwort ***while*** eingeleitet. Darauf folgen die Bedingung in runden Klammern und die Anweisungen der Schleife in geschweiften Klammern. Die zweite Schleife ist die Wiederholung mit Endbedingung. Sie ist ähnlich aufgebaut wie die erste Schleife, die Bedingung wird jetzt allerdings erst am Ende des Schleifendurchlaufs geprüft. Ist die Bedingung erfüllt, kommt es zu einem weiteren Durchlauf. Ist sie es nicht, ist die Schleife beendet. Diese Schleife beginnt mit dem Schlüsselwort ***do***, gefolgt von den Anweisungen in geschweiften Klammern. Am Ende steht das Schlüsselwort ***while***, gefolgt von der Bedingung in runden Klammern. Die letzte Schleife ist die Wiederholung mit fester Anzahl. Genau genommen ist es nur eine abkürzende Schreibweise der Wiederholung mit Anfangsbedingung. Die for-Schleife beginnt mit dem Schlüsselwort ***for***. In der darauf folgenden runden Klammer stehen folgende Elemente in fester Reihenfolge. Zuerst wird eine Schleifenvariable initialisiert. Von einem Strichpunkt getrennt folgt dann die Bedingung, die zum Ende der Schleife führt. Das letzte Element in der Klammer gibt den Ausdruck an, mit dem die Schleifenvariable nach jedem Schritt angepasst werden soll. Nach dieser runden Klammer folgen die Anweisungen in geschweiften Klammern.

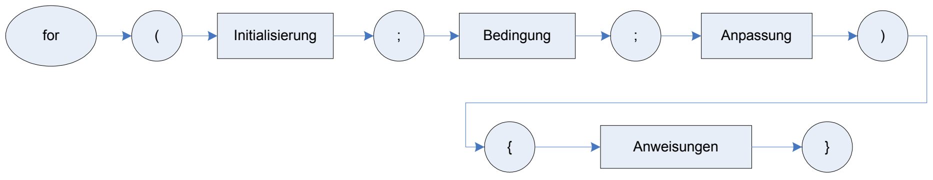
while-Schleife



do-while-Schleife

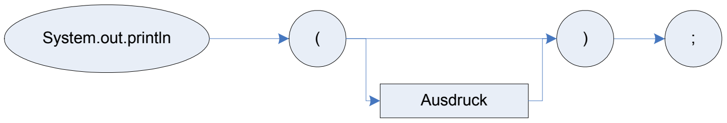


for-Schleife



Operatoren:
Java stellt eine ganze Reihe von Operatoren zur Verfügung. Zunächst einmal seien die mathematischen Operatoren genannt. Die Addition und Subtraktion werden ganz normal mit **+** und **–** dargestellt. Die Multiplikation wird durch *******, die Division durch ***/*** repräsentiert. Zusätzlich gibt es eine Möglichkeit, den Rest einer Division mit ***%*** zu berechnen. Als Kurzschreibweisen für das Erhöhen bzw. Verringern einer ganzen Zahl gibt es die Operatoren **++** und **--**. Die Zuweisung ist wie schon im letzten Blatt erwähnt mit **=** umgesetzt. Eine weitere große Gruppe bilden die logischen Operatoren. Hier ist an erster Stelle der Vergleichsoperator **==** zu erwähnen. Die Größenvergleiche werden ganz normal durch **<**, **<=**, **>=** und **>** umgesetzt. Ungleich wird mit ***!=*** dargestellt. Der Operator ***!*** bildet die Verneinung. ***&&*** verknüpft zwei logische Ausdrücke mit UND, **||** zwei Ausdrücke mit ODER.

Standardausgabe:
Um eine Textausgabe auf der Konsole zu bewerkstelligen, gibt es in Java die Standardausgabefunktion. Der Syntax lautet System.out.println(), wobei in die runden Klammern der Ausgabetext geschrieben werden muss.



Benutzereingabe:
Die Benutzereingabe ist in Java nur schwer möglich. Um den Vorgang der Eingabe zu vereinfachen gibt es eine Klasse In.java. Hier werden diverse Methoden zu Verfügung gestellt, mit denen die unterschiedlichsten Werte eingelesen werden können. Die häufigsten Aufrufe sollen hier kurz erwähnt werden: In.readBoolean() – In.readChar() – In.readDouble – In.readInt() – In.readString.

II. Aufgaben

1. Ermitteln Sie die Algorithmen Ihres Projektes. Überlegen Sie sich dazu für alle Methoden, die nicht nur Attributwerte setzen oder zurückgeben, was diese machen sollen. Schreiben Sie sich zu jedem Schritt, den die Methode durchführen soll einen kurzen Satz auf. Bei Wiederholungen oder Bedingungen empfiehlt es sich mit Zeilennummern zu arbeiten.
2. Nehmen Sie die Beschreibungen aus Aufgabe 1 und erstellen Sie jeweils ein Strukogramm oder ein Ablaufdiagramm.
3. Anhand der Beschreibungen und Diagramme aus den letzten beiden Aufgaben lassen sich die gewünschten Funktionen gut in Java umsetzen. Vervollständigen Sie nun Ihren Quelltext und probieren Sie die einzelnen Methoden mit BlueJ aus. Wenn Fehler auftreten, machen Sie sich Gedanken, wie Sie den Fehler möglichst schnell finden können.
4. Damit das Programm auch ohne BlueJ laufen kann ist es nötig die Hauptklasse mit einer main-Methode auszustatten. Schreiben Sie eine solche Methode und führen Sie das Programm von der Konsole aus. Informationen dazu können Sie im Internet recherchieren.
5. Wenn Ihr Programm sicher läuft, fehlt ihm nur noch ein passendes Gewand. Wenn Sie noch Zeit haben, versuchen Sie doch einmal eine Benutzeroberfläche zu erstellen. Auch hier kann die Internetrecherche weiterhelfen.

B.2 Specifications of the Worksheets

Id	Specification	Specified Concept	Specifying Concepts
1	Ein Objekt hat einen bestimmten Zustand, der durch seine Eigenschaften bestimmt ist. Diese Eigenschaften werden Attribute genannt. Der Zustand eines Objektes lässt sich über Methoden ändern.	Objekt	Zustand Attribut Methode
2	Attribute bezeichnen die Eigenschaften eines Objektes. Sie beschreiben den Zustand des Objektes. Dieser wird durch die Attributwerte festgelegt.	Attribut	Objekt Zustand
3	Die Werte der Attribute lassen sich über Methoden ändern	Attribut	Methode
4	Mit Hilfe von Methoden lassen sich die Attributwerte von Objekten verändern	Methode	Objekt Attribut
5	Sie dienen aber auch dazu zwischen Objekten zu kommunizieren oder Operationen auszuführen.	Methode	Objekt
6	Um einer Methode Eingabewerte zu übergeben, können Argumente, sog. Parameter angegeben werden	Methode	Parameter
7	Um festzulegen, welche Attribute ein Objekt haben kann ist es notwendig, die Objekte mit gleichen Attributen (aber in der Regel unterschiedlichen Attributwerten) zusammenzufassen. Mehrere gleichartige Objekte werden in einer Klasse beschrieben	Klasse	Objekt Attribut
8	In der Klasse befindet sich also eine Art Konstruktionsplan für die entsprechenden Objekte. Die Klasse ist keine Menge von Objekten und kann daher auch ohne Objekte existieren.	Klasse	Objekt
9	Der Konstruktor ist eine spezielle vordefinierte Methode. Mit ihm wird ein Objekt einer Klasse erzeugt.	Konstruktor	Methode Objekt Klasse
10	Das zentrale Prinzip der Objektorientierung ist das Prinzip der Datenkapselung. Das heißt, dass die Attribute und die zugehörigen Werte eines Objektes von außen (also von anderen Objekten) nicht sichtbar sind. Die Attributwerte können nur mit Hilfe von Methoden geändert werden. Dadurch kann sichergestellt werden, dass Attributwerte nur auf geeignete Art und Weise geändert werden können.	Datenkapselung	Attribut Objekt Methode
11	Der Konstruktor dient zum Erstellen von Objekten. Er ist die erste Methode, die aufgerufen wird, wenn das Objekt erstellt wird.	Konstruktor	Objekt Methode

B.3 Questionnaire MIPC

Umfrage zu den Vorprojekten
Wintersemester 2010/11

46

Personendaten:

Geschlecht:	☐männlich ☐weiblich
Alter:	☐unter 20 ☐20 – 25 ☐ über 25
Studiengang:	☐Informatik ☐Wirtschaftsinformatik ☐Bioinformatik
	☐LA Informatik ☐anderer:__________________
Block:	☐1 ☐2 ☐3 ☐4
Vorkenntnisse:	☐ohne ☐schon programmiert ☐schon objektorientiert programmiert
Schulausbildung Informatik:	☐Leistungskurs ☐Grundkurs ☐Pflichtfach ☐Wahlfach ☐keine
Herkunft/ Bundesland:	☐BE ☐BB ☐BW ☐BY ☐HB ☐HH ☐HE ☐MV ☐NDS ☐NRW ☐RP ☐SA ☐SH ☐SL ☐SN ☐TH ☐Ausland:________________

Veranstaltung:

Fanden Sie die Ankündigung rechtzeitig?

rechtzeitig ☐ ☐ ☐ ☐ ☐ zu spät/früh

Wie fanden Sie die Organisation?

gut ☐ ☐ ☐ ☐ ☐ schlecht

Waren Sie mit dem Ablauf zufrieden?

zufrieden ☐ ☐ ☐ ☐ ☐ nicht zufrieden

Wie haben Sie die Gruppengröße empfunden?

zu groß ☐ ☐ ☐ ☐ ☐ zu klein

Blätter:

Wie haben Sie den Informationsgehalt der Arbeitsblätter empfunden?

hoch ☐ ☐ ☐ ☐ ☐ niedrig

Wie verständlich waren die Informationen auf den Arbeitsblättern?

gut ☐ ☐ ☐ ☐ ☐ schlecht

Wie ausführlich fanden Sie die Arbeitsblätter?

nicht ausführlich ☐ ☐ ☐ ☐ ☐ zu ausführlich

bitte wenden!

Tutor:

Wie hilfreich fanden Sie die Anwesenheit eines Tutors?

sehr ☐ ☐ ☐ ☐ ☐ gar nicht

Waren die Erklärungen des Tutors verständlich?

verständlich ☐ ☐ ☐ ☐ ☐ nicht verständlich

Selbsteinschätzung:

Haben Sie in den Vorprojekten etwas gelernt?

viel ☐ ☐ ☐ ☐ ☐ wenig

Wie schätzen Sie Ihre Javakenntnisse nach den Vorprojekten ein?

nicht vorhanden ☐ ☐ ☐ ☐ ☐ selbstständiges Programmieren

Haben Sie das Konzept der Objektorientierung verstanden?

verstanden ☐ ☐ ☐ ☐ ☐ nichtverstanden

Kommentare, Kritik, Vorschläge zu den "Vorprojekten" :

Vielen Dank und viel Erfolg in Ihrem Studium

B.4 Concept Map Questionnaire MIPC

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

167

Prof. Dr. Peter Hubwieser
Marc Berges

Vorprojekte WS10/11

Fragebogen zur
Objektorientierung
Vortest

WAS IST OBJEKTORIENTIERUNG?

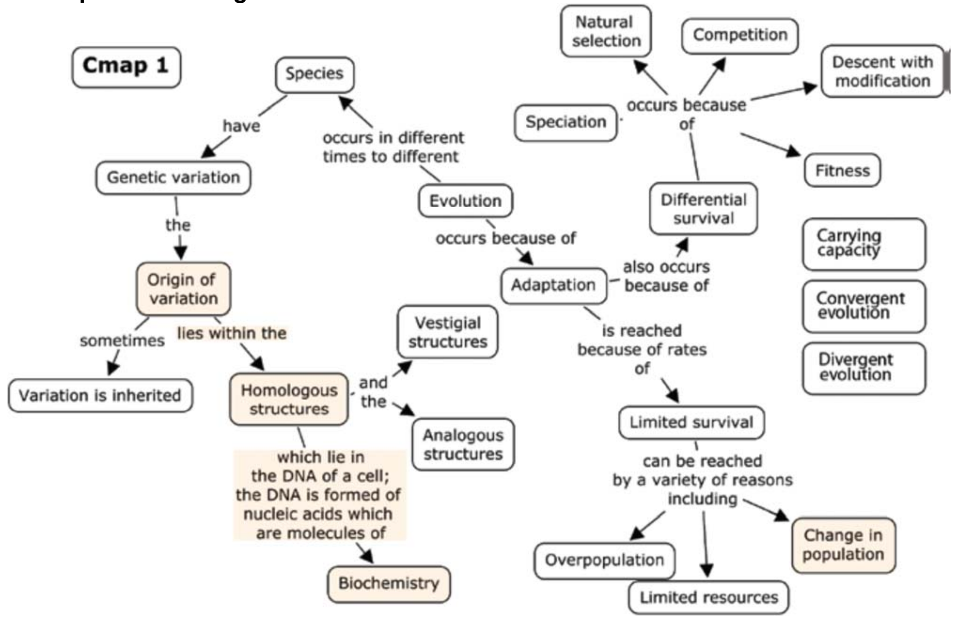
1. Aufgabe
In der untenstehenden Tabelle finden Sie Konzepte und Begriffe aus dem Bereich der Objektorientierung. Bitte geben Sie in der nachfolgenden Begriffstabelle an, ob Sie den jeweiligen Begriff kennen, d.h., ob Sie ihn ggf. erklären oder definieren könnten

Bitte zeichnen Sie dann auf dem 2. Blatt ein Begriffsnetz aus den Begriffen, die Sie in der Begriffstabelle als bekannt angekreuzt haben. Falls Sie jeweils zwei der gezeichneten Begriffe in Verbindung bringen können, zeichnen Sie diese Verbindung als Pfeil zwischen diesen Begriffen ein, allerdings nur, wenn Sie dafür auch einen Bezeichner angeben können. Es müssen nicht alle eingezeichneten Begriffe zueinander in Beziehung stehen.

2. Begriffstabelle

		bekannt			
Begriff	**unbekannt**	**vorher**	**durch Tutor**	**durch Recherche**	**durch Sonstiges**
Zustand	☐	☐	☐	☐	☐
Assoziation	☐	☐	☐	☐	☐
Instanz	☐	☐	☐	☐	☐
Klasse	☐	☐	☐	☐	☐
Methode	☐	☐	☐	☐	☐
Datenkapselung	☐	☐	☐	☐	☐
Operatoren	☐	☐	☐	☐	☐
Felder	☐	☐	☐	☐	☐
Datentyp	☐	☐	☐	☐	☐
Initialisierung	☐	☐	☐	☐	☐
Parameter	☐	☐	☐	☐	☐
Attribute	☐	☐	☐	☐	☐
Fallunterscheidung	☐	☐	☐	☐	☐
Objektorientierung	☐	☐	☐	☐	☐
Zuweisung	☐	☐	☐	☐	☐
Objekt	☐	☐	☐	☐	☐
Überladen	☐	☐	☐	☐	☐
Vererbung	☐	☐	☐	☐	☐
Wiederholung	☐	☐	☐	☐	☐
Zugriffsmodifikator	☐	☐	☐	☐	☐
Konstrukor	☐	☐	☐	☐	☐

3. Beispiel für ein Begriffsnetz



Beispiel aus Kern, C and Crippen K.J.: Mapping for Conceptual Change, The Science Teacher, September 2008

4. Ihr Begriffsnetz zur Objektorientierung
Verwenden Sie bitte nur die Begriffe, die sie als bekannt angekreuzt haben. Geben Sie zu jeder gezeichneten Verbindungskante einen Bezeichner an.

Bitte deutlich schreiben!!! «Zahlen»

B.5 Report Form for the Participant Questions

Technische Universität München
Fakultät für Informatik
Fachgebiet Didaktik der Informatik

Prof. Dr. Peter Hubwieser
Marc Berges

Vorprojekte WS10/11

Protokoll Studentenfragen

PROTOKOLL STUDENTENFRAGEN

Bitte alle Fragen der Studenten protokollieren. Für Fragen aus der Liste reicht die Nummer. Bitte geben Sie möglichst genau an, wie auf die Frage geantwortet wurde.
Erklärung der Spalten:

Tipp:	Als Antwort wird eine Lösungsidee genannt.
Code:	Als Antwort wird Java-Code selber in das Programm geschrieben oder dem Teilnehmer detailliert genannt.
Hinweis:	Als Antwort wird ein Hinweis auf weiterführende Texte gegen.

	Antwort durch			
Frage	**Tipp**	**Code**	**Hinweis**	**NR**
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	
	☐	☐	☐Java Insel ☐Doku ☐IDE-Hilfe	

B.6 Code Examples

B.6.1 Example for The Mastermind Task

```
/**
 * Bitte geben Sie Ihren Mastercode ein!
 */
public class Master
{

    int[] code;

    public Master(){
        code = new int[4];
    }
    public void CodeErstellen()
    {
        for (int i=0;i<4;i++)
        {
            System.out.print ("Master, geben Sie die "+i+". Zahl (zwischen 0
                und 9, aber keine doppelt) Ihres Codes ein: ");
            code[i] = In.readInt();
        }

    }

    public int[] getcode(){
        return code;
    }
}

/**
 * Spielercodeeingabe und Abgleich mit Mastercode.
 */

public class Spieler
{
    int[] code;

    public Spieler()
    {
        code = new int[4];
    }

public static void main (String[] args)
    {
        System.out.println ("Moegen die Spiele beginnen!");
        System.out.println ("Der Master gibt zuerst seinen Mastercode ein.");
        System.out.println ("Dann ist der Spieler an der Reihe\nund muss
            innerhalb von 12 Zuegen diesen Mastercode erraten!");

        Spieler michael = new Spieler();
        Master sepp = new Master();
        sepp.CodeErstellen();
        michael.aufMasterZugreifen(sepp);
    }
```

```
public void aufMasterZugreifen (Master master)
{
int count = 0;
   while ((code[0] != master.code[0] || code[1] != master.code[1] || code[2]
        != master.code[2] || code[3] != master.code[3]) && count < 12)
   {
        count++;
        for (int i=0;i<4;i++)
        {
            System.out.print ("Spieler, geben Sie die "+i+". Zahl (zwischen 0
                und 9, aber keine doppelt) Ihres Codes ein: ");
            code[i] = In.readInt();
        }


        if (code[0] == master.code[0])
        {
            System.out.println ("Die 1. Zahl stimmt ueberein.");

        }
            else
            {
                if (code[0] == master.code[1] || code[0] == master.code[2] ||
                    code[0] == master.code[3])
                {
                    System.out.println ("Die 1. Zahl ist zwar vorhanden, aber
                        nicht an der richtigen Stelle.");
                }
                else
                {
                    System.out.println ("Die 1. Zahl ist nicht vorhanden.");
                }
            }


        if (code[1] == master.code[1])
        {
            System.out.println ("Die 2. Zahl stimmt ueberein.");

        }
            else
            {
                if (code[1] == master.code[0] || code[1] == master.code[2] ||
                    code[1] == master.code[3])
                {
                    System.out.println ("Die 2. Zahl ist zwar vorhanden, aber
                        nicht an der richtigen Stelle.");
                }
                else
                {
                    System.out.println ("Die 2. Zahl ist nicht vorhanden.");
                }
            }


        if (code[2] == master.code[2])
        {
            System.out.println ("Die 3. Zahl stimmt ueberein.");
        }
            else
            {
                if (code[2] == master.code[0] || code[2] == master.code[1] ||
                    code[2] == master.code[3])
                {
                    System.out.println ("Die 3. Zahl ist zwar vorhanden, aber
                        nicht an der richtigen Stelle.");
                }
                else
                {
                    System.out.println ("Die 3. Zahl ist nicht vorhanden.");
                }
            }


        if (code[3] == master.code[3])
        {
            System.out.println ("Die 4. Zahl stimmt ueberein.");

        }
            else
            {
                if (code[3] == master.code[0] || code[3] == master.code[1] ||
                    code[3] == master.code[2])
                {
                    System.out.println ("Die 4. Zahl ist zwar vorhanden, aber
                        nicht an der richtigen Stelle.");
                }
                else
                {
                    System.out.println ("Die 4. Zahl ist nicht vorhanden.");
                }
            }


   if (count == 12 && (code[0] != master.code[0] || code[1] != master.code[1]
       || code[2] != master.code[2] || code[3] != master.code[3]))
   {System.out.println ("GAME OVER!!!");}

   if (code[0] == master.code[0] && code[1] == master.code[1] && code[2] ==
       master.code[2] && code[3] == master.code[3])
   {System.out.println ("Sie haben in "+count+" Zuegen gewonnen!");}

   if (count<12 && (code[0] != master.code[0] || code[1] != master.code[1] ||
       code[2] != master.code[2] || code[3] != master.code[3]))
   {int difference = 12 - count;
    System.out.println ("Sie haben noch "+difference+" Versuche!");}
   }
}
}
```

B.6.2 Example for The Ballsportmanager Task

```
import java.io.IOException;

public class Ballmanager {

        /**
         * Teilnehmernummer 2
         */

        public static String[] mannschaftsArray = new String[100];
        public static int mannschaftsIndex = 0; //Legt den Index fuer die
            Array-Positionen der Mannschaften fest.
        //Gleichzeitig die Anzahl der Mannschaften im Array.

        public static Spiel[] Spielarray = new Spiel[99];  //Hier werden die
            einzelnen Spiel-Objekte gespeichert.
        public static int spielIndex = 0;       //Legt den Index fuer die
            Array-Positionen der Spiel-Objekte fest.
        //Gleichzeitig die Anzahl der Spiele im Array.

        public static String mannschaftsName;
        public static String[][] rowData = new String[99][99];

        public static void main(String args[])
        {
                new Oberflaeche("Ballsportmanager");

                int x = 0;
                while (x<100) //Inhalt der Arrays gleich null setzen.
                {
                        mannschaftsArray[x] = null;
                        x++;
                }
                /*      System.out.println("Hallo Welt. Ballsportmanager
                    gestartet. Zunaechst muessen mindestens zwei Mannschaften
                    hinzugefuegt werden.\nErste Mannschaft erstellen, bitte
                    den Namen eingeben:");
                        In.close();
                        mannschaftsName = In.readLine();
                        mannschaftsArray[mannschaftsIndex] = mannschaftsName;

                        System.out.println("Mannschaftsname der ersten
                            Mannschaft: '"+mannschaftsName+"'. Die Mannschaft
                            wurde gespeichert.\nGeben Sie eine weitere
                            Mannschaft an:");
                        mannschaftsIndex++;
                        In.close();

                        mannschaftsName = In.readLine();
                        mannschaftsArray[mannschaftsIndex] = mannschaftsName;
                        System.out.println("Mannschaftsname der zweiten
                            Mannschaft: '"+mannschaftsName+"'. Die Mannschaft
                            wurde gespeichert.\nWeiter:");
                        mannschaftsIndex++;
                        In.close();
                        */

                        try {
                                new SpieleLaden();
                                }
                        catch (IOException e)
                                {
                                        // TODO Auto-generated catch block
                                        e.printStackTrace();
                                }
                        try {
                                new MannschaftLaden();
                        } catch (IOException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }
        }
}

import java.io.FileInputStream;
import java.io.IOException;
import java.util.StringTokenizer;

public class MannschaftLaden {

        public MannschaftLaden() throws IOException
        {
             byte zeichen;
             String text ="";
             String dateiName = "mannschaften.gme";
             FileInputStream leseStrom = new FileInputStream(dateiName);
             do{
               zeichen = (byte)leseStrom.read();
               text += (char)zeichen;
             } while (zeichen !=-1);


             //System.out.println(text);
             StringTokenizer tokenizer = new StringTokenizer( text, "&" );

             int i = tokenizer.countTokens();
             for (int a=0; a<(i-1); a++ )
             {
                Ballmanager.mannschaftsArray[a]=tokenizer.nextToken();
                Ballmanager.mannschaftsIndex++;
             }
             leseStrom.close();
        }
}

import java.io.FileOutputStream;
import java.io.IOException;

public class MannschaftSpeichern{

        public MannschaftSpeichern() throws IOException {

        FileOutputStream schreibeStrom = new FileOutputStream("mannschaften.
            gme");

    for (int i=0; i < Ballmanager.mannschaftsIndex; i++)
    {
        String text = Ballmanager.mannschaftsArray[i]+"&";
            for (int j=0; j < text.length(); j++)
            {
                schreibeStrom.write((byte)text.charAt(j));
            }

    }

    schreibeStrom.close();
        }
}

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class neueMannschaft extends JFrame implements ActionListener{

         Container cp;
         JButton buttonOK;
         JTextField tField;

         public neueMannschaft(String titel) //Konstruktor
         {
                super(titel);
                                    //Uebergibt Fenstertitel an den
                    ensprechenden JFrame-Konstruktor.
                setSize(400, 100);
            setVisible(true);
                        //Macht Fenster sichtbar
            cp = getContentPane();                      //Container, um
                Festnerinhalt aufnehmen zu koennen.
            cp.setLayout(new FlowLayout());

            tField = new JTextField(16);
        cp.add(tField);
            buttonOK = new JButton("Mannschaft eintragen");
            buttonOK.addActionListener(this);
        cp.add(buttonOK);

         }

        @Override
        public void actionPerformed(ActionEvent e) {
                Object obj = e.getSource();

                if (obj == buttonOK)
                {
                        Ballmanager.mannschaftsName = tField.getText(); //
                            Mannschaftsnamen aus Textfeld holen.
                        Ballmanager.mannschaftsArray[Ballmanager.
                            mannschaftsIndex] = Ballmanager.mannschaftsName;
                            //Mannschaftsnamen ins Mannschaftsarray schreiben
                        Ballmanager.mannschaftsIndex++;
                        try {
                                new MannschaftSpeichern();
                        } catch (IOException e1) {
                                // TODO Auto-generated catch block
                                e1.printStackTrace();
                        }
                        this.dispose();
                }
        }
}

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.IOException;

public class neuesSpiel extends JFrame implements ActionListener{

         Container cp;
         JButton buttonOK;
         JComboBox auswahlHeim;
         JComboBox auswahlGast;
         JComboBox auswahlspielerHeim;
         JComboBox auswahlspielerGast;
         JComboBox auswahlToreHeim;
         JComboBox auswahlToreGast;
         JLabel doppelpunkt;
         JLabel doppelpunkt1;
         JLabel text;

         String heim ="";
         String gast ="";
         String toreHeim;
         String toreGast;
         String spielerHeim;
         String spielerGast;

         public neuesSpiel(String titel) //Konstruktor
         {
                super(titel);
                                  //uebergibt Fenstertitel an den
                    ensprechenden JFrame-Konstruktor.
                setSize(480, 270);
            setVisible(true);
                        //Macht Fenster sichtbar
            cp = getContentPane();                    //Container, um
                Festnerinhalt aufnehmen zu koennen.
            cp.setLayout(null);


            String[] tore = {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9",
                 "10"};
            String[] spieler = {"11", "12", "13", "14", "15", "16", "17", "18"
                , "19"};

            auswahlHeim = new JComboBox(Ballmanager.mannschaftsArray);
            auswahlHeim.setBounds(20, 20, 200, 20);
            cp.add(auswahlHeim);

            doppelpunkt = new JLabel(" : ");
            doppelpunkt.setBounds(225, 20, 10, 20);
```

```
    cp.add(doppelpunkt);

    auswahlGast = new JComboBox(Ballmanager.mannschaftsArray);
    auswahlGast.setBounds(240, 20, 200, 20);
    cp.add(auswahlGast);

    auswahlToreHeim = new JComboBox(tore);
    auswahlToreHeim.setBounds(170, 60, 50, 20);
    cp.add(auswahlToreHeim);

    doppelpunkt1 = new JLabel(" : ");
    doppelpunkt1.setBounds(225, 60, 100, 20);
    cp.add(doppelpunkt1);

    auswahlToreGast = new JComboBox(tore);
    auswahlToreGast.setBounds(240, 60, 50, 20);
    cp.add(auswahlToreGast);

    text = new JLabel("Zugelassene Spieleranzahl:");
    text.setBounds(155, 100, 180, 20);
    cp.add(text);

    auswahlspielerHeim = new JComboBox(spieler);
    auswahlspielerHeim.setBounds(170, 140, 50, 20);
    cp.add(auswahlspielerHeim);

    doppelpunkt1 = new JLabel(" : ");
    doppelpunkt1.setBounds(225, 140, 100, 20);
    cp.add(doppelpunkt1);

    auswahlspielerGast = new JComboBox(spieler);
    auswahlspielerGast.setBounds(240, 140, 50, 20);
    cp.add(auswahlspielerGast);

    buttonOK = new JButton("Spielergebnis eintragen");
    buttonOK.addActionListener(this);
    buttonOK.setBounds(130, 190, 200, 20);
cp.add(buttonOK);
 }

@Override
public void actionPerformed(ActionEvent e) {

        heim = (String)auswahlHeim.getSelectedItem();
gast = (String)auswahlGast.getSelectedItem();
spielerHeim = (String)auswahlspielerHeim.getSelectedItem();
spielerGast = (String)auswahlspielerGast.getSelectedItem();
toreHeim = (String)auswahlToreHeim.getSelectedItem();
toreGast = (String)auswahlToreGast.getSelectedItem();

        if (heim != gast && heim != "" && gast != "")
        {
                int spielerHeim1 = Integer.parseInt(spielerHeim);
                int spielerGast1 = Integer.parseInt(spielerGast);
                int toreHeim1 = Integer.parseInt(toreHeim);
                int toreGast1 = Integer.parseInt(toreGast);

                Ballmanager.Spielarray[Ballmanager.spielIndex] = new
                    Spiel();
```

```
                        Ballmanager.Spielarray[Ballmanager.spielIndex].
                            mannschaftsNamen(heim, gast);
                        Ballmanager.Spielarray[Ballmanager.spielIndex].spieler
                            (spielerHeim1, spielerGast1);
                        Ballmanager.Spielarray[Ballmanager.spielIndex].
                            ergebnis(toreHeim1, toreGast1);
                        Ballmanager.spielIndex++;

                int i =0;
                while (i<Ballmanager.spielIndex)
                {
                        Ballmanager.rowData[i][0] = Ballmanager.Spielarray[i].
                            heim;
                        Ballmanager.rowData[i][1] = Ballmanager.Spielarray[i].
                            gast;
                        Ballmanager.rowData[i][2] = ""+Ballmanager.Spielarray[
                            i].toreHeim;
                        Ballmanager.rowData[i][3] = ""+Ballmanager.Spielarray[
                            i].toreGast;
                        Ballmanager.rowData[i][4] = ""+Ballmanager.Spielarray[
                            i].spielerHeim;
                        Ballmanager.rowData[i][5] = ""+Ballmanager.Spielarray[
                            i].spielerGast;
                        Ballmanager.rowData[i][6] = ""+Ballmanager.Spielarray[
                            i].punkteHeim;
                        Ballmanager.rowData[i][7] = ""+Ballmanager.Spielarray[
                            i].punkteGast;
                        i++;
                }

                        Oberflaeche.table.repaint();

                        try {                   //IOException-Handler fuer
                            SpieleSpeichern
                                new SpieleSpeichern();
                                }
                        catch (IOException e1)
                                {
                                        // TODO Auto-generated catch block
                                        System.out.println("Schreiben nicht
                                            moeglich.");
                                }
                        this.dispose();
                }
                else
                {
                        //Meldung: zwei verschiedene Mannschaften waehlen!!
                }
        }
}

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

 public class Oberflaeche extends JFrame implements ActionListener{

        Container cp;
        JButton button_neueMannschaft;
 JButton button_neuesSpiel;
 static JTable table;

 public Oberflaeche(String titel) //Konstruktor
 {
        super(titel);
                            //uebergibt Fenstertitel an den
            ensprechenden JFrame-Konstruktor.
        setDefaultCloseOperation(EXIT_ON_CLOSE);        //Beendet
            Programm beim Schliessen des Fensters.
        setSize(900, 700);
    setVisible(true);
                //Macht Fenster sichtbar
    cp = getContentPane();                       //Container, um
        Festnerinhalt aufnehmen zu koennen.
    cp.setLayout(null);

    button_neueMannschaft = new JButton("Mannschaft hinzufuegen");
    button_neueMannschaft.addActionListener(this);
    button_neueMannschaft.setBounds(20, 20, 200, 20);
cp.add(button_neueMannschaft);
    button_neuesSpiel = new JButton("Spielergebnis eintragen");
    button_neuesSpiel.addActionListener(this);
    button_neuesSpiel.setBounds(240, 20, 200, 20);
cp.add(button_neuesSpiel);

int i =0;
while (i<Ballmanager.spielIndex)
{
        Ballmanager.rowData[i][0] = Ballmanager.Spielarray[i].heim;
        Ballmanager.rowData[i][1] = Ballmanager.Spielarray[i].gast;
        Ballmanager.rowData[i][2] = ""+Ballmanager.Spielarray[i].
            toreHeim;
        Ballmanager.rowData[i][3] = ""+Ballmanager.Spielarray[i].
            toreGast;
        Ballmanager.rowData[i][4] = ""+Ballmanager.Spielarray[i].
            spielerHeim;
        Ballmanager.rowData[i][5] = ""+Ballmanager.Spielarray[i].
            spielerGast;
        Ballmanager.rowData[i][6] = ""+Ballmanager.Spielarray[i].
            punkteHeim;
        Ballmanager.rowData[i][7] = ""+Ballmanager.Spielarray[i].
            punkteGast;
        i++;
}

            String[] columnNames = {
                "Heim", "Gast", "Tore Heim", "Tore Gast", "Spieler
                    Heim", "Spieler Gast", "PunkteHeim", "PunkteGast"
            };

         table = new JTable(Ballmanager.rowData, columnNames);
          JScrollPane table_pane = new JScrollPane(table);
          table_pane.setBounds(20, 60, 850, 570);
          cp.add(table_pane);

 }

@Override
```

```
        public void actionPerformed(ActionEvent arg0) { //Event-listener fuer
            die Buttons
                Object obj = arg0.getSource();

                if (obj == button_neueMannschaft) new neueMannschaft("
                    Mannschaft registrieren");
        if (obj == button_neuesSpiel && Ballmanager.mannschaftsArray[0] !=
            null && Ballmanager.mannschaftsArray[1] != null)
        {
                new neuesSpiel("Spielergebnis eintragen");
        }
        else
        {
                //Fenster oeffnen, Meldung: erst zweio Mannschaften
                    hinzufuegen!!!
        }
        }
 }

public class Spiel {

        /**
         * Teilnehmernummer 2
         * Spiel mit Spielnummer, zwei Mannschaften und einem Ergebnis jeweils
              als Attribut
         */

        public String heim;
        public String gast;
        public int spielerHeim;
        public int spielerGast;
        public int toreHeim;
        public int toreGast;
        public int punkteHeim;
        public int punkteGast;

        public void mannschaftsNamen(String heim, String gast)
        {
                this.heim = heim;
                this.gast = gast;
        }
        public void spieler(int spielerHeim, int spielerGast)
        {
                this.spielerHeim = spielerHeim;
                this.spielerGast = spielerGast;
        }

        public void ergebnis(int toreHeim, int toreGast)
        {
                this.toreHeim = toreHeim;
                this.toreGast = toreGast;
                if (toreHeim > toreGast)
                {
                        this.punkteHeim = 3;
                        this.punkteGast = 0;
                }
                if (toreHeim == toreGast)
                {
                        this.punkteHeim = 1;
                        this.punkteGast = 1;
                }
                if (toreHeim < toreGast)
                {
                        this.punkteHeim = 0;
                        this.punkteGast = 3;
                }

        }
}

import java.io.*;
import java.util.StringTokenizer;

public class SpieleLaden {


        public SpieleLaden() throws IOException
        {
             byte zeichen;
             String text ="";
             String dateiName = "spiele.gme";
             FileInputStream leseStrom = new FileInputStream(dateiName);
             do{
               zeichen = (byte)leseStrom.read();
               text += (char)zeichen;
             } while (zeichen !=-1);


             //System.out.println(text);
             StringTokenizer tokenizer = new StringTokenizer( text, "$" );

             int i = tokenizer.countTokens();
             for (int a=0; a<(i-1); a++ )
             {
                String aktuellerToken = tokenizer.nextToken();
                 StringTokenizer tokenizer1 = new StringTokenizer(
                     aktuellerToken, "&" );
                 Ballmanager.Spielarray[a] = new Spiel();
                 Ballmanager.Spielarray[a].heim = tokenizer1.nextToken();
                 Ballmanager.Spielarray[a].gast = tokenizer1.nextToken();
                 Ballmanager.Spielarray[a].toreHeim = Integer.parseInt(
                     tokenizer1.nextToken());
                 Ballmanager.Spielarray[a].toreGast = Integer.parseInt(
                     tokenizer1.nextToken());
                 Ballmanager.Spielarray[a].spielerHeim = Integer.parseInt(
                     tokenizer1.nextToken());
                 Ballmanager.Spielarray[a].spielerGast = Integer.parseInt(
                     tokenizer1.nextToken());

                 Ballmanager.spielIndex++;

                Ballmanager.rowData[a][0] = Ballmanager.Spielarray[a].heim;
                Ballmanager.rowData[a][1] = Ballmanager.Spielarray[a].gast;
                Ballmanager.rowData[a][2] = ""+Ballmanager.Spielarray[a].
                    toreHeim;
                Ballmanager.rowData[a][3] = ""+Ballmanager.Spielarray[a].
                    toreGast;
                Ballmanager.rowData[a][4] = ""+Ballmanager.Spielarray[a].
                spielerHeim;
            Ballmanager.rowData[a][5] = ""+Ballmanager.Spielarray[a].
                spielerGast;
            Ballmanager.rowData[a][6] = ""+Ballmanager.Spielarray[a].
                punkteHeim;
            Ballmanager.rowData[a][7] = ""+Ballmanager.Spielarray[a].
                punkteGast;
             /*
                     Ballmanager.Spielarray[a].heim = tokenizer1.
                         nextToken();
                     Ballmanager.Spielarray[a].gast = tokenizer1.
                         nextToken();
                     Ballmanager.Spielarray[a].toreHeim = Integer.
                         parseInt(tokenizer1.nextToken());;
                     Ballmanager.Spielarray[a].toreGast = Integer.
                         parseInt(tokenizer1.nextToken());
                     Ballmanager.Spielarray[a].spielerHeim = Integer.
                         parseInt(tokenizer1.nextToken());
                     Ballmanager.Spielarray[a].spielerGast = Integer.
                         parseInt(tokenizer1.nextToken());
             */
         }
         leseStrom.close();
         Oberflaeche.table.repaint();

     }
}

import java.io.*;


public class SpieleSpeichern {

        public SpieleSpeichern() throws IOException
        {
                FileOutputStream schreibeStrom = new FileOutputStream("spiele.
                    gme");


            for (int i=0; i < Ballmanager.spielIndex; i++)
            {
                String text = Ballmanager.Spielarray[i].heim+"&";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }

                    text = Ballmanager.Spielarray[i].gast+"&";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }

                    text = Ballmanager.Spielarray[i].toreHeim+"&";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }


                    text = String.valueOf(Ballmanager.Spielarray[i].toreGast)+
                        "&";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }

                    text = String.valueOf(Ballmanager.Spielarray[i].
                        spielerHeim)+"&";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }

                    text = String.valueOf(Ballmanager.Spielarray[i].
                        spielerGast)+"$";
                    for (int j=0; j < text.length(); j++){
                        schreibeStrom.write((byte)text.charAt(j));
                    }
            }
            schreibeStrom.close();
        }
}
```

B.6.3 Example for The Kniffel Task

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package kniffel;

import java.util.Random;

/**
 *
 * @author vorpro
 * Nummer 6
 */
public class Dice {

    private Random random;
    private int lastNumber;
    private boolean active; // Nicht zur Seite gelegt

    public Dice() {
        random = new Random();
        lastNumber = 0;
        active = true;
    }

    /**
     * Erzeugt neue Augenzahlen
     */
    public void random() {
        if (active)
            lastNumber = random.nextInt(6)+1;
    }

    /**
     * @return the lastNumber
     */
    public int getLastNumber() {
        return lastNumber;
    }

    /**
     * @return the active
     */
    public boolean isActive() {
        return active;
    }

    /**
     * @param active the active to set
     */
    public void setActive(boolean active) {
        this.active = active;
    }
}

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package kniffel;

import java.util.ArrayList;

/**
 *
 * @author vorpro
 * Nummer 6
 */
public class DiceRound {

    private Dice[] dices;
    private final int countDices = 5;
    private int playCount;
    private final int maxPlayCount = 3;

    public DiceRound() {
        dices = new Dice[countDices];
        for (int i = 0; i < countDices; i++) {
            dices[i] = new Dice();
        }
    }

    /**
     * Laesst alle Wuerfel ihre Augenzahlen neu berechnen
     * @throws Exception    Wenn bereits zuviele Wuerfe durchgefuehrt wurden
     */
    public void newDice() throws Exception {
        if (getPlayCount() >= getMaxPlayCount()) {
            throw new Exception("Bereits zu viele Wuerfe");
        }

        for (int i = 0; i < dices.length; i++) {
            dices[i].random();
        }

        playCount++;
    }

    /**
     * Gibt alle Augenzahlen aller Wuerfel aus
     * @return
     */
    public int[] getResult() {
        int[] arr = new int[countDices];

        for (int i = 0; i < dices.length; i++) {
            arr[i] = dices[i].getLastNumber();
        }

        return arr;
    }

    /**
     * Summiert die Augenzahlen auf
     * @return result Summe der Augenzahlen
     */
    public int getSumResult() {
        int result = 0;

        for (int i = 0; i < dices.length; i++) {
            result += dices[i].getLastNumber();
        }

        return result;
    }

    /**
     * Zum Entfernen von gleichen Augenzahlens
     * @return int[] ohne duplicate
     */
    public int[] getUniqueResult() {
        ArrayList<Integer> arrayList = new ArrayList<Integer>();

        int[] duplicatesArray = new int[6];

        for (int i : this.getResult()) {
            if (duplicatesArray[i - 1] == 0) {
                duplicatesArray[i - 1] = i;
                arrayList.add(i);
            }
        }

        Integer[] arr = new Integer[arrayList.size()];
        arr = arrayList.toArray(arr);
        int[] results = new int[arr.length];

        for (int i = 0; i < results.length; i++) {
            results[i] = (int) arr[i];
        }
        return results;
    }

    /**
     * Zaehlt wieoft welche Zahl vorkommt
     * @return Haeufigkeit der Zahlen
     */
    public int[] countNumbersOfDices() {
        int[] arr = new int[6];

        for (int i = 0; i < dices.length; i++) {
            arr[dices[i].getLastNumber() - 1]++;
        }

        return arr;
    }

    /**
     * Zaehlt wieoft eine bestimmte Zahl vorkommt
     * @param number Die Zahl, nach deren Haeufigkeit gesucht wird
     * @return result Haeufigkeit der geforderten Zahl
     */
    public int countNumber(int number) {
        int count = 0;
        for (int i = 0; i < dices.length; i++) {
            if (dices[i].getLastNumber() == number) {
                count++;
            }
        }
        return count;
    }

    /**
     * Sperrt einen bestimmten Wuerfel
     * @param index Der gewuenschte Wuerfel
     */
    public void saveDice(int index) {
        dices[index].setActive(false);
    }

    /**
     * @return the playCount
     */
    public int getPlayCount() {
        return playCount;
    }

    /**
     * @return the maxPlayCount
     */
    public int getMaxPlayCount() {
        return maxPlayCount;
    }
}

/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package kniffel;

import java.util.Arrays;

/**
 *
 * @author vorpro
 * Nummer 6
 *  Spiel       Punkte  Punkte  Punkte  Anmerkungen
 *  Einer               Nur Einer zaehlen            | Index 0 of saveBlock
 *  Zweier              Nur Zweier zaehlen           | Index 1 sof saveBlock
 *  Dreier              Nur Dreier zaehlen           | Index 2 of saveBlock
 *  Vierer              Nur Vierer zaehlen           | Index 3 of saveBlock
 *  Fuenfer             Nur Fuenfer zaehlen           | Index 4 of saveBlock
 *  Sechser             Nur Sechser zaehlen          | Index 5 of saveBlock
 *
 *  Bonus               35 Punkte, wenn oben mindestens 63 Punkte          |
     Index 0 of otherBlock
 *  Dreierpasch         Drei gleiche Wuerfel - Alle Augen zaehlen
                 Index 1 of otherBlock
 *  Viererpasch         Vier gleiche Wuerfel - Alle Augen zaehlen
                 Index 2 of otherBlock
 *  Full House          Drei gleiche und zwei gleiche Wuerfel - 25 Punkte
     Index 3 of otherBlock
 *  Kleine Strasse      1-2-3-4, 2-3-4-5, oder 3-4-5-6 - 30 Punkte
     Index 4 of otherBlock
 *  Grosse Strasse      1-2-3-4-5 oder 2-3-4-5-6 - 40 Punkte
     Index 5 of otherBlock
 *  Kniffel/Yahtzee     Fuenf gleiche Wuerfel - 50 Punkte
                          Index 6 of otherBlock
 *  Chance              Alle Augen zaehlen
 *  Summe unten
 * Gesamtsumme                          Oben + Bonus + Unten
 */
public class GameSheet {

    private final int maxPlayRounds = 3;
    private int actualPlayRound;
    private DiceRound[][] saveBlock;
    private DiceRound[][] otherBlock;
    private int[] insertCount;

    /**
     * Bildet die Funktionalitaet eines Spielzettels nach.
     */
    public GameSheet() {
        actualPlayRound = 0;
        saveBlock = new DiceRound[maxPlayRounds][6];
        otherBlock = new DiceRound[maxPlayRounds][7];
        insertCount = new int[maxPlayRounds];
    }

    /**
     * Neue Runde starten, ausser maximale Rundenzahl ist erreicht.
     * @throws Exception
     */
    public void newRound() throws Exception {
        if (actualPlayRound >= maxPlayRounds) {
            throw new Exception("Maximale Spielzahl erreicht");
        }
        actualPlayRound++;
    }

    /**
     * Speichert die Wuerfel Runde im Sammelblock
     * @param index         Welche Position in der Liste? (1er, 2er, 3er, ...)
     * @param diceRound     Die Wuerfelrunde
     */
    public void addDiceRoundToSaveBlock(int index, DiceRound diceRound) {
        if (saveBlock[actualPlayRound][index] == null) {
            saveBlock[actualPlayRound][index] = diceRound;
            insertCount[actualPlayRound]++;
        }
    }

    /**
     * Speichert die Wuerfel Runde im anderen Block
     * @param index         Welche Position in der Liste? (Dreierpasch,
         Viererpasch, ...)
     * @param diceRound     Die Wuerfelrunde
     */
    public void addDiceRoundToOtherBlock(int index, DiceRound diceRound) {
        if (otherBlock[actualPlayRound][index] == null) {
            otherBlock[actualPlayRound][index] = diceRound;
            insertCount[actualPlayRound]++;
        }
    }

    /**
     * Berechnet das Gesamtergebnis jeder Runde und jeden Blocks.
     * @return result   Die Summe der Spielstaende
     */
    public int calculateOverallResult() {
        int result = 0;

        // saveBlock Rounds
        for (int i = 0; i < saveBlock.length; i++) {
            // saveBlock items
            for (int j = 0; j < saveBlock[i].length; j++) {
                // Item 0 soll auf 1er achten
                result += saveBlock[i][j].countNumber(j + 1);
            }
        }
        if (result >= 63) {
            result += 35;
        }
        return result;
    }

    /**
     * Berechnet das Ergebnis des Sammelblocks fuer 1 Runde
     * @param round     Die gesuchte Runde
     * @return result   Die Summe der Runde
     */
    public int calculateCollectBoxRoundResult(int round) {
        int result = 0;

        for (int j = 0; j < saveBlock[round].length; j++) {
            // Item 0 soll auf 1er achten
            result += this.getCollectBoxRoundRowResult(round, j);
        }

        // Eventueller Bonus
        if (result >= 63) {
            result += 35;
        }

        return result;
    }

    /**
     * Berechnet das Ergebnis des anderen Blockes fuer 1 Runde
     * @param round     Die gesuchte Runde
     * @return result   Die Summe der Runde
     */
    public int calculateOtherBoxRoundResult(int round) {
        int result = 0;

        for (int j = 0; j < otherBlock[round].length; j++) {
            // Item 0 soll auf 1er achten
            result += this.getOtherBoxRoundRowResult(round, j);
        }

        return result;
```

```
}

/**
 * Berechnet das Ergebnis eines bestimmten Zeilen / Spalten Eintrages
 * @param round     Die gesuchte Runde
 * @param row       Die gesuchte Reihe (1er, 2er, 3er, ...)
 * @return result   Das Summenergebnis
 */
public int getCollectBoxRoundRowResult(int round, int row) {
    int result = 0;
    try {
        result = saveBlock[round][row].countNumber(row + 1);
    } catch (NullPointerException ex) {
        result = 0;
    }
    return result;
}

/**
 * Berechnet das Ergebnis eines bestimmten Zeilen / Spalten Eintrages
 * @param round     Die gesuchte Runde
 * @param row       Die gesuchte Reihe (1er, 2er, 3er, ...)
 * @return result   Das Summenergebnis
 */
public int getOtherBoxRoundRowResult(int round, int row) {
    int result = 0;
    int[] counts;
    int[] results;
    try {
        switch (row) {
            // Dreierpasch
            case 0:
                counts = otherBlock[round][row].countNumbersOfDices();
                for (int i : counts) {
                    if (i >= 3) {
                        result = otherBlock[round][row].getSumResult();
                    }
                }
                break;
            // Viererpasch
            case 1:
                counts = otherBlock[round][row].countNumbersOfDices();
                for (int i : counts) {
                    if (i >= 3) {
                        result = otherBlock[round][row].getSumResult();
                    }
                }
                break;

            // Full House
            case 2:
                counts = otherBlock[round][row].countNumbersOfDices();
                boolean check = false;
                for (int i : counts) {
                    if (i == 3 || i == 2) {
                        check = true;
                    }

                    if (check && (i == 3 || i == 2)) {
                        result = 25;
                    }
                }
                break;
            // Kleine Strasse
            case 3:
                results = otherBlock[round][row].getUniqueResult();
                Arrays.sort(results);

                if (results.length < 4) {
                    result = 0;
                    break;
                }
                int alastNumber = results[0];
                for (int i = results[1]; i < results[0] + 3; i++) {
                    if (alastNumber - 1 != i) {
                        result = 0;
                        break;
                    }
                    alastNumber++;
                }
                result = 30;
                break;
            // Kleine Strasse
            case 4:
                results = otherBlock[round][row].getUniqueResult();
                Arrays.sort(results);

                if (results.length < 5) {
                    result = 0;
                    break;
                }
                int blastNumber = results[0];
                for (int i = results[1]; i < results[0] + 4; i++) {
                    if (blastNumber - 1 != i) {
                        result = 0;
                        break;
                    }
                    blastNumber++;
                }
                result = 40;
                break;
            // Kniffel
            case 5:
                results = otherBlock[round][row].getUniqueResult();
                if (results.length == 1) {
                    result = 50;
                }
                result = 0;
                break;
            // Chance
            case 6:
                result = otherBlock[round][row].getSumResult();
                break;
        }
    } catch (NullPointerException ex) {
        // Sollte ein Eintrag noch nicht gesetzt sein, wird eine
            Nullpointer geworfen und
        // 0 zurueckgegeben
```

```
            return 0;
        }
        return result;
    }

    /**
     * Gibt an, wieviele Felder in der aktuellen Runde bereits gesetzt wurden.
     * @return the insertCount
     */
    public int getInsertCount() {
        return insertCount[actualPlayRound];
    }
}

/*
 * KniffelAboutBox.java
 */

package kniffel;

import org.jdesktop.application.Action;

public class KniffelAboutBox extends javax.swing.JDialog {

    public KniffelAboutBox(java.awt.Frame parent) {
        super(parent);
        initComponents();
        getRootPane().setDefaultButton(closeButton);
    }

    @Action public void closeAboutBox() {
        dispose();
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN
        :initComponents
    private void initComponents() {

        closeButton = new javax.swing.JButton();
        javax.swing.JLabel appTitleLabel = new javax.swing.JLabel();
        javax.swing.JLabel versionLabel = new javax.swing.JLabel();
        javax.swing.JLabel appVersionLabel = new javax.swing.JLabel();
        javax.swing.JLabel vendorLabel = new javax.swing.JLabel();
        javax.swing.JLabel appVendorLabel = new javax.swing.JLabel();
        javax.swing.JLabel homepageLabel = new javax.swing.JLabel();
        javax.swing.JLabel appHomepageLabel = new javax.swing.JLabel();
        javax.swing.JLabel appDescLabel = new javax.swing.JLabel();
        javax.swing.JLabel imageLabel = new javax.swing.JLabel();

        setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE)
            ;
        org.jdesktop.application.ResourceMap resourceMap = org.jdesktop.
            application.Application.getInstance(kniffel.KniffelApp.class).
            getContext().getResourceMap(KniffelAboutBox.class);
```

```
setTitle(resourceMap.getString("title")); // NOI18N
setModal(true);
setName("aboutBox"); // NOI18N
setResizable(false);

javax.swing.ActionMap actionMap = org.jdesktop.application.Application
    .getInstance(kniffel.KniffelApp.class).getContext().getActionMap(
    KniffelAboutBox.class, this);
closeButton.setAction(actionMap.get("closeAboutBox")); // NOI18N
closeButton.setName("closeButton"); // NOI18N

appTitleLabel.setFont(appTitleLabel.getFont().deriveFont(appTitleLabel
    .getFont().getStyle() | java.awt.Font.BOLD, appTitleLabel.getFont
    ().getSize()+4));
appTitleLabel.setText(resourceMap.getString("Application.title")); //
    NOI18N
appTitleLabel.setName("appTitleLabel"); // NOI18N

versionLabel.setFont(versionLabel.getFont().deriveFont(versionLabel.
    getFont().getStyle() | java.awt.Font.BOLD));
versionLabel.setText(resourceMap.getString("versionLabel.text")); //
    NOI18N
versionLabel.setName("versionLabel"); // NOI18N

appVersionLabel.setText(resourceMap.getString("Application.version"));
     // NOI18N
appVersionLabel.setName("appVersionLabel"); // NOI18N

vendorLabel.setFont(vendorLabel.getFont().deriveFont(vendorLabel.
    getFont().getStyle() | java.awt.Font.BOLD));
vendorLabel.setText(resourceMap.getString("vendorLabel.text")); //
    NOI18N
vendorLabel.setName("vendorLabel"); // NOI18N

appVendorLabel.setText(resourceMap.getString("Application.vendor"));
    // NOI18N
appVendorLabel.setName("appVendorLabel"); // NOI18N

homepageLabel.setFont(homepageLabel.getFont().deriveFont(homepageLabel
    .getFont().getStyle() | java.awt.Font.BOLD));
homepageLabel.setText(resourceMap.getString("homepageLabel.text")); //
     NOI18N
homepageLabel.setName("homepageLabel"); // NOI18N

appHomepageLabel.setText(resourceMap.getString("Application.homepage")
    ); // NOI18N
appHomepageLabel.setName("appHomepageLabel"); // NOI18N

appDescLabel.setText(resourceMap.getString("appDescLabel.text")); //
    NOI18N
appDescLabel.setName("appDescLabel"); // NOI18N

imageLabel.setIcon(resourceMap.getIcon("imageLabel.icon")); // NOI18N
imageLabel.setName("imageLabel"); // NOI18N

javax.swing.GroupLayout layout = new javax.swing.GroupLayout(
    getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
        LEADING)
    .addGroup(layout.createSequentialGroup()
        .addComponent(imageLabel)
        .addGap(18, 18, 18)
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
            Alignment.TRAILING)
            .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
                layout.createSequentialGroup()
                .addGroup(layout.createParallelGroup(javax.swing.
                    GroupLayout.Alignment.LEADING)
                    .addComponent(versionLabel)
                    .addComponent(vendorLabel)
                    .addComponent(homepageLabel))
                .addPreferredGap(javax.swing.LayoutStyle.
                    ComponentPlacement.RELATED)
                .addGroup(layout.createParallelGroup(javax.swing.
                    GroupLayout.Alignment.LEADING)
                    .addComponent(appVersionLabel)
                    .addComponent(appVendorLabel)
                    .addComponent(appHomepageLabel)))
            .addComponent(appTitleLabel, javax.swing.GroupLayout.
                Alignment.LEADING)
            .addComponent(appDescLabel, javax.swing.GroupLayout.
                Alignment.LEADING, javax.swing.GroupLayout.
                DEFAULT_SIZE, 266, Short.MAX_VALUE)
            .addComponent(closeButton))
        .addContainerGap())
);
layout.setVerticalGroup(
    layout.createParallelGroup(javax.swing.GroupLayout.Alignment.
        LEADING)
    .addComponent(imageLabel, javax.swing.GroupLayout.PREFERRED_SIZE,
        javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
        .addComponent(appTitleLabel)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(appDescLabel, javax.swing.GroupLayout.
            PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
            javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
            Alignment.BASELINE)
            .addComponent(versionLabel)
            .addComponent(appVersionLabel))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
            Alignment.BASELINE)
            .addComponent(vendorLabel)
            .addComponent(appVendorLabel))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addGroup(layout.createParallelGroup(javax.swing.GroupLayout.
            Alignment.BASELINE)
            .addComponent(homepageLabel)
            .addComponent(appHomepageLabel))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED, 19, Short.MAX_VALUE)
        .addComponent(closeButton)
        .addContainerGap())
);

pack();
}// </editor-fold>//GEN-END:initComponents

// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JButton closeButton;
// End of variables declaration//GEN-END:variables

}

/*
* KniffelApp.java
*/

package kniffel;

import org.jdesktop.application.Application;
import org.jdesktop.application.SingleFrameApplication;

/**
* The main class of the application.
* Number 6
*/
public class KniffelApp extends SingleFrameApplication {

/**
* At startup create and show the main frame of the application.
*/
@Override
protected void startup() {
show(new KniffelView(this));
}

/**
* This method is to initialize the specified window by injecting
 resources.
* Windows shown in our application come fully initialized from the GUI
* builder, so this additional configuration is not needed.
*/
@Override
protected void configureWindow(java.awt.Window root) {
}

/**
* A convenient static getter for the application instance.
* @return the instance of KniffelApp
*/
public static KniffelApp getApplication() {
return Application.getInstance(KniffelApp.class);
}

/**
* Main method launching the application.
     */
    public static void main(String[] args) {
        launch(KniffelApp.class, args);
    }
}

/*
 * KniffelView.java
 */
package kniffel;

import java.util.logging.Level;
import java.util.logging.Logger;
import org.jdesktop.application.Action;
import org.jdesktop.application.ResourceMap;
import org.jdesktop.application.SingleFrameApplication;
import org.jdesktop.application.FrameView;
import org.jdesktop.application.TaskMonitor;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.Timer;
import javax.swing.Icon;
import javax.swing.JDialog;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JToggleButton;

/**
 * The application's main frame.
 * Nummer 6
 */
public class KniffelView extends FrameView {

    private Player player;
    private GameSheet lastGameSheet;
    private DiceRound actualDiceRound;

    public KniffelView(SingleFrameApplication app) {
        super(app);

        initComponents();

// <editor-fold defaultstate="collapsed" desc="generated code">
// status bar initialization - message timeout, idle icon and busy animation,
    etc
        ResourceMap resourceMap = getResourceMap();
        int messageTimeout = resourceMap.getInteger("StatusBar.messageTimeout"
            );
        messageTimer = new Timer(messageTimeout, new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                statusMessageLabel.setText("");
            }
        });
        messageTimer.setRepeats(false);
        int busyAnimationRate = resourceMap.getInteger("StatusBar.
            busyAnimationRate");
        for (int i = 0; i < busyIcons.length; i++) {
            busyIcons[i] = resourceMap.getIcon("StatusBar.busyIcons[" + i + "]
                ");
        }
        busyIconTimer = new Timer(busyAnimationRate, new ActionListener() {

            public void actionPerformed(ActionEvent e) {
                busyIconIndex = (busyIconIndex + 1) % busyIcons.length;
                statusAnimationLabel.setIcon(busyIcons[busyIconIndex]);
            }
        });
        idleIcon = resourceMap.getIcon("StatusBar.idleIcon");
        statusAnimationLabel.setIcon(idleIcon);
        progressBar.setVisible(false);

        // connecting action tasks to status bar via TaskMonitor
        TaskMonitor taskMonitor = new TaskMonitor(getApplication().getContext
            ());
        taskMonitor.addPropertyChangeListener(new java.beans.
            PropertyChangeListener() {

            public void propertyChange(java.beans.PropertyChangeEvent evt) {
                String propertyName = evt.getPropertyName();
                if ("started".equals(propertyName)) {
                    if (!busyIconTimer.isRunning()) {
                        statusAnimationLabel.setIcon(busyIcons[0]);
                        busyIconIndex = 0;
                        busyIconTimer.start();
                    }
                    progressBar.setVisible(true);
                    progressBar.setIndeterminate(true);
                } else if ("done".equals(propertyName)) {
                    busyIconTimer.stop();
                    statusAnimationLabel.setIcon(idleIcon);
                    progressBar.setVisible(false);
                    progressBar.setValue(0);
                } else if ("message".equals(propertyName)) {
                    String text = (String) (evt.getNewValue());
                    statusMessageLabel.setText((text == null) ? "" : text);
                    messageTimer.restart();
                } else if ("progress".equals(propertyName)) {
                    int value = (Integer) (evt.getNewValue());
                    progressBar.setVisible(true);
                    progressBar.setIndeterminate(false);
                    progressBar.setValue(value);
                }
            }
        });// </editor-fold>

        player = new Player();
        player.addGameSheet();
        lastGameSheet = player.getLastGameSheet();
        actualDiceRound = new DiceRound();
        this.updateTable();
    }

    @Action
    public void showAboutBox() {
        if (aboutBox == null) {
            JFrame mainFrame = KniffelApp.getApplication().getMainFrame();
            aboutBox = new KniffelAboutBox(mainFrame);
```

```
            aboutBox.setLocationRelativeTo(mainFrame);
        }
        KniffelApp.getApplication().show(aboutBox);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-BEGIN
        :initComponents
    private void initComponents() {

        mainPanel = new javax.swing.JPanel();
        dicePanel = new javax.swing.JPanel();
        countLabel = new javax.swing.JLabel();
        dice1 = new javax.swing.JToggleButton();
        dice3 = new javax.swing.JToggleButton();
        dice2 = new javax.swing.JToggleButton();
        dice4 = new javax.swing.JToggleButton();
        dice5 = new javax.swing.JToggleButton();
        playButton = new javax.swing.JButton();
        controlPanel = new javax.swing.JPanel();
        jLabel1 = new javax.swing.JLabel();
        sammelblockSelectBox = new javax.swing.JComboBox();
        jButton7 = new javax.swing.JButton();
        jLabel3 = new javax.swing.JLabel();
        otherBlockSelectBox = new javax.swing.JComboBox();
        jScrollPane1 = new javax.swing.JScrollPane();
        collectTable = new javax.swing.JTable();
        jScrollPane2 = new javax.swing.JScrollPane();
        otherTable = new javax.swing.JTable();
        menuBar = new javax.swing.JMenuBar();
        javax.swing.JMenu fileMenu = new javax.swing.JMenu();
        jMenuItem1 = new javax.swing.JMenuItem();
        javax.swing.JMenuItem exitMenuItem = new javax.swing.JMenuItem();
        javax.swing.JMenu helpMenu = new javax.swing.JMenu();
        javax.swing.JMenuItem aboutMenuItem = new javax.swing.JMenuItem();
        statusPanel = new javax.swing.JPanel();
        javax.swing.JSeparator statusPanelSeparator = new javax.swing.
            JSeparator();
        statusMessageLabel = new javax.swing.JLabel();
        statusAnimationLabel = new javax.swing.JLabel();
        progressBar = new javax.swing.JProgressBar();

        mainPanel.setName("mainPanel"); // NOI18N

        org.jdesktop.application.ResourceMap resourceMap = org.jdesktop.
            application.Application.getInstance(kniffel.KniffelApp.class).
            getContext().getResourceMap(KniffelView.class);
        dicePanel.setBorder(javax.swing.BorderFactory.createTitledBorder(javax
            .swing.BorderFactory.createTitledBorder(resourceMap.getString("
            dicePanel.border.border.title")))); // NOI18N
        dicePanel.setName("dicePanel"); // NOI18N

        countLabel.setHorizontalAlignment(javax.swing.SwingConstants.CENTER);
        countLabel.setText(resourceMap.getString("countLabel.text")); //
            NOI18N
        countLabel.setName("countLabel"); // NOI18N

        dice1.setMnemonic('1');
        dice1.setText(resourceMap.getString("dice1.text")); // NOI18N
        dice1.setFocusable(false);
        dice1.setName("dice1"); // NOI18N
        dice1.setRolloverEnabled(false);
        dice1.setSelectedIcon(resourceMap.getIcon("dice1.selectedIcon")); //
            NOI18N
        dice1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                dice1ActionPerformed(evt);
            }
        });

        dice3.setMnemonic('3');
        dice3.setText(resourceMap.getString("dice3.text")); // NOI18N
        dice3.setCursor(new java.awt.Cursor(java.awt.Cursor.DEFAULT_CURSOR));
        dice3.setFocusable(false);
        dice3.setName("dice3"); // NOI18N
        dice3.setRolloverEnabled(false);
        dice3.setSelectedIcon(resourceMap.getIcon("dice3.selectedIcon")); //
            NOI18N
        dice3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                dice3ActionPerformed(evt);
            }
        });

        dice2.setMnemonic('2');
        dice2.setText(resourceMap.getString("dice2.text")); // NOI18N
        dice2.setFocusable(false);
        dice2.setName("dice2"); // NOI18N
        dice2.setRolloverEnabled(false);
        dice2.setSelectedIcon(resourceMap.getIcon("dice2.selectedIcon")); //
            NOI18N
        dice2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                dice2ActionPerformed(evt);
            }
        });

        dice4.setMnemonic('4');
        dice4.setText(resourceMap.getString("dice4.text")); // NOI18N
        dice4.setFocusable(false);
        dice4.setName("dice4"); // NOI18N
        dice4.setRolloverEnabled(false);
        dice4.setSelectedIcon(resourceMap.getIcon("dice4.selectedIcon")); //
            NOI18N
        dice4.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                dice4ActionPerformed(evt);
            }
        });

        dice5.setMnemonic('5');
        dice5.setText(resourceMap.getString("dice5.text")); // NOI18N
        dice5.setFocusable(false);
```

```
dice5.setName("dice5"); // NOI18N
dice5.setRolloverEnabled(false);
dice5.setSelectedIcon(resourceMap.getIcon("dice5.selectedIcon")); //
    NOI18N
dice5.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        dice5ActionPerformed(evt);
    }
});

playButton.setMnemonic('W');
playButton.setText(resourceMap.getString("playButton.text")); //
    NOI18N
playButton.setName("playButton"); // NOI18N
playButton.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        playButtonActionPerformed(evt);
    }
});

javax.swing.GroupLayout dicePanelLayout = new javax.swing.GroupLayout(
    dicePanel);
dicePanel.setLayout(dicePanelLayout);
dicePanelLayout.setHorizontalGroup(
    dicePanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addComponent(countLabel, javax.swing.GroupLayout.DEFAULT_SIZE,
        100, Short.MAX_VALUE)
    .addGroup(dicePanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(dicePanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.LEADING)
            .addComponent(dice1, javax.swing.GroupLayout.Alignment.
                TRAILING, javax.swing.GroupLayout.PREFERRED_SIZE, 80,
                javax.swing.GroupLayout.PREFERRED_SIZE)
            .addComponent(dice2, javax.swing.GroupLayout.Alignment.
                TRAILING, javax.swing.GroupLayout.PREFERRED_SIZE, 80,
                javax.swing.GroupLayout.PREFERRED_SIZE)
            .addComponent(dice3, javax.swing.GroupLayout.Alignment.
                TRAILING, javax.swing.GroupLayout.PREFERRED_SIZE, 80,
                javax.swing.GroupLayout.PREFERRED_SIZE)
            .addComponent(dice4, javax.swing.GroupLayout.Alignment.
                TRAILING, javax.swing.GroupLayout.PREFERRED_SIZE, 80,
                javax.swing.GroupLayout.PREFERRED_SIZE)
            .addComponent(dice5, javax.swing.GroupLayout.Alignment.
                TRAILING, javax.swing.GroupLayout.PREFERRED_SIZE, 80,
                javax.swing.GroupLayout.PREFERRED_SIZE))
        .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.
            MAX_VALUE))
    .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
        dicePanelLayout.createSequentialGroup()
        .addContainerGap()
        .addComponent(playButton, javax.swing.GroupLayout.DEFAULT_SIZE
            , 80, Short.MAX_VALUE)
        .addContainerGap())
);
dicePanelLayout.setVerticalGroup(
    dicePanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addGroup(dicePanelLayout.createSequentialGroup()
        .addComponent(countLabel)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(playButton)
        .addGap(18, 18, 18)
        .addComponent(dice1, javax.swing.GroupLayout.PREFERRED_SIZE,
            65, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(dice2, javax.swing.GroupLayout.PREFERRED_SIZE,
            65, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(dice3, javax.swing.GroupLayout.PREFERRED_SIZE,
            65, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(dice4, javax.swing.GroupLayout.PREFERRED_SIZE,
            65, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(dice5, javax.swing.GroupLayout.PREFERRED_SIZE,
            65, javax.swing.GroupLayout.PREFERRED_SIZE)
        .addContainerGap(javax.swing.GroupLayout.DEFAULT_SIZE, Short.
            MAX_VALUE))
);

controlPanel.setBorder(javax.swing.BorderFactory.createTitledBorder(
    resourceMap.getString("controlPanel.border.title"))); // NOI18N
controlPanel.setName("controlPanel"); // NOI18N

jLabel1.setText(resourceMap.getString("jLabel1.text")); // NOI18N
jLabel1.setName("jLabel1"); // NOI18N
jLabel1.setNextFocusableComponent(sammelblockSelectBox);

sammelblockSelectBox.setModel(new javax.swing.DefaultComboBoxModel(new
     String[] { "None", "1er", "2er", "3er", "4er", "5er", "6er" }));
sammelblockSelectBox.setName("sammelblockSelectBox"); // NOI18N

jButton7.setMnemonic('E');
jButton7.setText(resourceMap.getString("jButton7.text")); // NOI18N
jButton7.setName("jButton7"); // NOI18N
jButton7.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jButton7ActionPerformed(evt);
    }
});

jLabel3.setText(resourceMap.getString("jLabel3.text")); // NOI18N
jLabel3.setName("jLabel3"); // NOI18N

otherBlockSelectBox.setModel(new javax.swing.DefaultComboBoxModel(new
    String[] { "None", "Dreierpasch", "Viererpasch", "Full House", "
    Kleine Strasse", "Grosse Strasse", "Kniffel", "Chance" }));
otherBlockSelectBox.setName("otherBlockSelectBox"); // NOI18N

javax.swing.GroupLayout controlPanelLayout = new javax.swing.
    GroupLayout(controlPanel);
```

```
controlPanel.setLayout(controlPanelLayout);
controlPanelLayout.setHorizontalGroup(
    controlPanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addGroup(controlPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(controlPanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.LEADING)
            .addGroup(controlPanelLayout.createSequentialGroup()
                .addGroup(controlPanelLayout.createParallelGroup(javax
                    .swing.GroupLayout.Alignment.LEADING)
                    .addComponent(jLabel1)
                    .addComponent(jLabel3))
                .addPreferredGap(javax.swing.LayoutStyle.
                    ComponentPlacement.UNRELATED)
                .addGroup(controlPanelLayout.createParallelGroup(javax
                    .swing.GroupLayout.Alignment.LEADING, false)
                    .addComponent(otherBlockSelectBox, 0, javax.swing.
                        GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                    .addComponent(sammelblockSelectBox, 0, 75, Short.
                        MAX_VALUE)))
            .addComponent(jButton7, javax.swing.GroupLayout.Alignment.
                TRAILING))
        .addContainerGap())
);
controlPanelLayout.setVerticalGroup(
    controlPanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addGroup(controlPanelLayout.createSequentialGroup()
        .addGap(19, 19, 19)
        .addGroup(controlPanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel1)
            .addComponent(sammelblockSelectBox, javax.swing.
                GroupLayout.PREFERRED_SIZE, javax.swing.GroupLayout.
                DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            UNRELATED)
        .addGroup(controlPanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel3)
            .addComponent(otherBlockSelectBox, javax.swing.GroupLayout
                .PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
                 javax.swing.GroupLayout.PREFERRED_SIZE))
        .addGap(25, 25, 25)
        .addComponent(jButton7)
        .addContainerGap(297, Short.MAX_VALUE))
);

jScrollPane1.setName("jScrollPane1"); // NOI18N

collectTable.setModel(new javax.swing.table.DefaultTableModel(
    new Object [][] {
        {"1er", null, null, null},
        {"2er", null, null, null},
        {"3er", null, null, null},
        {"4er", null, null, null},
        {"5er", null, null, null},
        {"6er", null, null, null},
        {"Summe", null, null, null}
    },
    new String [] {
        "Spiel", "Punkte", "Punkte", "Punkte"
    }
) {
    Class[] types = new Class [] {
        java.lang.String.class, java.lang.Integer.class, java.lang.
            Integer.class, java.lang.Integer.class
    };
    boolean[] canEdit = new boolean [] {
        false, false, false, false
    };

    public Class getColumnClass(int columnIndex) {
        return types [columnIndex];
    }

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return canEdit [columnIndex];
    }
});
collectTable.setName("collectTable"); // NOI18N
jScrollPane1.setViewportView(collectTable);

jScrollPane2.setName("jScrollPane2"); // NOI18N

otherTable.setModel(new javax.swing.table.DefaultTableModel(
    new Object [][] {
        {"Dreierpasch", null, null, null},
        {"Viererpasch", null, null, null},
        {"Full House", null, null, null},
        {"Kleine Strasse", null, null, null},
        {"Grosse Strasse", null, null, null},
        {"Kniffel", null, null, null},
        {"Chance", null, null, null},
        {"Summe", null, null, null}
    },
    new String [] {
        "Spiel", "Punkte", "Punkte", "Punkte"
    }
) {
    Class[] types = new Class [] {
        java.lang.String.class, java.lang.Integer.class, java.lang.
            Integer.class, java.lang.Integer.class
    };
    boolean[] canEdit = new boolean [] {
        false, false, false, false
    };

    public Class getColumnClass(int columnIndex) {
        return types [columnIndex];
    }

    public boolean isCellEditable(int rowIndex, int columnIndex) {
        return canEdit [columnIndex];
    }
});
otherTable.setName("otherTable"); // NOI18N
```

```
jScrollPane2.setViewportView(otherTable);

javax.swing.GroupLayout mainPanelLayout = new javax.swing.GroupLayout(
    mainPanel);
mainPanel.setLayout(mainPanelLayout);
mainPanelLayout.setHorizontalGroup(
    mainPanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addGroup(mainPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addComponent(dicePanel, javax.swing.GroupLayout.
            PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
            javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addComponent(controlPanel, javax.swing.GroupLayout.
            PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
            javax.swing.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED)
        .addGroup(mainPanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.LEADING)
            .addComponent(jScrollPane1, javax.swing.GroupLayout.
                PREFERRED_SIZE, 412, javax.swing.GroupLayout.
                PREFERRED_SIZE)
            .addComponent(jScrollPane2, javax.swing.GroupLayout.
                PREFERRED_SIZE, 412, javax.swing.GroupLayout.
                PREFERRED_SIZE))
        .addGap(34, 34, 34))
);
mainPanelLayout.setVerticalGroup(
    mainPanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addGroup(mainPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addGroup(mainPanelLayout.createParallelGroup(javax.swing.
            GroupLayout.Alignment.LEADING)
            .addGroup(mainPanelLayout.createSequentialGroup()
                .addComponent(jScrollPane1, javax.swing.GroupLayout.
                    PREFERRED_SIZE, 139, javax.swing.GroupLayout.
                    PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.
                    ComponentPlacement.UNRELATED)
                .addComponent(jScrollPane2, javax.swing.GroupLayout.
                    PREFERRED_SIZE, 156, javax.swing.GroupLayout.
                    PREFERRED_SIZE))
            .addComponent(dicePanel, javax.swing.GroupLayout.Alignment
                .TRAILING, javax.swing.GroupLayout.DEFAULT_SIZE, javax
                .swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
            .addComponent(controlPanel, javax.swing.GroupLayout.
                DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
                Short.MAX_VALUE))
        .addContainerGap())
);

menuBar.setName("menuBar"); // NOI18N

fileMenu.setText(resourceMap.getString("fileMenu.text")); // NOI18N
fileMenu.setName("fileMenu"); // NOI18N
```

```
jMenuItem1.setAccelerator(javax.swing.KeyStroke.getKeyStroke(java.awt.
    event.KeyEvent.VK_N, java.awt.event.InputEvent.CTRL_MASK));
jMenuItem1.setText(resourceMap.getString("jMenuItem1.text")); //
    NOI18N
jMenuItem1.setName("jMenuItem1"); // NOI18N
jMenuItem1.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        jMenuItem1ActionPerformed(evt);
    }
});
fileMenu.add(jMenuItem1);

javax.swing.ActionMap actionMap = org.jdesktop.application.Application
    .getInstance(kniffel.KniffelApp.class).getContext().getActionMap(
    KniffelView.class, this);
exitMenuItem.setAction(actionMap.get("quit")); // NOI18N
exitMenuItem.setName("exitMenuItem"); // NOI18N
fileMenu.add(exitMenuItem);

menuBar.add(fileMenu);

helpMenu.setText(resourceMap.getString("helpMenu.text")); // NOI18N
helpMenu.setName("helpMenu"); // NOI18N

aboutMenuItem.setAction(actionMap.get("showAboutBox")); // NOI18N
aboutMenuItem.setName("aboutMenuItem"); // NOI18N
helpMenu.add(aboutMenuItem);

menuBar.add(helpMenu);

statusPanel.setName("statusPanel"); // NOI18N

statusPanelSeparator.setName("statusPanelSeparator"); // NOI18N

statusMessageLabel.setName("statusMessageLabel"); // NOI18N

statusAnimationLabel.setHorizontalAlignment(javax.swing.SwingConstants
    .LEFT);
statusAnimationLabel.setName("statusAnimationLabel"); // NOI18N

progressBar.setName("progressBar"); // NOI18N

javax.swing.GroupLayout statusPanelLayout = new javax.swing.
    GroupLayout(statusPanel);
statusPanel.setLayout(statusPanelLayout);
statusPanelLayout.setHorizontalGroup(
    statusPanelLayout.createParallelGroup(javax.swing.GroupLayout.
        Alignment.LEADING)
    .addComponent(statusPanelSeparator, javax.swing.GroupLayout.
        DEFAULT_SIZE, 785, Short.MAX_VALUE)
    .addGroup(statusPanelLayout.createSequentialGroup()
        .addContainerGap()
        .addComponent(statusMessageLabel)
        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
            RELATED, 615, Short.MAX_VALUE)
        .addComponent(progressBar, javax.swing.GroupLayout.
            PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
            javax.swing.GroupLayout.PREFERRED_SIZE)
```

```
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
                    RELATED)
                .addComponent(statusAnimationLabel)
                .addContainerGap())
        );
        statusPanelLayout.setVerticalGroup(
            statusPanelLayout.createParallelGroup(javax.swing.GroupLayout.
                Alignment.LEADING)
            .addGroup(statusPanelLayout.createSequentialGroup()
                .addComponent(statusPanelSeparator, javax.swing.GroupLayout.
                    PREFERRED_SIZE, 2, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.
                    RELATED, javax.swing.GroupLayout.DEFAULT_SIZE, Short.
                    MAX_VALUE)
                .addGroup(statusPanelLayout.createParallelGroup(javax.swing.
                    GroupLayout.Alignment.BASELINE)
                    .addComponent(statusMessageLabel)
                    .addComponent(statusAnimationLabel)
                    .addComponent(progressBar, javax.swing.GroupLayout.
                        PREFERRED_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
                        javax.swing.GroupLayout.PREFERRED_SIZE))
                .addGap(3, 3, 3))
        );

        setComponent(mainPanel);
        setMenuBar(menuBar);
        setStatusBar(statusPanel);
    }// </editor-fold>//GEN-END:initComponents

    private void playButtonActionPerformed(java.awt.event.ActionEvent evt) {//
        GEN-FIRST:event_playButtonActionPerformed
        try {
            actualDiceRound.newDice();
            int[] numbers = actualDiceRound.getResult();

            dice1.setText("" + numbers[0]);
            dice2.setText("" + numbers[1]);
            dice3.setText("" + numbers[2]);
            dice4.setText("" + numbers[3]);
            dice5.setText("" + numbers[4]);

        } catch (Exception ex) {
            Logger.getLogger(KniffelView.class.getName()).log(Level.SEVERE,
                null, ex);
        }

        if (actualDiceRound.getPlayCount() >= actualDiceRound.getMaxPlayCount
            ()) {
            playButton.setEnabled(false);
        }

        countLabel.setText(actualDiceRound.getPlayCount() + ". Wurf");

    }//GEN-LAST:event_playButtonActionPerformed

    private void dice1ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
        FIRST:event_dice1ActionPerformed
        this.lockDice(dice1, 0);
    }//GEN-LAST:event_dice1ActionPerformed
```

```
private void dice2ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
    FIRST:event_dice2ActionPerformed
    this.lockDice(dice2, 1);
}//GEN-LAST:event_dice2ActionPerformed

private void dice3ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
    FIRST:event_dice3ActionPerformed
    this.lockDice(dice3, 2);
}//GEN-LAST:event_dice3ActionPerformed

private void dice4ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
    FIRST:event_dice4ActionPerformed
    this.lockDice(dice4, 3);
}//GEN-LAST:event_dice4ActionPerformed

private void dice5ActionPerformed(java.awt.event.ActionEvent evt) {//GEN-
    FIRST:event_dice5ActionPerformed
    this.lockDice(dice5, 4);
}//GEN-LAST:event_dice5ActionPerformed

private void jButton7ActionPerformed(java.awt.event.ActionEvent evt) {//
    GEN-FIRST:event_jButton7ActionPerformed
    int selectedIndexCollectBox = sammelblockSelectBox.getSelectedIndex();
    int selectedIndexOtherBox = otherBlockSelectBox.getSelectedIndex();


    if (selectedIndexCollectBox > 0 && selectedIndexOtherBox > 0) {
        JOptionPane.showMessageDialog(null, "Bitte nur eine
            Speicherungsart verwenden");
    } else {

        if (selectedIndexCollectBox > 0) {
            lastGameSheet.addDiceRoundToSaveBlock(selectedIndexCollectBox
                - 1, actualDiceRound);
        } else if (selectedIndexOtherBox > 0) {
            lastGameSheet.addDiceRoundToOtherBlock(selectedIndexOtherBox -
                 1, actualDiceRound);
        }

        actualDiceRound = new DiceRound();
        this.updateTable();
        this.resetDices();


        System.out.println("InsertCount: " + lastGameSheet.getInsertCount
            ());
        if (lastGameSheet.getInsertCount() >= 13) {
            try {
                lastGameSheet.newRound();
                System.out.println("NewRound!");
            } catch (Exception ex) {
                Logger.getLogger(KniffelView.class.getName()).log(Level.
                    SEVERE, null, ex);
            }
        }
    }
    sammelblockSelectBox.setSelectedIndex(0);
    otherBlockSelectBox.setSelectedIndex(0);
```

```
}//GEN-LAST:event_jButton7ActionPerformed

private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {//
    GEN-FIRST:event_jMenuItem1ActionPerformed
    player.addGameSheet();
    this.lastGameSheet = player.getLastGameSheet();
    actualDiceRound = new DiceRound();
    countLabel.setText("");
    sammelblockSelectBox.setSelectedIndex(0);
    otherBlockSelectBox.setSelectedIndex(0);

    this.updateTable();
    this.resetDices();
}//GEN-LAST:event_jMenuItem1ActionPerformed
// Variables declaration - do not modify//GEN-BEGIN:variables
private javax.swing.JTable collectTable;
private javax.swing.JPanel controlPanel;
private javax.swing.JLabel countLabel;
private javax.swing.JToggleButton dice1;
private javax.swing.JToggleButton dice2;
private javax.swing.JToggleButton dice3;
private javax.swing.JToggleButton dice4;
private javax.swing.JToggleButton dice5;
private javax.swing.JPanel dicePanel;
private javax.swing.JButton jButton7;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel3;
private javax.swing.JMenuItem jMenuItem1;
private javax.swing.JScrollPane jScrollPane1;
private javax.swing.JScrollPane jScrollPane2;
private javax.swing.JPanel mainPanel;
private javax.swing.JMenuBar menuBar;
private javax.swing.JComboBox otherBlockSelectBox;
private javax.swing.JTable otherTable;
private javax.swing.JButton playButton;
private javax.swing.JProgressBar progressBar;
private javax.swing.JComboBox sammelblockSelectBox;
private javax.swing.JLabel statusAnimationLabel;
private javax.swing.JLabel statusMessageLabel;
private javax.swing.JPanel statusPanel;
// End of variables declaration//GEN-END:variables

private void lockDice(JToggleButton diceButton, int index) {
    if (actualDiceRound.getPlayCount() > 0) {
        diceButton.setEnabled(false);
        actualDiceRound.saveDice(index);
    } else {
        diceButton.setSelected(false);
    }
}

private void resetDices() {
    countLabel.setText("");

    dice1.setText("");
    dice1.setEnabled(true);
```

```
        dice1.setSelected(false);

        dice2.setText("");
        dice2.setEnabled(true);
        dice2.setSelected(false);

        dice3.setText("");
        dice3.setEnabled(true);
        dice3.setSelected(false);

        dice4.setText("");
        dice4.setEnabled(true);
        dice4.setSelected(false);

        dice5.setText("");
        dice5.setEnabled(true);
        dice5.setSelected(false);

        playButton.setEnabled(true);
    }

    public void updateTable() {

        for (int row = 0; row <= 5; row++) {
            for (int round = 0; round <= 2; round++) {
                collectTable.setValueAt("" + lastGameSheet.
                    getCollectBoxRoundRowResult(round, row), row, round + 1);
            }
        }

        for (int row = 0; row <= 6; row++) {
            for (int round = 0; round <= 2; round++) {
                otherTable.setValueAt("" + lastGameSheet.
                    getOtherBoxRoundRowResult(round, row), row, round + 1);
            }
        }

        // Summe row = 6
        for (int round = 0; round <= 2; round++) {
            collectTable.setValueAt("" + lastGameSheet.
                calculateCollectBoxRoundResult(round), 6, round + 1);
            otherTable.setValueAt("" + lastGameSheet.
                calculateOtherBoxRoundResult(round), 7, round + 1);
        }

    }
    private final Timer messageTimer;
    private final Timer busyIconTimer;
    private final Icon idleIcon;
    private final Icon[] busyIcons = new Icon[15];
    private int busyIconIndex = 0;
    private JDialog aboutBox;
}

            /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
```

```
package kniffel;

import java.util.Vector;

/**
 *
 * @author vorpro
 * Nummer 6
 */
public class Player {
    private String name;
    private Vector<GameSheet> gameSheets;

    public Player () {
        name = "Player";
        gameSheets = new Vector<GameSheet>();
    }

    public GameSheet getLastGameSheet() {
        return gameSheets.lastElement();
    }

    public void addGameSheet() {
        gameSheets.add(new GameSheet());
    }

    /**
     * @return the gameSheets
     */
    public Vector<GameSheet> getGameSheets() {
        return gameSheets;
    }
}
```

B.7 Concept Maps

B.7.1 List of 2-rated associations of programming novices in the pre-test

Id	concept1	concept2	label	label normal
3	Objekt	Attribute	haben	hat
3	Initialisierung	Instanz	führt zu	wird
8	Objekt	Attribute	besitzen	besitzt
8	Objekt	Klasse	sind einer ... zugewiesen	weist zu
10	Zustand	Objekt	ist in	?
10	Attribute	Objekt	definiert durch	definiert
10	Objekt	Klasse	gehört zu	?
10	Methode	Attribute	verändert durch	verändert
21	Attribute	Vererbung	können an eine weitere Klasse vererbt werden	?
21	Objekt	Methode	kann bearbeitet werden mit	bearbeitet
21	Methode	Operatoren	enthält	enthält
28	Methode	Fallunterscheidung	kann enthalten	enthält
31	Attribute	Zuweisung	legen Eigen-schaften fest	legt fest
32	Felder	Zuweisung	bekommt	bekommt
32	Klasse	Assoziation	verbunden	verbindet
74	Operatoren	Zuweisung	v.a. verwendet für	verwendet
83	Objekt	Attribute	hat	hat
83	Objekt	Datentyp	gibt es als ver-schiedene	?
87	Objekt	Assoziation	Verbindung	?
102	Objektorientierung	Objekt	benötigt	benötigt
102	Objekt	Methode	besteht aus	besteht aus
115	Datentyp	Attribute	bestimmen näher	bestimmt
115	Parameter	Datentyp	bestimmen näher	bestimmt
127	Objekt	Attribute	kann haben	hat
135	Klasse	Objekt	definiert	definiert

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
153	Objekt	Datentyp	hat einen bestimmten	hat
153	Vererbung	Objektorientierung	ist ein Merkmal der	?
153	Objekt	Zustand	hat	hat
153	Konstruktor	Objekt	erschafft ein	erschafft
159	Objekt	Attribute	kann besitzen	besitzt

B.7.2 List of 0-rated associations of programming novices in the pre-test

Id	concept1	concept2	label	label normal
3	Objekt	Parameter	definiert sich über	definiert
3	Objekt	Fallunterscheidung	kann durchgeführt werden	führt durch
7	Operatoren	Initialisierung	sorgen möglicherweise für eine	?
7	Parameter	Zuweisung	erzeugen in manchen Fällen	erzeugt
7	Operatoren	Methode	kennzeichnen eine	kennzeichnet
8	Zustand	Attribute	eines Objektes	?
8	Zuweisung	Felder	in ... eingeteilt	?
8	Zustand	Datentyp	eines Objektes	?
8	Zustand	Klasse	eines Objektes	?
8	Objekt	Zuweisung	durch	?
10	Parameter	Objekt	definiert durch	definiert
10	Methode	Parameter	verändert durch	verändert
10	Methode	Wiederholung	mögliche Art einer Methode	?
13	Datentyp	Parameter	definiert	definiert
13	Initialisierung	Datentyp	und	?
13	Klasse	Zuweisung	enthält	enthält
21	Klasse	Instanz	Synonym für Klasse	?
21	Felder	Parameter	mögliche Eingaben	?
23	Klasse	Operatoren	können verbunden/ getrennt werden durch	?
23	Klasse	Zuweisung	bekommen "Inhalt" durch	?
28	Klasse	Methode	definiert durch	definiert
28	Fallunterscheidung	Datentyp	besteht aus	besteht aus
28	Datentyp	Wiederholung	besteht aus	besteht aus
28	Zuweisung	Datentyp	definiert	definiert
31	Parameter	Zuweisung	legen Eigenschaften fest	legt fest

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
31	Wiederholung	Objekt	Befehl für	?
31	Zuweisung	Objekt	bestimmt	bestimmt
31	Fallunterscheidung	Objekt	Befehl für	?
32	Fallunterscheidung	Operatoren	weiter durch	?
32	Zuweisung	Fallunterscheidung	bringen	bringt
32	Operatoren	Klasse	unterteilt	unterteilt
32	Parameter	Wiederholung	bei bestimmten	?
32	Parameter	Datentyp	gespeichert als	?
47	Fallunterscheidung	Objektorientierung	Gegensatz zu	?
71	Konstruktor	Operatoren	erstellt für	erstellt
74	Wiederholung	Parameter	definiert durch	definiert
74	Zuweisung	Parameter	definiert durch	definiert
83	Objekt	Parameter	besteht aus	besteht aus
83	Datentyp	Parameter	besteht aus	besteht aus
83	Attribute	Parameter	hat	hat
102	Objekt	Klasse	besteht aus	besteht aus
109	Objekt	Datentyp	hat	hat
127	Objekt	Datentyp	notwendig	?
132	Wiederholung	Operatoren	are caused by	?
132	Parameter	Objekt	influence	?
132	Objekt	Operatoren	instruct	?
139	Datentyp	Operatoren	Zuweisung	?
141	Überladen	Datenkapselung	kein Speicherplatz vorhanden Daten müssen komprimiert werden zip Ordner	?
141	Vererbung	Wiederholung	Wenn ein Programm fehlerhaft ist und weiter erweitert wird, wird der Fehler vererbt	?
153	Methode	Zuweisung	Beispiel	?
153	Zuweisung	Objekt	weißt einem Objekt einen Wert zu	weist zu
153	Methode	Operatoren	Beispiel	?
153	Objekt	Parameter	hat	hat

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
155	Fallunterscheidung	Zuweisung	ist nötig zur	?
155	Parameter	Fallunterscheidung	erfordert	erfordert
159	Parameter	Attribute	besitzen	besitzt
159	Objekt	Parameter	kann besitzen	besitzt
159	Objekt	Datentyp	ist ein bestimmter	bestimmt

B.7.3 List of 2-rated associations of programming novices in the post-test

Id	concept1	concept2	label	label normal
3	Objekt	Attribute	haben	hat
3	Objekt	Klasse	übergeordnet	ist übergeordnet
3	Konstruktor	Objekt	erzeugt	erzeugt
5	Objektorientierung	Datenkapselung	durch	?
5	Zugriffsmodifikator	Datenkapselung	erlaubt/ unterbindet	?
5	Zugriffsmodifikator	Objektorientierung	erlaubt/ unterbindet	?
5	Objekt	Klasse	Bestandteil	besteht aus
5	Methode	Attribute	ändert	ändert
5	Konstruktor	Objekt	erstellt	erstellt
5	Attribute	Objekt	beschreibt Eigenschaften	beschreibt
7	Zugriffsmodifikator	Methode	macht von außen sichtbar/unsichtbar	macht sichtbar
7	Datenkapselung	Attribute	machen von außen sichtbar	macht sichtbar
7	Attribute	Objekt	beschreiben	beschreibt
7	Methode	Zustand	verändert	verändert
7	Konstruktor	Objekt	erschaffen	erschafft
8	Methode	Objekt	verändern den Zustand	verändert
8	Methode	Überladen	... möglich	?
8	Klasse	Objekt	kann erzeugen	erzeugt
8	Objekt	Zustand	besitzen einen	besitzt
10	Parameter	Methode	enthält	enthält
10	Objekt	Methode	Bearbeitung	?
10	Objekt	Attribute	Eigenschaft	?
13	Assoziation	Klasse	stellt Zusammenhang her	?
13	Objekt	Attribute	besitzt	besitzt
13	Methode	Attribute	ändert	ändert
13	Attribute	Zustand	beschreiben	beschreibt

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
13	Operatoren	Fallunterscheidung	werden benötigt bei	benötigt
13	Zuweisung	Felder	um Daten zu übergeben	?
13	Konstruktor	Objekt	erstellt	erstellt
13	Datenkapselung	Objektorientierung	Zentrales Prinzip der	?
13	Operatoren	Felder	stellen Vergleiche oder mathematische Verknüpfungen her	?
13	Zugriffsmodifikator	Attribute	regelt Zugriff auf	?
13	Parameter	Methode	werden übergeben um Eigenschaften zu ändern	?
15	Objekt	Klasse	sind Grundpläne für	?
15	Objekt	Datentyp	sind keine primitiven	?
15	Klasse	Attribute	enthalten	enthält
15	Objektorientierung	Klasse	mit Hilfe von	?
15	Klasse	Methode	enthalten	enthält
15	Datenkapselung	Objektorientierung	durch	?
15	Objekt	Instanz	sind	ist
21	Datenkapselung	Objekt	Zugriff auf	?
21	Operatoren	Zuweisung	verändert den Wert einer Variablen durch	?
21	Objektorientierung	Objekt	speichert Daten in	?
21	Zugriffsmodifikator	Datenkapselung	bestimmt Art der	?
21	Objekt	Instanz	Synonym	?
21	Objekt	Methode	enthält	enthält
21	Parameter	Datentyp	hat einen bestimmten	?
23	Datentyp	Felder	hat bestimmten	hat
23	Klasse	Objekt	enthält	enthält
23	Datentyp	Attribute	hat bestimmten	hat
23	Zuweisung	Attribute	von Attributwert	?

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
23	Konstruktor	Objekt	macht aus Klasse	?
23	Methode	Attribute	kann verändern	verändert
23	Objektorientierung	Datenkapselung	beschreibt	beschreibt
23	Objekt	Attribute	hat	hat
23	Attribute	Felder	können als Felder beschreiben werden	?
23	Datenkapselung	Attribute	Sichtbarkeit von	?
28	Methode	Objekt	verändert	verändert
28	Zugriffsmodifikator	Objekt	beeinflusst	beeinflusst
28	Objekt	Klasse	bestimmen	bestimmt
31	Attribute	Datentyp	bestimmt mit	bestimmt
31	Attribute	Vererbung	können durch ... weitergegen werden	gibt weiter
31	Attribute	Klasse	besitzt	besitzt
31	Klasse	Assoziation	verbinden mit anderen Klassen ist	verbindet
31	Methode	Vererbung	können durch ... weitergegen werden	gibt weiter
31	Datenkapselung	Methode	Schutz vor Zugriff von außen durch	?
31	Zugriffsmodifikator	Datenkapselung	mit Hilfe von	?
31	Methode	Fallunterscheidung	besitzen unter Umständen	besitzt
31	Methode	Wiederholung	besitzen unter Umständen	besitzt
31	Datenkapselung	Attribute	Schutz vor Zugriff von außen durch	?
32	Klasse	Attribute	haben	hat
32	Objekt	Konstruktor	werden erstellt durch	erstellt
32	Objektorientierung	Objekt	arbeitet mit	?
35	Initialisierung	Konstruktor	durch	?
35	Klasse	Konstruktor	enthält	enthält
35	Objekt	Methode	enthält/ ruft auf & definiert	enthält

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
35	Konstruktor	Methode	ist eine	ist
35	Methode	Parameter	übergebt	übergibt
35	Klasse	Attribute	definiert	definiert
47	Klasse	Methode	besteht aus	besteht aus
48	Klasse	Objekt	durch Erstellung Konnstruktor	?
48	Objekt	Attribute	hat	hat
48	Attribute	Methode	verändert	verändert
65	Klasse	Attribute	besteht aus	besteht aus
65	Klasse	Konstruktor	besteht aus	besteht aus
65	Klasse	Methode	besteht aus	besteht aus
69	Methode	Attribute	können verändern	verändert
69	Konstruktor	Objekt	erstellt	erstellt
69	Klasse	Methode	hat	hat
69	Attribute	Zuweisung	Wert	?
69	Methode	Konstruktor	z.B. den	?
69	Attribute	Datentyp	haben	hat
69	Objektorientierung	Objekt	basiert auf Verwendung von	?
69	Klasse	Attribute	besitzt	besitzt
69	Methode	Parameter	gegebenfalls	?
69	Methode	Fallunterscheidung	uses	?
69	Methode	Operatoren	benutzen	benutzt
69	Objekt	Datenkapselung	bietet den Vorteil der	?
69	Objekt	Klasse	wird erstellt aus	erstellt
71	Objekt	Zustand	hat verschiedenen	hat
74	Objekt	Methode	Zugriff durch	?
74	Methode	Zustand	ändert eines Objektes . . .	ändert
74	Methode	Fallunterscheidung	können verwenden	verwendet
74	Methode	Überladen	Möglichkeit des	?
74	Objekt	Datenkapselung	verwenden	verwendet
74	Methode	Wiederholung	können verwenden	verwendet
74	Methode	Parameter	verwenden	verwendet
83	Objektorientierung	Klasse	durch	?

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
83	Datentyp	Attribute	ist darin vorhanden	?
83	Objekt	Attribute	wird beschrieben durch	beschreibt
83	Methode	Wiederholung	einbaubar	?
83	Attribute	Methode	veränderbar	?
83	Objekt	Datenkapselung	besitzt eine	besitzt
83	Datentyp	Felder	ist darin vorhanden	?
87	Objekt	Assoziation	Verbindung	?
94	Fallunterscheidung	Methode	kann Teil einer Methode sein	?
94	Klasse	Methode	beteht aus	besteht aus
94	Klasse	Konstruktor	besteht aus	besteht aus
94	Attribute	Zuweisung	brauchen	braucht
94	Objektorientierung	Datenkapselung	basiert auf	basiert auf
102	Klasse	Methode	haben	hat
106	Instanz	Klasse	ist von einer	?
106	Attribute	Datentyp	Variablen haben einen	?
106	Attribute	Datenkapselung	sind unveränder-bar durch	?
106	Klasse	Methode	hat	hat
106	Methode	Parameter	manchmal mit	?
106	Instanz	Zustand	hat	hat
106	Klasse	Konstruktor	hat	hat
106	Klasse	Attribute	hat	hat
107	Objekt	Attribute	in bestimmtem Zu-stand	?
107	Objekt	Methode	veränderbar	?
107	Klasse	Objekt	mit gleichen At-tributen	?
109	Attribute	Zuweisung	brauchen	braucht
109	Objekt	Zustand	hat	hat
109	Parameter	Datentyp	haben	hat
109	Klasse	Attribute	bestimmt	bestimmt
109	Parameter	Methode	bestimmen	bestimmt
109	Methode	Wiederholung	können enthalten	enthält

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
109	Klasse	Konstruktor	hat	hat
109	Konstruktor	Objekt	erzeugt	erzeugt
109	Methode	Fallunterscheidung	können enthalten	enthält
115	Objekt	Konstruktor	hat	hat
115	Zugriffsmodifikator	Objekt	versteckt	versteckt
115	Objekt	Attribute	besitzt verschiedene	besitzt
115	Klasse	Objekt	Bauplan für	?
115	Klasse	Konstruktor	hat	hat
115	Methode	Objekt	verändert	verändert
119	Methode	Wiederholung	können sein	ist
119	Objekt	Konstruktor	wird durch	ist
119	Konstruktor	Attribute	initialisiert mit	initialisiert
119	Methode	Fallunterscheidung	können sein	ist
119	Objekt	Zustand	hat	hat
119	Klasse	Attribute	legt fest	legt fest
119	Objekt	Attribute	hat	hat
127	Klasse	Konstruktor	enthält	enthält
127	Objektorientierung	Datenkapselung	vereinfacht	vereinfacht
127	Überladen	Parameter	unterschiedliche	?
127	Objekt	Attribute	verfügt über	verfügt über
127	Attribute	Datentyp	hat	hat
127	Klasse	Methode	verfügt über	verfügt über
127	Klasse	Attribute	verfügt über	verfügt über
127	Konstruktor	Objekt	erstellt	erstellt
127	Parameter	Datentyp	muss	muss
132	Methode	Konstruktor	können sein	ist
132	Methode	Überladen	können ... werden	ist
132	Klasse	Attribute	haben	hat
132	Klasse	Objekt	werden aus ... erzeugt	erzeugt
132	Parameter	Datentyp	haben verschiedene	hat
132	Methode	Wiederholung	können enthalten	enthält
132	Methode	Fallunterscheidung	können enthalten	enthält
132	Methode	Operatoren	verwenden	verwendet

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
132	Methode	Objekt	verändern	verändert
133	Attribute	Objekt	Eigenschaften	?
133	Methode	Attribute	manipuliert	manipuliert
133	Zugriffsmodifikator	Datenkapselung	legt fest	legt fest
133	Datenkapselung	Attribute	verbirgt	verbirgt
133	Klasse	Objekt	legt bestimmte Attribute fest	legt fest
133	Konstruktor	Objekt	konstruiert	konstruiert
135	Attribute	Objekt	beschreiben	beschreibt
135	Klasse	Objekt	legt fest	legt fest
135	Parameter	Methode	können enthalten	enthält
135	Klasse	Methode	enthält und beschreibt	enthält
135	Datenkapselung	Klasse	macht sicherer	?
135	Klasse	Konstruktor	enthält und beschreibt	enthält
135	Initialisierung	Attribute	legt fest	legt fest
135	Methode	Objekt	ändert oder gibt weiter	ändert
139	Klasse	Objekt	hat	hat
139	Methode	Attribute	ändert	ändert
141	Zustand	Attribute	Attribute zeigen auf in welchem Zustand sich die Klassen befinden	?
141	Objekt	Attribute	Eigenschaften von Objekten bzw. einer Klasse	?
141	Methode	Wiederholung	Kann man z.B. in Methoden einbauen, damit sie sich immer wiederholen "schleifen"	?
148	Zugriffsmodifikator	Datenkapselung	ermöglichen	ermöglicht
148	Parameter	Methode	in	?
148	Konstruktor	Objekt	erstellt	erstellt
148	Methode	Attribute	kann Werte verändern	verändert

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
148	Attribute	Objekt	sind Eigenschaften von	?
148	Datenkapselung	Attribute	ermöglicht, dass sie nicht von überall geändert werden können	?
148	Klasse	Objekt	besteht aus	besteht aus
153	Konstruktor	Objekt	erschafft ein	erschafft
153	Objekt	Klasse	gerhört an	gehört an
153	Zuweisung	Attribute	Weißt einem Attribut einen bestimmten Wert zu	?
153	Objektorientierung	Objekt	hat im Zentrum	hat
153	Zugriffsmodifikator	Datenkapselung	bewirkt eine	bewirkt
153	Objekt	Attribute	hat verschiedene	hat
155	Objektorientierung	Klasse	hat "Bausteine"	?
155	Klasse	Methode	hat	hat
155	Klasse	Attribute	hat	hat
155	Konstruktor	Objekt	"baut"	?
155	Klasse	Konstruktor	hat	hat
155	Klasse	Objekt	enthält	enthält
155	Objektorientierung	Datenkapselung	führt zu	wird
159	Objekt	Attribute	hat	hat
159	Klasse	Objekt	hat	hat
161	Objekt	Zustand	hat	hat
161	Klasse	Attribute	legt fest	legt fest
161	Zustand	Attribute	festgelegt durch	legt fest
161	Objekt	Klasse	gehört zu	?
161	Objekt	Attribute	hat	hat
161	Klasse	Objekt	beinhaltet "Bauplan"	?
161	Methode	Attribute	kann ändern	ändert
161	Objektorientierung	Objekt	übergeordnetes Konzept	?

B.7.4 List of 0-rated associations of programming novices in the post-test

Id	concept1	concept2	label	label normal
3	Parameter	Fallunterscheidung	kann durchgeführt werden	führt durch
5	Parameter	Attribute	gibt Werte an	gibt an
5	Operatoren	Attribute	Hilfsmittel	?
10	Klasse	Instanz	Unterkategorie	?
13	Datenkapselung	Attribute	vereint gleiche	vereint
13	Initialisierung	Felder	zur Benennung von	?
21	Instanz	Klasse	mehrere Instanzen zusammen ergeben eine	?
21	Methode	Operatoren	enthält	enthält
21	Vererbung	Klasse	neue Klassen mit Ähnlichen Attributen können durch Vererbung angelegt werden	?
21	Methode	Parameter	liefert	liefert
21	Initialisierung	Parameter	bestimmt Anfangswert	?
21	Objekt	Felder	speichert Daten in	?
23	Wiederholung	Methode	von	?
23	Fallunterscheidung	Methode	von	?
23	Wiederholung	Attribute	von	?
23	Fallunterscheidung	Attribute	von	?
28	Fallunterscheidung	Klasse	teil einer	?
31	Objektorientierung	Klasse	besitzt ein oder mehrere	besitzt
31	Parameter	Zuweisung	erfolgt	erfolgt
31	Operatoren	Vererbung	können durch ... weitergegen werden	gibt weiter
31	Klasse	Methode	und	?
31	Methode	Operatoren	oder	?
31	Klasse	Konstruktor	erzeugt durch	erzeugt

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
32	Methode	Operatoren	umgesetzt durch	setzt um
32	Klasse	Methode	benutzen	benutzt
32	Objekt	Klasse	werden unterteilt	unterteilt
32	Klasse	Vererbung	neue Attribute hinzufügen	?
35	Objekt	Initialisierung	enthält	enthält
47	Methode	Fallunterscheidung	wird eingegeben als	?
47	Methode	Konstruktor	und eingeben in	?
47	Methode	Zugriffsmodifikator	besteht aus	besteht aus
47	Methode	Vererbung	wird eingegeben als	?
47	Methode	Wiederholung	wird eingegeben als	?
47	Klasse	Objekt	besteht aus	besteht aus
48	Parameter	Methode	verändert	verändert
48	Objekt	Überladen	kann Wiederholungen enthalten	?
69	Felder	Datentyp	speichern	speichert
71	Attribute	Datentyp	sind	ist
74	Objekt	Attribute	einer Klasse unterscheiden sich in	?
74	Methode	Datentyp	festlegen des	legt fest
74	Wiederholung	Instanz	können "verschachtelt" werden	?
74	Methode	Zugriffsmodifikator	teils als	?
74	Objekt	Parameter	verwenden	verwendet
74	Datentyp	Initialisierung	muss zunächst .. werden	ist
83	Klasse	Objekt	beinhaltet	beinhaltet
83	Datentyp	Klasse	ist darin vorhanden	?
83	Attribute	Fallunterscheidung	hat zeitweise	hat
83	Objekt	Assoziation	wenn mehrere zusammenpassen	?
83	Überladen	Attribute	durch zu viel	?
83	Attribute	Zustand	haben einen	hat
83	Objekt	Überladen	kann	kann

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
83	Attribute	Parameter	besitzt	besitzt
83	Attribute	Felder	beinhaltet	beinhaltet
83	Konstruktor	Objekt	folgt danach	?
83	Klasse	Konstruktor	"Grundgerüst"	?
87	Methode	Fallunterscheidung	mögliche Methode	?
87	Attribute	Felder	Umfang	?
87	Konstruktor	Attribute	werden dort eingetragen	trägt ein
87	Methode	Wiederholung	kann wiederholt werden	?
94	Zuweisung	Datentyp	des	?
94	Wiederholung	Methode	kann Teil einer Wiederholung sein	?
94	Parameter	Objekt	bestimmt Zustand von Objekt	?
94	Attribute	Parameter	besitzen abhängig von Objekt bestimmte	?
94	Klasse	Instanz	läuft in einer	?
94	Fallunterscheidung	Zugriffsmodifikator	ist	ist
94	Klasse	Attribute	weißt Objekten Attribute zu	?
102	Felder	Parameter	haben	hat
102	Felder	Attribute	haben	hat
102	Methode	Felder	brauchen	braucht
106	Fallunterscheidung	Objektorientierung	benutzt man zur	?
109	Initialisierung	Parameter	erstellt	erstellt
119	Methode	Zuweisung	sind " Werkzeuge" zur	?
119	Attribute	Parameter	die durch ... festgelegt sind	legt fest
119	Zuweisung	Parameter	von	?
119	Methode	Operatoren	mit Hilfe von	?
127	Fallunterscheidung	Wiederholung	kann benutzt werden	benutzt
127	Objekt	Parameter	hat	hat
127	Zugriffsmodifikator	Zuweisung	wird benutzt	benutzt

Continued on next page

continued from previous page

Id	concept1	concept2	label	label normal
132	Parameter	Initialisierung	entstehen durch	entsteht
133	Parameter	Attribute	gibt genauen Wert an	?
135	Wiederholung	Konstruktor	widerholt	wiederholt
135	Datentyp	Operatoren	beschreibt	beschreibt
135	Parameter	Initialisierung	lassen bestimmte Effekte aus, bei	?
135	Datentyp	Objekt	beschreibt	beschreibt
135	Wiederholung	Methode	wiederholt	wiederholt
141	Attribute	Parameter	definiert den Zustand (höhe,länge, breite Farbe) eines Attributes	?
141	Klasse	Objekt	ist Überbegriff von	?
143	Methode	Objekt	Zuweisung	?
148	Parameter	Attribute	verändert	verändert
148	Klasse	Attribute	gleiche	?
153	Methode	Wiederholung	Beispiel	?
153	Methode	Initialisierung	Beispiel	?
155	Fallunterscheidung	Zuweisung	ermöglicht	ermöglicht
159	Methode	Fallunterscheidung	mögliche Methoden sind	?
159	Operatoren	Parameter	können besitzen	besitzt
159	Methode	Operatoren	hat	hat
161	Felder	Parameter	speichert	speichert
161	Initialisierung	Parameter	legt Wert fest	?

B.8 Student Questions

Id	question	answer	knowledge	school	concept
2	Ich kann Klassen auch nutzen um Funktionen auszulagern?	1	2	4	class
2	Wie kann ich auf eine Var. Einer anderen Klasse zugreifen?	2	2	4	association
2	Var. Aus anderen Klassen def.	1	2	4	attribute
2	Wie führe ich Konstruktor aus?	1	2	4	constructor
2	Was ist static	1	2	4	access modifier
2	Array mit Objekten	2	2	4	array
2	Was sind Wrapperklassen?	1	2	4	datatype
2	Wie kennzeichne ich Parameter einer Methode?	1	2	4	parameter
2	Wie funktoniert ein Konstruktor?	2	2	4	constructor
2	Wie kann ich mehrere Param. Überg.	1	2	4	parameter
2	Was bedeutet das "this. . . ."	1	2	4	object
2	Wo erstelle ich die Objekte?	1	2	4	object
2	Warum funktioniert die Zuweisung nicht? (static reference)	1	2	4	assignment
2	Was steht in einem gerade Instanzierten Array?	1	2	4	array
2	Cast von Objekt aus String	2	2	4	datatype
2	Wie krieg ich Scrollbars?	1	2	4	other
2	Wie nutzte ich zweidimensionale Arrays?	1	2	4	array
2	Wie kann ich meine ITable aktualisieren?	1	2	4	other
2	Brauch eine Klasse einen Konstruktor?	1	2	4	class
2	Objekte initialisieren bevor sie genutzt werden?	1	2	4	initialization
3	Nach "if" mehrere Befehle geben (Block durch geschweifte Klammern)	1	1	5	cond. Statement

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
3	Klammern zusammenfassen Syntax: Mehrere Bedingungen durch	1	1	5	operators
3	Wertebereich einschränken	1	1	5	datatype
3	Was ist x=2 und x==2	1	1	5	operators
5	Was machen die Parameter in Klammern bei Methoden?	3	1	5	parameter
8	Wie kann ich Zahlen in fester Reihenfolge zufällig generieren?	1	1	5	other
9	Was sind das für Zeichen, wenn ich eine Array ausgeben möchte?	2	2	5	array
9	s=="1" funktioniert nicht?	2	2	5	syntax
10	Muss eine Static- Methode definieren	1	1	5	other
10	Keine Methode in einer Methode definieren	1	1	5	other
10	Wie kann ich auf Var. Verschiedener Klassen zugreifen?	2	1	5	association
10	Was ist ein Enam	1	1	5	datatype
10	Was bedeutet "Void"?	1	1	5	datatype
10	Wie sieht das aus wenn eine Funktion etwas zurückgibt?	2	1	5	method
10	Wann setzte ich ein Attribut auf einen Wert?	1	1	5	attribute
10	Wie sieht ein Konstruktor aus?	2	1	5	constructor
10	Fallunterscheidung?	2	1	5	cond. Statement
10	Wie mache ich eine Klasse (???)?	1	1	5	class
10	Syntaxfehler () hinter Methode vergessen	2	1	5	syntax
10	Variableninitialisierung	1	1	5	initialization
10	Wie kann ich versch. Daten in eine Array packen?	1	1	5	array
10	Warum brauch er die Parameter?	2	1	5	parameter

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
10	Brauche ich eine elo für if-Abfragen?	1	1	5	cond. Statement
14	Wie vergleiche ich Zahlen?	2	1	4	operators
15	statische Methoden	3	1	5	method
18	Was bedeutet das "privat"?	1	0	0	access modifier
21	Wie kann man Benutzereingaben verwenden?	2	1	5	other
22	Wie importiere ich " random"?	9	2	3	other
22	Schachtelungen von Funktionsaufrufen möglich?	5	2	3	method
22	Warum wandelt die toString Methode einen Integer in was anderes um als die ursprüngliche eingegebene Zahl	8	2	3	datatype
22	Wollte in einer Klasse auf ein Attribut einer anderen Vergleichen "hatte "="statt ==+ Methode war nicht static	2	2	3	syntax
23	Beim Konstuktor muss nicht unbedingt angegeben werden was er zurückliefert, oder?	1	1	5	constructor
23	Werden Methoden nacheinander abgearbeitet wie sie im Code stehen?	1	1	5	other
23	Mehrere Rückgabewerte in einer Funktion möglich?	1	1	5	method
23	Fehlender Rückgabewert beim compilieren?	3	1	5	syntax
23	Unreachable Code? (beim Kompilieren)	1	1	5	syntax
23	Wird eine Methode immer wieder aufgerufen oder wird sich der Wert gemerkt?	1	1	5	method
23	Wie kann ich den Konstruktor aufrufen?	1	1	5	constructor
25	Wie schachtel ich if- Abfragen richtig?	1	1	4	cond. Statement
28	Unterschied Klasse / Objekt?	1	1	5	object
29	Ist Constructor= compiler?	1	0	0	other

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
31	Ist char ein einzelnes Zeichen?	1	1	5	datatype
31	Muss ich für zwei Arrays zwei Attribute benutzen? Bzw. wie unterscheide ich zwei Arrays?	1	1	5	array
31	Gibt es einen except Befehl?	1	1	5	other
32	Wie geht die For-Schleife?	2	1	5	loop
32	If- Abfrage	2	1	5	cond. Statement
32	Fehlerhafte abbruchbedingung	1	1	5	loop
32	Zufallszahlen mit Ausnahme generieren	1	1	5	other
32	Array. Initialisieren	1	1	5	array
35	Objekt erzeugen im Mode?	1	1	5	object
35	Methode mit Rückgabewert	1	1	5	method
36	Was bedeutet "new" bei der Definition von einem Array?	1	1	3	array
39	Objekt erzeugen im Mode (???)	2	0	0	object
39	lokale Variablen	2	0	0	other
41	Wenn innerhalb einer Methide andere Aufgerufen werden, müssen diese "vorher" definiert werden?	1	2	4	method
42	Was machen die () Klammern hinter dem Methodenaufrufen?	3	1	4	method
47	Muss meine Methode einen Rückgabewert haben?	1	1	5	method
47	Wie gebe ich Textauf der Konsole aus?	11	1	5	other
49	Array erklärt	2	0	0	array
49	For-Schleife erklärt	2	0	0	loop
55	Wie greife ich auf andere Klassen zu?	2	2	0	association
55	Wie geht ein "oder" in einer Abfrage?	1	2	0	operators
55	Var. Aus anderen Klassen def.	1	2	0	association
55	Nicht statische Methode statisch aufrufen.	1	2	0	other
55	Wie mache ich Eingaben?	1	2	0	other

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
55	Aufruf über mehrere Klassen	1	2	0	association
55	Wie sichere ich mein Porgramm gegen falsch Eingaben?	1	2	0	other
55	Warum werden 0000 als 0 gelesen?	1	2	0	datatype
59	Buttons wie	25	0	0	other
62	private? Vs puplic	2	1	5	access modifier
65	Objekt erzeugen im Mode (???)	2	1	5	object
69	Methode mit Rückgabewert	1	1	5	method
71	Was bedeutet "new int(5)"?	1	1	5	initialization
71	Wie geht if Abfrage	1	1	5	cond. Statement
71	Var. Ref. Aus anderer Klasse	2	1	5	association
71	Methodenaufruf in anderer Klasse	1	1	5	association
71	Mehrere Abfragen in einer if-Abfrage	2	1	5	cond. Statement
71	Wie gehen Arrays	2	1	5	array
72	Wie gehen Arrays	2	1	4	array
73	Wie kann ich if -Abfragen "gleichwertig" machen?	2	1	5	cond. Statement
73	Wie mache ich Wiederholungen?	2	1	5	loop
74	Probleme mit Schleifenvariable "inti=" funktioniert nicht?	2	1	5	syntax
74	Wie breche ich eine Schleife ab?	2	1	5	loop
74	Wie wird der Konstruktor aufgerufen?	3	1	5	constructor
74	Vergleich mit "=" funktioniert nicht?	2	1	5	syntax
83	Wo müssen die Attribute einer Klasse stehen in der Klasse? Reihenfolge wichtig?	2	1	5	class
83	Wo müssen Methoden notiert werden?	3	1	5	class

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
83	Methodenaufrufe in Methoden?	3	1	5	method
83	Wie funktioniert das mit der main-Methode?	3	1	5	other
83	Array out Of Bounds Exception?	2	1	5	other
89	Was ist x=2 und x==2	1	1	5	operators
90	String vergleichen (.equals(str))	26	3	4	datatype
91	Fehler wegen fehlender Semikolon	1	2	3	syntax
92	Wie man input von Zahlen macht. Mit System.inetc.	8	2	5	other
92	Dynamische Arrays	8	2	5	array
94	Ist ein Array ein Attribut?	5	1	5	attribute
94	Wie bekomme ich Variablen von einer Klasse in eine andere?	10	1	5	association
97	Objekt erzeugen im Mode (???)	2	1	4	object
98	Wie erstelle ich ein Frame mit einem Button?	8	3	4	other
98	Wie erstelle ich eine nummerierete Liste von Objekten –> List, Array	1	3	4	array
98	Wie ordne ich meine Objekt in einem Panel an? –> Layout Manager	1	3	4	other
100	private? Vs puplic	2	0	0	access modifier
102	Objekterstellung im Code	2	1	5	object
104	Wie kann man innergalb der main-Methode nicht-static Variablen und Fkt. Aufrufen?	1	0	0	other
104	Wie kann man eine dynamische Anzahl an Objekten erzeugen?	1	0	0	other
106	Objekt erzeugen im Mode (???)	2	1	5	object

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
106	Methode Objekt als Parameter übergeben	2	1	5	parameter
109	Objekterzeugung im Code?	2	1	5	object
109	Inititialisierung von Attributen über Konstruktor?	2	1	5	initialization
109	lokale Variablen	1	1	5	other
113	array.length liefert nicht Anzahl sinnvoll belegter Elemente , sondern Gesamtgröße des Arrays	1	3	0	array
114	Wie verwendet man Felde?	1	0	0	array
114	Wie verwendet man Schleifen?	1	0	0	loop
118	(???) Datentypen , Arrays in Java	2	0	0	datatype
118	Konstruktor mit/ ohne Panometer?	2	0	0	constructor
118	for-schleife	2	0	0	loop
121	Was ist eine Klasse/ ein Objekt	1	0	0	class
123	Verständnisprobleme mit Array und Input!	9	2	1	array
123	Was ist Void und wofür braucht man das?	5	2	1	datatype
123	Unterschied Objekt und Klasse?	1	2	1	class
124	Scope von Variablen	2	1	2	attribute
124	Kann man While-Schleifen zählen lassen?	1	1	2	loop
125	Objekterzeugung im Code?	2	0	0	object
125	Inititialisierung von Attributen über Konstruktor?	2	0	0	initialization
125	Arrays in Java?	2	0	0	array
127	Objekterzeugung im Code?	2	1	5	object
127	Inititialisierung von Attributen über Konstruktor?	2	1	5	initialization
132	Objekterzeugung im Code?	2	1	5	object
132	Inititialisierung von Attributen über Konstruktor?	2	1	5	initialization
132	lokale Variablen	1	1	5	other

Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
133	lokale Variablen	2	1	5	other
133	Objekt erzeugen im Mode (???)	2	1	5	object
134	Leerzeichen in Variablennamen?	1	0	0	syntax
134	Wie erstelle ich ein neues Array?	1	0	0	array
134	Benutze ich while oder for für die 12 Schritte bei Mastermind?	1	0	0	loop
134	Wie greife ich mit this auf Klassen Variablen zu?	1	0	0	object
135	Objekt erzeugen im Mode (???)	2	1	5	object
135	Konstruktor von Mode mit/ ohne Parameter	1	1	5	constructor
139	Objekt erzeugen im Mode (???)	2	1	5	object
140	Wie erstelle ich eine neue Array-Instanz (new Operator)?	2	2	5	initialization
140	Index-Bereich eines Arrays der Größe n?	1	2	5	array
141	Was sind Methodenköpfe, was Methodenrümpfe?	2	1	5	method
141	Wie funktioniert system.out.println?	2	1	5	other
142	Was ist x=2 und x==2	1	1	5	operators
145	Listen, Stacks, . . . in Java?	9	2	5	other
147	Rückgabewert, Parameter einer Funktion?	1	2	4	method
147	Wie ruft man static- Methoden einer anderen Klasse auf?	1	2	4	association
147	Unterschied zwischen int und Integer?	1	2	4	datatype
147	Globale Variable wurde nicht geschrieben weil durch lokale Variable verdeckt	1	2	4	attribute
147	Allgemeine Frage zur Vererbung	1	2	4	inheritance

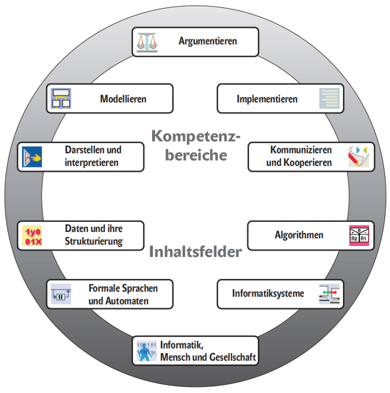
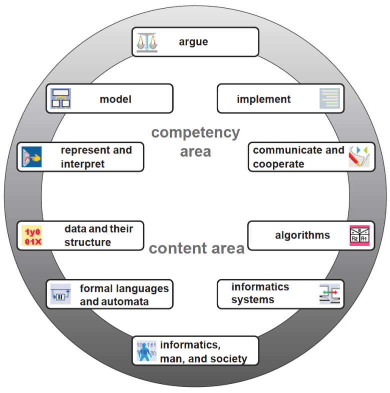
Continued on next page

continued from previous page

Id	question	answer	knowledge	school	concept
148	Wie wird ein Objekt erzeugt?	1	1	5	initialization
148	Zugriff auf Attribut	1	1	5	attribute
148	Rückgabewert von Methoden spezifizieren	3	1	5	method
148	Wie liefert Methode Wert zurück	1	1	5	method
151	Wo Attribute definieren	1	1	3	class
151	Syntax der if-Anweisung?	2	1	3	cond. Statement
151	Konstruktor	3	1	3	constructor
151	for-schleife	2	1	3	loop
152	Kann ich in einer Schleife Funktionen aufrufen?	1	2	4	loop
156	Wie kann ich eine Schleife abbrechen?	2	2	1	loop
156	Kann eine Methode auch booleans zurückliefern?	1	2	1	method
157	Wie wandelt man einen String in eine Zahl um?	2	3	4	datatype

C Translations

Page	Original Source	Translation
p. 76	“Der Hauptunterschied beider fachdidaktischen Vorgehensweisen ist die sequenzielle Abfolge der Themen. Der Schwerpunkt beider Vorgehensweisen liegt eher in der Programmierung als in der Modellierung, ohne dass auf die Modellierung verzichtet wird.” (Ehlert 2012, p. 16)	The main difference of both teaching approaches is the sequential order of topics. Both approaches focus more in programming as in the modeling, without giving up the modeling.
p. 76	“Es wurden keine signifikanten Unterschiede zwischen dem OOP-First-Vorgehen und dem OOP-Later-Vorgehen im Hinblick auf den Lernerfolg der Schülerinnen und Schüler festgestellt bzw. gemessen (die p-Werte für die neun abgefragten Themen liegen zwischen 0,18 und 0,83). Die OOP-Later-Schüler hatten in vielen Bereichen ein (signifikant) besseres subjektives Erleben, sowohl in den Frage-Bereichen („Schule“, „Fach Informatik“ und „Thema“) als auch in den Erlebnis-Dimensionen (emotionales, kognitives und motivationales Erleben).” (Ehlert 2012, p. 184)	No significant differences between the OOP-first approach and the OOP-later approach were observed or measured regarding the learning gain of pupils / students (p-values for the nine requested topics are between 0.18 and 0.83). OOP-later students had a (significantly) better subjective experience in the relevant fields (“school”, “subject” and “topic”) as well as in the dimensions of experience (emotional, cognitive and motivational experience).
p. 89	“kennen die Begriffe »Klasse«, »Objekt«, »Attribut« und »Attributwert« und benutzen sie in Anwendungssituationen” (Gesellschaft für Informatik (GI) 2008, p. 14)	understand the terms “class”, “object”, “attribute”, and “attribute value” and apply them.
p. 89	“kennen Änderungsmöglichkeiten für Attributwerte von Objekten in altersgemäßen Anwendungen und reflektieren, wie sie die Informationsdarstellung unterstützen” (Gesellschaft für Informatik (GI) 2008, p. 15)	understand the manipulation capabilities for attribute values of objects in age-appropriate applications and reflect how they support the presentation of information.

Page	Original Source	Translation
p. 89	“stellen Objekte der jeweiligen Anwendung in einer geeigneten Form dar” (Gesellschaft für Informatik (GI) 2008, p. 16)	represent the objects of the respective application in a suitable form.
p. 89	“identifizieren Objekte in Informatiksystemen und erkennen Attribute und deren Werte” (Gesellschaft für Informatik (GI) 2008, p. 19)	identify objects in informatics systems and identify attributes and their values.
p. 89	“erstellen Diagramme und Grafiken zum Veranschaulichen einfacher Beziehungen zwischen Objekten der realen Welt” (Gesellschaft für Informatik (GI) 2008, p. 22)	create charts and graphs to illustrate simple associations between real-world objects.
p. 89	“entwickeln für einfache Sachverhalte objektorientierte Modelle und stellen diese mit Klassendiagrammen dar” (Gesellschaft für Informatik (GI) 2008, p. 19)	develop object-oriented models for simple issues and represent them with class diagrams.
p. 91	 Argumentieren Modellieren Implementieren Kompetenz-bereiche Darstellen und interpretieren Kommunizieren und Kooperieren Daten und ihre Strukturierung Algorithmen Inhaltsfelder Formale Sprachen und Automaten Informatiksysteme Informatik, Mensch und Gesellschaft (Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen 2014)	 argue model implement competency area represent and interpret communicate and cooperate data and their structure algorithms content area formal languages and automata informatics systems informatics, man, and society
p 91	• “ermitteln bei der Analyse einfacher Problemstellungen Objekte, ihre Eigenschaften, ihre Operationen und ihre Beziehungen (M),	• identify objects, their properties, their operations and their relationships in the analysis of simple problems (M),

Page	Original Source	Translation
p 91	• modellieren Klassen mit ihren Attributen, ihren Methoden und Assoziationsbeziehungen (M),	• model classes with their attributes, methods and associations (M),
	• modellieren Klassen unter Verwendung von Vererbung (M),	• model classes using inheritance (M),
	• ordnen Attributen, Parametern und Rückgaben von Methoden einfache Datentypen, Objekttypen oder lineare Datensammlungen zu (M),	• assign simple data types, object types, or linear data structures to attributes, parameters, and return values of methods (M),
	• ordnen Klassen, Attributen und Methoden ihren Sichbarkeitsbereich zu (M),	• assign a visibility to classes, attributes, and methods (M),
	• stellen den Zustand eines Objekts dar (D),	• represent the state of an object (D),
	• stellen die Kommunikation zwischen Objekten grafisch dar (M),	• graphically represent the communication between objects (M),
	• stellen Klassen, Assoziations- und Vererbungsbeziehungen in Diagrammen grafisch dar (D),	• represent classes, associations and inheritance in diagrams (D),
	• dokumentieren Klassen durch Beschreibung der Funktionalität der Methoden (D),	• document classes by description of the functionality of the methods (D),
	• analysieren und erläutern eine objektorientierte Modellierung (A),	• analyze and explain an object-oriented modeling (A),

Page	Original Source	Translation
p 91	• implementieren Klassen in einer Programmiersprache auch unter Nutzung dokumentierter Klassenbibliotheken (I)" (Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen 2014)	• implement classes in a programming language including documented class libraries (I).
p. 92	• ermitteln bei der Analyse von Problemstellungen Objekte, ihre Eigenschaften, ihre Operationen und ihre Beziehungen (M),	• identify objects, their properties, their operations and their associations in the analysis of problems (M),
	• stellen lineare und nichtlineare Strukturen grafisch dar und erläutern ihren Aufbau (D),	• graphically represent linear and non-linear structures and explain them (D),
	• modellieren Klassen mit ihren Attributen, Methoden und ihren Assoziationsbeziehungen unter Angabe von Multiplizitäten (M),	• model classes with their attributes, methods and associations specifying multiplicities (M),
	• modellieren abstrakte und nicht abstrakte Klassen unter Verwendung von Vererbung durch Spezialisieren und Generalisieren (M),	• model abstract and non-abstract classes using inheritance by specializing and generalizing (M),
	• ordnen Attributen, Parametern und Rückgaben von Methoden einfache Datentypen, Objekttypen sowie lineare und nichtlineare Datensammlungen zu (M),	• assign simple data types, object types, and linear and nonlinear data structures to attributes, parameters, and return values of methods (M),

Page	Original Source	Translation
p. 92	• verwenden bei der Modellierung geeigneter Problemstellungen Möglichkeiten der Polymorphie (M),	• use possibilities of polymorphism in the modeling of appropriate problems (M),
	• ordnen Klassen, Attributen und Methoden ihre Sichtbarkeitsbereiche zu (M),	• assign a visibility to classes, attributes, and methods (M),
	• stellen die Kommunikation zwischen Objekten grafisch dar (D),	• graphically represent the communication between objects (D),
	• stellen Klassen und ihre Beziehungen in Diagrammen grafisch dar (D),	• represent classes and their associations in diagrams (D),
	• dokumentieren Klassen (D),	• document classes (D),
	• analysieren und erläutern objekorientierte Modellierungen (A),	• analyze and explain object-oriented modeling (A),
	• implementieren Klassen in einer Programmiersprache auch unter Nutzung dokumentierter Klassenbibliotheken (I).” (Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen 2014)	• implement classes in a programming language including documented class libraries (I).

Page	Original Source	Translation
p. 93	Überlegungen zur Struktur von Graphiken führen zur objektorientierten Sichtweise. Die Schüler erkennen, dass jedes Objekt der Graphik bestimmte Eigenschaften hat und einer Klasse gleichartiger Objekte zugeordnet ist. • Objekte einer Vektorgraphik: Attribut, Attributwert und Methode • Beschreibung gleichartiger Objekte durch Klassen: Rechteck, Ellipse, Textfeld, Linie (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	Considerations about the structure of graphs lead to the object-oriented perspective. The students recognize that each of the graphics objects has certain properties and is associated with a class of similar objects. • Objects of a vector graphic: attribute, attribute value, and method • Description of similar objects by class: line, rectangle, ellipse, text box
p. 94	Bei der praktischen Arbeit mit Textverarbeitungssoftware wird das Verständnis für diese Begriffe vertieft; dabei zeigt sich, dass einzelne Objekte miteinander in Beziehung stehen können. Die Schüler erkennen, dass viele alltägliche Zusammenhänge ebenfalls durch Beziehungen zwischen Objekten beschrieben werden können, diese Begriffe also eine allgemeinere Bedeutung haben. • Verbesserung der Informationsdarstellung durch geeignetes Ändern von Attributwerten • die Enthält-Beziehung zwischen Objekten; Entwerfen von Objekt- und Klassendiagrammen (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	The understanding of these concepts is deepened in practical work with word processing software; it is shown that individual objects can be associated with each other. The students recognize that many everyday connections can also be described by relations between objects, so these terms have a more general meaning. • Improve the representation of information by changing the appropriate attribute values • The contains-association between objects; design of object and class diagrams

Page	Original Source	Translation
p. 94	Verschiedenartige Animationen, wie sie Präsentationssoftware zur Gestaltung bietet, helfen den Schülern, das Prinzip der Methoden von Objekten besser zu verstehen. (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	Various animations, such as presentation software provides for the design, help students to understand the principle of the method.
p. 94	Sie erkennen, dass hierarchische Ordnungen durch die Enthält-Beziehung zwischen Objekten der gleichen Klasse ermöglicht werden. • erweiterte Anwendung der Enthält-Beziehung: Ordner enthalten Ordner (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	They realize that hierarchical orders are enabled by the contains-association between objects of the same class. • Advanced application of the contains-association: folder containing folders
p. 94	Die Schüler erfahren, dass inhaltliche Zusammenhänge zwischen Dokumenten zu vernetzten Strukturen führen können, für die eine hierarchische Darstellung nicht ausreicht. • die Beziehung “verweist auf” zwischen Objekten (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	The students will learn that content relationships between documents may lead to networked structures, for which a hierarchical representation is not enough. • The association ‘refers to’ between objects

Page	Original Source	Translation
p. 94	Dabei erkennen sie, dass die Struktur der Klassen sowie deren Beziehungen sehr übersichtlich in Klassendiagrammen dargestellt werden können. Um das Modell nutzen und seine Brauchbarkeit überprüfen zu können, realisieren sie es mit einem relationalen Datenbanksystem. • Objekt (Entität), Klasse, Attribut und Wertebereich • Beziehungen zwischen Klassen, Kardinalität, graphische Darstellung • Realisierung von Objekten, Klassen und Beziehungen in einem relationalen Datenbanksystem: Datensatz, Tabelle, Wertebereich, Schlüsselkonzept (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	They realize that the structure of classes and their associations can be represented very clearly in class diagrams. To take advantage of the model and check its usefulness, they implement it with a relational database system. • Object (entity), class, attribute and value range • Associations between classes, cardinality, graphical representation • Implementation of objects, classes, and associations in a relational database system: data set, table, range of values, concept of keys
p. 95	Unter Verwendung einer geeigneten Entwicklungsumgebung für die objektorientierte Modellierung wiederholen und präzisieren die Schüler anhand von einfachen Beispielen die bekannten Begriffe und Notationen der objektorientierten Sichtweise. Dabei wird ihnen deutlich, dass Objekte im Wesentlichen eine Einheit aus Attributen und Methoden darstellen. • Objekt als Kombination aus Attributen und Methoden • graphische Darstellung von Klassen und Objekten, Beschreibung statischer Beziehungen durch Objekt- bzw. Klassendiagramme (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	By using a suitable development environment for object-oriented modeling the students repeat and clarify the known terms and notations of the object-oriented perspective with simple examples. This will clarify that objects essentially represent a unit of attributes and methods. • Object as a combination of attributes and methods • Graphical representation of classes and objects, description of static associations through object or class diagrams

Page	Original Source	Translation
p. 95	Die Schüler lernen, die Veränderungen von Objekten mithilfe von Zuständen und Übergängen zu beschreiben sowie mit Zustandsübergangsdiagrammen zu dokumentieren. Bei der Umsetzung dieser Zustandsmodelle in objektorientierte Programme legen sie die Zustände durch Werte von Attributen (Variablen) fest und ordnen den Übergängen Methodenaufrufe zu. • Zustand von Objekten: Festlegung durch Zustände der Attribute, Zustandsübergang durch Wertzuweisung • Lebenszyklus von Objekten von der Instanzierung über die Initialisierung bis zur Freigabe (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	The students learn to describe the changes of objects using states and transitions, as well as to document with state diagrams. By implementing these state models in object-oriented programs, they set the states by values of attributes (variables) and assign method calls to the transitions. • State of objects: Determination by states of attributes, state transition by value assignment • Life cycle of objects from the instantiation to initialization to release

Page	Original Source	Translation
p. 95	Die Schüler erkennen, dass wesentliche Abläufe eines Systems auf der Kommunikation zwischen seinen Objekten basieren. Für die vollständige Beschreibung müssen neben den bereits kennengelernten statischen auch die dynamischen Beziehungen zwischen Objekten bzw. Klassen erfasst werden. Hierfür lernen die Jugendlichen geeignete graphische Notationen kennen und erarbeiten Möglichkeiten zur Realisierung der Beziehungen in einer Programmiersprache. • Kommunikation zwischen Objekten durch Aufruf von Methoden; Interaktionsdiagramme; Datenkapselung • Realisierung der Enthält-Beziehung, Referenzen auf Objekte (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	The students recognize that essential processes of a system are based on communication between its objects. For the full description, the dynamic associations between objects or classes have to be learned in addition to the already learned static ones. For this purpose, the young people get to know appropriate graphical notations and develop possibilities for the implementation of these associations in a programming language. • Communication between objects by calling methods; Interaction diagrams; Data encapsulation • Implementation of the contains-association, references to objects

Page	Original Source	Translation
p. 95	Die Jugendlichen verwenden hierarchische Strukturen zur Ordnung ihrer Erfahrungswelt. Sie erkennen, dass sich diese oft durch eine spezielle Art von Beziehungen zwischen den Klassen eines Modells darstellen lassen. Die Schüler lernen hier das Prinzip der Vererbung kennen und wenden es an. Sie beschäftigen sich insbesondere mit der Möglichkeit einer zunehmenden Spezialisierung durch Veränderung ererbter Methoden. • Generalisierung bzw. Spezialisierung durch Ober- bzw. Unterklassen, Abbildung in Klassendiagramme, Vererbung • Polymorphismus und Überschreiben (overriding) von Methoden (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	Young people use hierarchical structures to order their realm of experience. They realize that these often can be represented through a special kind of association between the classes of a model. The students learn the concept of inheritance and apply it. In particular, they deal with the possibility of increasing specialization by changing inherited methods. • Generalization and specialization by super- or subclasses, representation in class diagrams, inheritance • Polymorphism and overriding methods

Page	Original Source	Translation
p. 96	Eine erste Implementierung mit einem Feld zeigt schnell die Grenzen dieser statischen Lösung auf und führt die Jugendlichen zu einer dynamischen Datenstruktur wie der einfach verketteten Liste. Sie erarbeiten deren prinzipielle Funktionsweise sowie deren rekursiven Aufbau und wenden hierbei das Prinzip der Referenz auf Objekte an. Die Jugendlichen erkennen, dass die rekursive Struktur der Liste für viele ihrer Methoden einen rekursiven Algorithmus nahelegt. Sie verstehen, dass eine universelle Verwendbarkeit der Klasse Liste nur möglich ist, wenn auf eine klare Trennung von Struktur und Daten geachtet wird. • Implementierung einer einfach verketteten Liste als Klasse mittels Referenzen unter Verwendung eines geeigneten Softwaremusters (Composite); Realisierung der Methoden zum Einfügen, Suchen und Löschen (Bayerisches Staatsministerium für Unterricht und Kultus 2004)	A first implementation with an array quickly shows the limits of this static solution and leads the youth to a dynamic data structure such as the simply linked list. They learn its principle functionality, as well as the recursive structure and apply the concept of reference on objects. The youth realize that the recursive structure of the list suggests a recursive algorithm for many of its methods. They understand that a universal applicability of the class list is only possible if attention is paid on a clear separation of structure and data. • Implementation of a simply linked list using references by a suitable software pattern (composite); Implementation of the methods to add, find, and delete

References

Abelson, H., Sussman, G. J. and Sussman, J. (1996), *Structure and interpretation of computer programs*, 2nd edition, MIT Press, Cambridge.

ACM/IEEE-CS Joint Task Force on Computing Curricula (2013), Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science.
URL: `http://www.acm.org/education/CS2013-final-report.pdf` (accessed 05.12.2014)

Adams, J. and Frens, J. (2003), Object centered design for Java: Teaching OOD in CS-1, *in* Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, February 19-23 2003, ACM Press, New York, pp. 273–277.

Albert, D. and Steiner, C. M. (2005), Empirical Validation of Concept Maps: Preliminary Methodological Considerations, *in* Proceedings of the 5th IEEE International Conference on Advanced Learning Technologies, Kaohsiung, July 5 - 8 2005, IEEE Press, Los Alamitos, pp. 952–953.

Alphonce, C. and Ventura, P. (2002), Object orientation in CS1-CS2 by design, *in* Proceedings of the 7th annual conference on Innovation and technology in computer science education, Aarhus, Juni 24-26 2002, ACM Press, New York, pp. 70–74.

Anderson, J. R. (2005), *Cognitive psychology and its implications*, 6th edition, Worth Publishers, New York.

Anderson, L. W. and Krathwohl, D. R. (2009), *A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives*, Longman, New York.

Armstrong, D. J. (2006), The quarks of object-oriented development, *COMMUNICATIONS OF THE ACM* **49**(2), 123–128.

Ayala, R. J. d. (2009), *The theory and practice of item response theory*, Methodology in the social sciences, Guilford Press, New York.

Bandura, A. (1977), *Social learning theory*, Prentice Hall, Englewood Cliffs, N.J.

Bandura, A. (1986), *Social foundations of thought and action: A social cognitive theory*, Prentice-Hall, Englewood Cliffs, N.J.

Bandura, A. (1989), Social cognitive theory, *in* R. Vasta, ed., Annals of child development, Vol. 6 of *Six theories of child development*, CT: JAI Press, Greenwich, pp. 1–60.

Bandura, A. (1997), *Self-efficacy: The exercise of control*, W.H.Freeman & Co Ltd, New York.

Bandura, A. (2001), Social Cognitive Theory: An Agentic Perspective, *Annual Review of Psychology* **52**(1), 1–26.

Bartholomew, D. J. (2008), *Analysis of multivariate social science data*, Chapman & Hall/CRC statistics in the social and behavioral sciences series, 2nd edition, CRC Press, Boca Raton.

Bayerisches Staatsministerium für Unterricht und Kultus (2004), Lehrplan für das Gymnasium in Bayern: Informatik.
URL: `http://www.isb-gym8-lehrplan.de/contentserv/3.1.neu/g8.de/index.php?StoryID=26172` (accessed 14.11.2014)

Ben-Ari, M. (1998), Constructivism in computer science education, *in* Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education, Atlanta, February 25- March 1 1998, ACM, New York, pp. 257–261.

Bennedsen, J. and Caspersen, M. E. (2004), Programming in context: A model-first approach to CS1, *in* Proceedings of the 35th SIGCSE technical symposium on Computer science education, Norfolk, March 3–7 2004, ACM Press, New York, pp. 477–481.

Bennedsen, J. and Schulte, C. (2006), A Competence Model for Object-Interaction in Introductory Programming, *in* 18th Workshop of the Psychology of Programming Interest Group,, Vol. 18 of *Brighton*, September 7-8 2006, pp. 215–229.
URL: `http://www.ppig.org/papers/18th-bennedsen.pdf` (accessed 13.12.2014)

Bennedsen, J. and Schulte, C. (2008), What does ’objects-first’ mean? An international study of teachers’ perceptions of objects-first, *in* Seventh Baltic Sea Conference on Computing Education Research, Koli National Park, Finland, November 15-18 2007, Australian Computer Society, Inc, Darlinghurst, pp. 21–29.

Bennedsen, J. and Schulte, C. (2013), Object Interaction Competence Model v. 2.0, *in* Learning and Teaching in Computing and Engineering, Macau, March 22-24 2013, IEEE Press, Los Alamitos, pp. 9–16.

Berges, M. and Hubwieser, P. (2010), Vorkurse in objektorientierter Programmierung: Lösungsansatz für einen leichteren Einstieg in die Informatik, *in* D. Engbring, R. Keil, J. Magenheim and H. Selke, eds, Tagungsband der 4. Fachtagung zur "Hochschuldidaktik Informatik", Vol. 4 of *Paderborn*, Okt 09-10 2010, Universitätsverlag Potsdam, Potsdam, pp. 13–22.

Berges, M. and Hubwieser, P. (2012), Towards an Overview Map of Object-Oriented Programming and Design, *in* Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling ’12, Koli National Park, Finland, November 15–18 2012, ACM Press, New York, pp. 135–136.

Berges, M. and Hubwieser, P. (2013), Concept specification maps: displaying content structures, *in* Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, July 1-3 2013, ACM Press, New York, USA, pp. 291–296.

Berges, M. and Hubwieser, P. (2015), Evaluation of Source Code with Item Response Theory, *in* Proceedings of the 20th SIGCSE Conference on Innovation and Technology in Computer Science Education, Vilnius, July 6-8, ACM Press, New York.

Berges, M., Mühling, A. and Hubwieser, P. (2012), The Gap Between Knowledge and Ability, *in* Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling ’12, Koli National Park, Finland, November 15–18 2012, ACM Press, New York, pp. 126–134.

Bergin, J., Bruce, K. and Kölling, M. (2005), Objects-early tools: a demonstration, *in* Proceedings of the 36th SIGCSE technical symposium on Computer science education, St. Louis, Missouri, February 23-27 2005, ACM, New York, pp. 390–391.

Bergin, S. and Reilly, R. (2005*a*), Programming: factors that influence success, *in* Proceedings of the 36th SIGCSE technical symposium on Computer science education, Vol. 37 of *St. Louis, Missouri*, February 23-27 2005, ACM, New York, pp. 411–415.

Bergin, S. and Reilly, R. (2005*b*), The influence of motivation and comfort-level on learning to program, *in* 17th Workshop of the Psychology of Programming Interest Group, Brighton, June 29-July 1 2005.
URL: `http://www.ppig.org/workshops/17th-programme.html` (accessed 13.12.2014)

Black, A. P. (2013), Object-oriented programming: some history, and challenges for the next fifty years, *Journal Information and Computation* **231**, 3–20.

Blair, G. (1991), *Object-oriented languages, systems and applications*, Pitman, London.

Bonar, J. and Soloway, E. (1983), Uncovering principles of novice programming, *in* Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Austin, Texas, 1983, ACM Press, New York, pp. 10–13.

Bonar, J. and Soloway, E. (1985), Preprogramming knowledge: a major source of misconceptions in novice programmers, *Human-Computer Interaction* **1**(2), 133–161.

Bond, T. G. and Fox, C. M. (2007), *Applying the Rasch model: Fundamental measurement in the human sciences*, 2 edition, Lawrence Erlbaum Associates Publishers, Mahwah.

Booch, G. (1994), *Object-oriented analysis and design with applications*, The Benjamin/Cummings series in object-oriented software engineering, 2nd edition, Benjamin/Cummings Pub. Co., Redwood City, Calif.

Booch, G., Rumbaugh, J. and Jacobson, I. (1999), *The unified modeling language user guide*, Addison-Wesley, Reading.

Borge, R. E. (2004), Teaching OOP using graphical programming environments: An expermental study, PhD thesis, University of Oslo, Oslo.

Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C. and Thomas, L. (2009), An evaluation of object oriented example programs in introductory programming textbooks, *in* Proceedings of the 14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Vol. 41 of *Paris*, July 6-9 2009, ACM Press, New York, pp. 126–143.

Börstler, J., Henrik B. Christensen, Jens Bennedsen, Nordström, M., Lena Kallin Westin, Moström, J. E. and Michael E. Caspersen (2008), Evaluating OO example programs for CS1, *in* Proceedings of the 13th annual conference on Innovation and technology in computer science education, Madrid, June 30 - July 2 2008, ACM Press, New York, pp. 47–52.

Börstler, J. and Sperber, M. (2010), Systematisches Programmieren in der Anfängerausbildung, *Informatica Didactica* **8**.
URL: `http://www.informatica-didactica.de/cmsmadesimple/index.php?page=Boerstler2010` (accessed 15.12.2014)

Bortz, J. and Schuster, C. (2010), *Statistik für Human- und Sozialwissenschaftler*, 7th edition, Springer, Berlin.

Boytchev, P. (2011), Wild Programming – One Unintended Experiment with Inquiry Based Learning, *in* I. Kalaš and R. Mittermeir, eds, Informatics in Schools. Contributing to 21st Century Education, Vol. 7013 of *Bratislava*, Oct. 26-29 2011, Springer, Berlin, Heidelberg, pp. 1–8.

Brinda, T., Puhlmann, H. and Schulte, C. (2009), Bridging ICT and CS: Educational Standards for Computer Science in Lower Secondary Education, *in* Proceedings of the 14th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, Paris, July 6-9 2009, ACM Press, New York, pp. 288–292.

Brooks, R. E. (1980), Studying programmer behavior experimentally: the problems of proper methodology, *COMMUNICATIONS OF THE ACM* **23**(4), 207–213.

Brown, J. S., Collins, A. and Duguid, P. (1989), Situated Cognition and the Culture of Learning, *Educational Researcher* **18**(1), 32–42.

Broy, M. and Siedersleben, J. (2002), Objektorientierte Programmierung und Softwareentwicklung: Eine kritische Einschätzung, *Informatik-Spektrum* **25**(1), 3–11.

Bruce, C., Buckingham, L., Hynd, J., McMahon, C., Roggenkamp, M. and Stoodley, I. (2004), Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductory Programming Students at University, *Journal of Information Technology Education* **3**, 143–160.

Bruce, K. B., Danyluk, A. and Murtagh, T. (2001), A library to support a graphics-based object-first approach to CS 1, *in* Proceedings of the 32nd SIGCSE technical symposium on Computer science education, Vol. 33 of *Charlotte*, Feb 21-25 2001, ACM Press, New York, pp. 6–10.

Bruning, R. H., Schraw, G. J. and Norby, M. M. (2011), *Cognitive psychology and instruction*, 5th edition, Allyn & Bacon/Pearson, Boston.

Burnham, K. P. and Anderson, D. R. (2002), *Model selection and multimodel inference: A practical information-theoretic approach*, 2nd edition, Springer, New York.

Capretz, L. F. (2003), A brief history of the object-oriented approach, *SIGSOFT Software Engineering Notes* **28**(2), 6–15.

Cardelli, L. and Wegner, P. (1985), On Understanding Types, Data Abstraction, and Polymorphism, *ACM COMPUTING SURVEYS* **17**(4), 471–523.

Carey, S. (1985), *Conceptual change in childhood*, MIT Press series in learning, development, and conceptual change, MIT Press, Cambridge.

Carter, J., Bouvier, D., Cardell-Oliver, R., Hamilton, M., Kurkovsky, S., Markham, S., McClung, O. W., McDermott, R., Riedesel, C., Shi, J. and White, S. (2011), Motivating all our students?, *in* Proceedings of the 16th annual conference reports on Innovation and technology in computer science education - working group reports, Darmstadt, June 25-29 2011, ACM Press, New York, pp. 1–18.

Chandler, P. and Sweller, J. (1991), Cognitive Load Theory and the Format of Instruction, *Cognition and Instruction* **8**(4), 293–332.

Christensen, H. B. (2005), Implications of perspective in teaching objects first and object design, *in* Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, Monte de Caparica, June 27-29, ACM Press, New York, pp. 94–98.

Clancy, M. (2004), Misconceptions and Attitudes that Interfere with Learning to Program, *in* S. Fincher, ed., Computer Science Education Research, Taylor & Francis, London, pp. 85–100.

Clark, R. C., Nguyen, F. and Sweller, J. (2005), *Efficiency in learning: Evidence-based guidelines to manage cognitive load*, Pfeiffer, San Francisco.

Collins, A., Brown, J. S. and Newman, S. E. (1989), Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics, *in* L. B. Resnick, ed., Knowing, learning, and instruction, L. Erlbaum Associates, Hillsdale, N.J., pp. 453–493.

Compeau, D. R. and Higgins, C. A. (1995), Computer Self-Efficacy: Development of a Measure and Initial Test, *MIS Quarterly* **19**(2), 189–211.

Cooper, S., Dann, W. and Pausch, R. (2003), Teaching objects-first in introductory computer science, *in* Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, February 19-23 2003, ACM Press, New York, pp. 191–195.

Cottam, J. A., Menzel, S. and Greenblatt, J. (2011), Tutoring for retention, *in* Proceedings of the 42nd ACM technical symposium on Computer science education, Dallas, March 9-12 2011, ACM Press, New York, pp. 213–218.

Crow, D. (2014), Why every child should learn to code.
URL: `http://www.theguardian.com/technology/2014/feb/07/year-of-code-dan-crow-songkick` (accessed 10.11.2014)

Dahl, O.-J. (2002), The roots of object orientation: the Simula language, *in* M. Broy and E. Denert, eds, Software Pioneers, Springer-Verlag, New York, pp. 78–90.

Dahl, O.-J. and Nygaard, K. (1966), SIMULA: an ALGOL-based simulation language, *COMMUNICATIONS OF THE ACM* **9**(9), 671–678.

Dahl, O.-J. and Nygaard, K. (1967), Class and subclass declarations, *in* J. N. Buxton, ed., Simulation Programming Languages, Oslo, May 1967, Amsterdam, pp. 158–174.

Dahncke, H. and Reiska, P. (2008), Testing Achievement with Concept Mapping in School Physics, *in* Proceedings of the Third International Conference on Concept Mapping, Tallinn, Sept 22-25 2008, pp. 403–410.

Daly, J., Brooks, A., Miller, J., Roper, M. and Wood, M. (1996), Evaluating inheritance depth on the maintainability of object-oriented software, *Empirical Software Engineering* **1**(2), 109–132.

Dangwal, R. and Mitra, S. (2005), A Model of How Children Acquire Computing Skills from Hole-in-the-Wall Computers in Public Places, *Information Technologies and International Development* **2**(4), 41–60.

Davier, M. v. (1997), Bootstrapping Goodness-of-Fit Statistics for Sparse Categorical Data: Results of a Monte Carlo Study, *Methods of Psychological Research Online* **2**(2), 29–48.

Decker, A. (2003), A tale of two paradigms, *Journal of Computing Sciences in Colleges* **19**(2), 238–246.

Decker, R. and Hirshfield, S. (1994), The top 10 reasons why object-oriented programming can't be taught in CS 1, *in* Proceedings of the 25th SIGCSE symposium on Computer science education, Phoenix, March 10-12 1994, ACM Press, New York, pp. 51–55.

DeClue, T. (1996), Object-orientation and the principles of learning theory: a new look at problems and benefits, *in* Proceedings of the 27th SIGCSE technical symposium on Computer science education, Philadelphia, Feb 15-18 1996, ACM Press, New York, pp. 232–236.

Deitel, P. J. and Deitel, H. M. (2012), *Java: How to program*, 9th edition, Prentice Hall, Upper Saddle River.

Dierbach, C., Taylor, B., Zhou, H. and Zimand, I. (2005), Experiences with a CS0 course targeted for CS1 success, *in* Proceedings of the 36th SIGCSE technical symposium on Computer science education, Vol. 37 of *St. Louis, Missouri*, February 23-27 2005, ACM, New York, pp. 317–320.

Diethelm, I. (2007), Strictly models and objects first": Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht, Dissertationsschrift, Universität Kassel, Kassel.

diSessa, A. A. (1993), Toward an Epistemology of Physics, *Cognition and Instruction* **10**(2-3), 105–225.

Drasutis, S., Motekaityte, V. and Noreika, A. (2010), A Method for Automated Program Code Testing, *Informatics in Education* **9**(2), 199–208.

du Boulay, B. (1988), Some Difficulties of Learning to Program, *in* E. Soloway and J. Spohrer, eds, Studying the novice programmer, Interacting With Computers : Iwc, L. Erlbaum Associates, Mahwah, pp. 283–299.

Duit, R. and Treagust, D. F. (2003), Conceptual change: A powerful framework for improving science teaching and learning, *International Journal of Science Education* **25**(6), 671–688.

Eckel, B. (2006), *Thinking in Java*, 4th edition, Prentice Hall, Upper Saddle River.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K. and Zander, C. (2006), Putting threshold concepts into context in computer science education, *in* Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, Bologna, June 26-28 2006, ACM Press, New York, pp. 103–107.

Edmondson, C. (2009), Proglets for first-year programming in Java, *SIGCSE Bulletin* **41**(2), 108–112.

Edmondson, K. M. (2005), Assessing science understanding through concept maps, *in* J. J. Mintzes, ed., Assessing science understanding, Educational psychology, Elsevier, Amsterdam, pp. 15–40.

Ehlert, A. (2012), Empirische Studie: Unterschiede im Lernerfolg und Unterschiede im subjektiven Erleben des Unterrichts von Schülerinnen und Schülern im Informatik-Anfangsunterricht (11. Klasse Berufliches Gymnasium) in Abhängigkeit von der zeitlichen Reihenfolge der Themen (OOP-First und OOP-Later), Dissertationsschrift, Freie Universität Berlin, Berlin.

Ehlert, A. and Schulte, C. (2009*a*), Empirical comparison of objects-first and objects-later, *in* Proceedings of the 5th international workshop on Computing education research workshop, Berkeley, Aug 10-11 2009, ACM Press, New York, pp. 15–26.

Ehlert, A. and Schulte, C. (2009*b*), Unterschiede im Lernerfolg von Schülerinnen und Schülern in Abhängigkeit von der zeitlichen Reihenfolge der Themen (OOP-First bzw. OOP-Later), *in* Tagungsband zur 13. GI-Fachtagung Informatik und Schule, Berlin, Sept 21-24 2009, Gesellschaft für Informatik, Bonn, pp. 121–132.

Everitt, B. (2011), *Cluster analysis*, Wiley series in probability and statistics, 5th edition, Wiley, Chichester.

Faessler, L., Hinterberger, H., Dahinden, M. and Wyss, M. (2006), Evaluating student motivation in constructivistic, problem-based introductory computer science courses, *in* Thomas Reeves and Shirley Yamashita, eds, Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2006, Honolulu, Oct 13-17 2006, AACE, Honolulu, pp. 1178–1185.

Feldgen, M. and Clua, O. (2003), New motivations are required for freshman introductory programming, *in* Proceedings to 33rd Frontiers in Education Conference, Vol. 1 of *Westminster*, Nov 5-8 2003, IEEE Press, Westminster, p. 24.

Fischer, G. H. and Molenaar, I. W. (1995), *Rasch Models: Foundations, recent developments, and applications*, Springer, New York.

Flanagan, D. (2005), *Java in a nutshell*, 5th edition, O'Reilly, Sebastopol.

Fleury, A. E. (2000), Programming in Java: student-constructed rules, *in* Proceedings of the 31st SIGCSE technical symposium on Computer science education, Vol. 32 of *Austin*, March 8-12 2000, ACM Press, New York, pp. 197–201.

Fraley, C. and Raftery, A. E. (2000), Model-Based Clustering, Discriminant Analysis, and Density Estimation, *Journal of the American Statistical Association* **97**(458), 611–631.

Fraley, C., Raftery, A. E., Murphy, T. B. and Scrucca, L. (2012), mclust Version 4 for R: Normal Mixture Modeling for Model-Based Clustering, Classification, and Density Estimation, Technical Report 597, University of Washington, Seattle.

Frey, E., Hubwieser, P. and Winhard, F. (2004), *Informatik 1: Objekte, Strukturen, Algorithmen*, Klett, Stuttgart.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C. and Thompson, E. (2007), Developing a computer science-specific learning taxonomy, *SIGCSE Bulletin* **39**(4), 152–170.

Garrigue, J. (1998), Programming with polymorphic variants, *in* The 1998 ACM SIGPLAN Workshop on ML, Baltimore, Sept 26 1998, ACM Press, New York.

Garrigue, J. (2000), Code reuse through polymorphic variants, *in* Workshop on Foundations of Software Engineering, Sasaguri, Nov 2000.
URL: `http://www.math.nagoya-u.ac.jp/~garrigue/papers/fose2000.html` (accessed 15.12.2014)

Garrison, D. R. (1997), Self-Directed Learning: Toward a Comprehensive Model, *Adult Education Quarterly* **48**(1), 18–33.

Gerjets, P., Scheiter, K. and Cierniak, G. (2009), The Scientific Value of Cognitive Load Theory: A Research Agenda Based on the Structuralist View of Theories, *Educational Psychology Review* **21**(1), 43–54.

Gesellschaft für Informatik (GI) (2008), Grundsätze und Standards für die Informatik in der Schule: Bildungsstandards Informatik für die Sekundarstufe I, *LOGIN* **28**(150/151).

Giannakos, M. N., Hubwieser, P. and Ruf, A. (2012), Is Self-efficacy in Programming Decreasing with the Level of Programming Skills?, *in* Proceedings of the 7th Workshop in Primary and Secondary Computing Education, Hamburg, Nov 8-9 2012, ACM, New York, pp. 16–21.

Gill, T. G. and Holton, C. F. (2006), A Self-Paced Introductory Programming Course, *Journal of Information Technology Education* **5**, 95–105.

Glaserfeld, E. v. (1983), Learning as Constructive Activity, *in* Proceedings of the 5th Annual Meeting of the North American Group of Psychology in Mathematics Education, Montreal, pp. 41–101.

Glasersfeld, E. v. (1989*a*), Cognition, construction of knowledge, and teaching, *Synthese* **80**(1), 121–140.

Glasersfeld, E. v. (1989*b*), Constructivism in Education, *in* T. Husén, ed., The international encyclopedia of education, Vol. 1: Supplement, Pergamon Pr, Oxford, pp. 162–163.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C. and Zilles, C. (2008), Identifying important and difficult concepts in introductory computing courses using a delphi process, *in* Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, Mar 12-15 2008, ACM Press, New York, pp. 256–260.

Goldsmith, T. E. and Davenport, D. M. (1990), Assessing structural similarity of graphs, *in* R. W. Schvaneveldt, ed., Pathfinder associative networks, Ablex Publishing Corp, Norwood, New Jersey, pp. 75–87.

Gower, J. C. (1971), A General Coefficient of Similarity and Some of Its Properties, *Biometrics* **27**(4), p 857–871.

Greening, T. (1998), Computer science: through the eyes of potential students, *in* Proceedings of the 3rd Australasian conference on Computer science education, Brisbane, July 08-10 1998, ACM Press, New York, pp. 145–154.

Hadar, I. and Hadar, E. (2007), An iterative methodology for teaching object oriented concepts, *Informatics in Education* **6**(1), 67–80.

Hadjerrouit, S. (1999), A constructivist approach to object-oriented design and programming, *in* Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education, Vol. 31 of *Cracow*, 6/99, ACM Press, New York, pp. 171–174.

Hanks, B. (2007), Problems encountered by novice pair programmers, *in* Proceedings of the 3rd international workshop on Computing education research, Atlanta, Sept 15-16 2007, ACM Press, New York, pp. 159–164.

Hanks, B., McDowell, C., Draper, D. and Krnjajic, M. (2004), Program quality with pair programming in CS1, *in* Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, Vol. 36 of *Leeds*, June 28-30 2004, ACM Press, New York, pp. 176–180.

Hanks, B., Murphy, L., Simon, B., McCauley, R. and Zander, C. (2009), CS1 students speak: advice for students by students, *in* Proceedings of the 40th ACM technical symposium on Computer science education, Chattanooga, Mar 4-7 2009, ACM Press, New York, pp. 19–23.

Hansen, S. A. (2009), Analyzing programming projects, *in* Proceedings of the 40th ACM technical symposium on Computer science education, Chattanooga, Mar 4-7 2009, ACM Press, New York, pp. 377–381.

Hares, J. S. and Smart, J. D. (1994), *Object orientation: Technology, techniques, management, and migration*, Wiley professional computing, Wiley, Chichester.

Henderson-Sellers, B. (1992), *A book of object-oriented knowledge - Object-oriented analysis, design, and implementation: a new approach to software engineering*, Prentice Hall object-oriented series, Prentice Hall, New York.

Hewson, P. W. (1981), A Conceptual Change Approach to Learning Science, *European Journal of Science Education* **3**(4), 383–396.

Holland, S., Griffiths, R. and Woodman, M. (1997), Avoiding object misconceptions, *in* Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education, Vol. 29 of *San Jose*, Feb 27 - Mar 1 1997, ACM Press, New York, pp. 131–134.

Hristakeva, M. and Vuppala, R. (2009), A Survey of Object Oriented Programming Languages, Technical report, University of California, Santa Cruz, California.
URL: `http://users.soe.ucsc.edu/~vrk/Reports/oopssurvey.pdf` (accessed 16.12.2014)

Hu, C. (2011), When to inherit a type: what we do know and what we might not, *ACM Inroads* **2**(2), 52–58.

Hubwieser, P. (2006), Functions, objects and states: Teaching informatics in secondary schools, *in* R. Mittermeir, ed., Informatics Education – The Bridge between Using and Understanding Computers, Vol. 4226 of *Vilnius*, Nov 7-11 2006, Springer, Berlin, pp. 104–116.

Hubwieser, P. (2007*a*), A smooth way towards object oriented programming in secondary schools, *Informatics, Mathematics and ICT: a'golden triangle'. IFIP WG* **3**.

Hubwieser, P. (2007*b*), *Didaktik der Informatik: Grundlagen, Konzepte, Beispiele*, EXamen.press, 3rd edition, Springer, Berlin.

Hubwieser, P. (2007*c*), *Tabellenkalkulationssysteme, Datenbanken*, Vol. 2 of *Informatik*, Klett, Stuttgart.

Hubwieser, P. (2008*a*), *Algorithmen, objektorientierte Programmierung, Zustandsmodellierung*, Vol. 3 of *Informatik*, Klett, Stuttgart.

Hubwieser, P. (2008*b*), Analysis of Learning Objectives in Object Oriented Programming, *in* R. T. Mittermeir and M. M. Syslo, eds, Proceedings of the 3rd international conference on Informatics in Secondary Schools - Evolution and Perspectives, Vol. 5090 of *Torun*, July 1-4 2008, Springer-Verlag, Berlin, pp. 142–150.

Hubwieser, P. (2009), *Rekursive Datenstrukturen, Softwaretechnik*, Vol. 4 of *Informatik*, Klett, Stuttgart.

Hubwieser, P. (2010), *Formale Sprachen, Kommunikation und Synchronisation von Prozessen, Funktionsweise eines Rechners, Grenzen der Berechenbarkeit*, Vol. 5 of *Informatik*, Klett, Stuttgart.

Hubwieser, P. (2012), Computer Science Education in Secondary Schools – The Introduction of a New Compulsory Subject, *ACM Transactions on Computing Education* **12**(4), 1–41.

Hubwieser, P. and Berges, M. (2011), Minimally invasive programming courses: learning OOP with(out) instruction, *in* Proceedings of the 42nd ACM technical symposium on Computer science education, Dallas, March 9-12 2011, ACM Press, New York, pp. 87–92.

Hubwieser, P. and Bitzl, M. (2010), Modeling Educational Knowledge. Supporting the Collaboration of Computer Science Teachers, *in* J. L. G. Dietz, ed., Proceedings of the 2nd International Conference on Knowledge Engineering and Ontology Development, Valencia, Oct 25-28 2010, SciTePress.

Hubwieser, P. and Mühling, A. (2011*a*), Investigating cognitive structures of object oriented programming, *in* Proceedings of the 16th annual joint conference on Innovation and technology in computer science education, Darmstadt, June 25-29 2011, ACM, New York, pp. 377–377.

Hubwieser, P. and Mühling, A. (2011*b*), Knowpats: Patterns of Declarative Knowledge - Searching Frequent Knowledge Patterns about Object-orientation, *in* J. Filipe and Ana L. N. Fred, eds, Proceedings of the International Conference on Knowledge Discovery and Information Retrieval, Paris, Oct 26-29 2011, SciTePress, pp. 358–364.

Hubwieser, P. and Mühling, A. (2011*c*), What students (should) know about object oriented programming, *in* Proceedings of the seventh international workshop on Computing education research, Providence, Aug 8-9 2011, ACM Press, New York, pp. 77–84.

Hubwieser, P. and Mühling, A. (2014), Competency Mining in Large Data Sets - Preparing Large Scale Investigations in Computer Science Education, *in* A. Fred and J. Filipe, eds, Proceedings of the 6th International KDIR - International Conference on Knowledge Discovery and Information Retrieval, Rome, Oct 14-16 2014, SciTePress, pp. 315–322.

Ifenthaler, D. (2006), Diagnose lernabhängiger Verändung mentaler Modelle: Entwicklung der SMD-Technologie als methodologisches Verfahren zur relationalen, struckturellen und semantischen Analyse individueller Modellkonstruktionen, Dissertationsschrift, Albert-Ludwigs-Universtät Freiburg im Breisgau, Freiburg im Breisgau.

Ifenthaler, D. (2010), Relational, structural, and semantic analysis of graphical representations and concept maps, *Educational Technology Research and Development* **58**, 81–97.

Isomöttönen, V., Tirronen, V. and Cochez, M. (2013), Issues with a course that emphasizes self-direction, *in* Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, July 1-3 2013, ACM Press, New York, USA, pp. 111–116.

Jacobson, I. (1992), *Object-oriented software engineering: A use case driven approach*, ACM Press, New York.

Jähnichen, S. and Herrmann, S. (2002), Was, bitte, bedeutet Objektorientierung?, *Informatik-Spektrum* **25**(4), 266–276.

Jakovljevic, M. (2003), Concept mapping and appropriate instructional strategies in promoting programming skills of holistic learners, *in* Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, pp. 308–315.

Jana, D. (2005), *Java and object-oriented programming paradigm*, Prentice-Hall of India, New Delhi.

Jenkins, T. (2002), On the Difficulty of Learning to Program, *in* 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, Loughbourgh, Aug 23 2002.

Kaufman, L. and Rousseeuw, P. J. (1990), *Finding groups in data: An introduction to cluster analysis*, Wiley series in probability and mathematical statistics. Applied probability and statistics, Wiley, New York.

Kay, A. C. (1993), The early history of Smalltalk, *in* The second ACM SIGPLAN conference on History of programming languages, Cambridge, Apr 20-23 1993, ACM Press, New York, pp. 69–95.

Kedar, S. V. (2007), *Programming paradigms and methodology*, 3rd edition, Technical Publications Pune, Pune, India.

Kelleher, C. and Pausch, R. (2005), Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers, *ACM COMPUTING SURVEYS* **37**(2), 83–137.

Kemkes, G., Vasiga, T. and Cormack, G. (2006), Objective Scoring for Computing Competition Tasks, *in* R. Mittermeir, ed., Informatics Education – The Bridge between Using and Understanding Computers, Vol. 4226 of *Vilnius*, Nov 7-11 2006, Springer, Berlin, pp. 230–241.

Keppens, J. and Hay, D. (2008), Concept map assessment for teaching computer programming, *Computer science education* **18**(1), 31–42.

Kern, C. and Crippen, K. J. (2008), Mapping for Conceptual Change, *The Science Teacher* **75**(6), 32–38.

Knowles, M. S. (1975), *Self-directed learning: A guide for learners and teachers*, Association Press, New York.

Knudsen, J. L. and Madsen, O. L. (1988), Teaching Object-Oriented Programming Is More than Teaching Object-Oriented Programming Languages, *in* Proceedings of the European Conference on Object-Oriented Programming, Vol. 322 of *Oslo*, Aug 1988, Springer-Verlag, London, pp. 21–40.

Kollee, C., Magenheim, J., Nelles, W., Rhode, T., Schaper, N., Schubert, S. and Stechert, P. (2009), Computer Science Education and Key Competencies, *in* 9th IFIP World Conference on Computers in Education, Vol. 302 of *Bento Goncalves*, July 27-31 2009, Springer, New York.

Koller, I., Alexandrowicz, R. and Hatzinger, R. (2012), *Das Rasch Modell in der Praxis*, Vol. 3786 of *utb-studi-e-book*, UTB GmbH, Stuttgart.

Koller, I. and Hatzinger, R. (2013), Nonparametric tests for the Rasch model: explanation, development, and application of quasi-exact tests for small samples, *InterStat* **11**, 1–16.

Kölling, M. (1999*a*), The problem of teaching object-oriented programming, Part I: Languages, *JOURNAL OF OBJECT-ORIENTED PROGRAMMING* **11**(8), 8–15.

Kölling, M. (1999*b*), The problem of teaching object-oriented programming, Part II: Environments, *JOURNAL OF OBJECT-ORIENTED PROGRAMMING* **11**(9), 6–12.

Kölling, M., Quig, B., Patterson, A. and Rosenberg, J. (2003), The BlueJ system and its pedagogy, *Journal of Computer Science Education* **13**(4), 249–268.

Kölling, M. and Rosenberg, J. (1996), An Object-Oriented Program Development Environment for the First Programming Course, *in* Proceedings of the 27th SIGCSE technical symposium on Computer science education, Philadelphia, Feb 15-18 1996, ACM Press, New York, pp. 83–87.

Kölling, M. and Rosenberg, J. (2001), Guidelines for teaching object orientation with Java, *in* Proceedings of the 6th annual conference on Innovation and technology in computer science education, ACM, Canterbury and United Kingdom, pp. 33–36.

Krippendorff, K. (2004), *Content analysis: An introduction to its methodology*, 2nd edition, Sage, Thousand Oaks, Calif.

Krippendorff, K. (2011), Computing Krippendorff's Alpha-Reliability, *Departmental Papers (ASC)* .
URL: `http://repository.upenn.edu/asc_papers/43` (accessed 16.12.2014)

Kuhn, T. S. (1996), *The structure of scientific revolutions*, 3rd edition, Univ. of Chicago Press, Chicago.

Kumar, D. A. and Annie, M. L. (2012), Clustering Dichotomous Data for Health Care, *International Journal of Information Sciences and Techniques (IJIST)* **2**(2), 23–33.

Lahtinen, E., Ala-Mutka, K. and Järvinen, H.-M. (2005), A study of the difficulties of novice programmers, *in* Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, Monte de Caparica, June 27-29, ACM Press, New York, pp. 14–18.

Lewis, J. (2000), Myths about Object-Orientation and its Pedagogy, *in* Proceedings of the 31st SIGCSE technical symposium on Computer science education, Austin, March 8-12 2000, ACM Press, New York, pp. 245–249.

Li, T. (2005), A General Model for Clustering Binary Data, *in* Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining, Chicago, Aug 21-24 2005, ACM Press, New York, pp. 188–197.

Linck, B., Ohrndorf, L., Schubert, S., Stechert, P., Magenheim, J., Nelles, W., Neugebauer, J. and Schaper, N. (2013), Competence model for informatics modelling and system comprehension, *in* IEEE Global Engineering Education Conference, Berlin, Mar 13-15 2013, IEEE Press, Los Alamitos, pp. 85–93.

Luker, P. A. (1989), Never mind the language, what about the paradigm?, *in* Proceedings of the twentieth SIGCSE technical symposium on Computer science education, Louisville, Feb 23-25 1989, ACM Press, New York, pp. 252–256.

Luker, P. A. (1994), There's more to OOP than syntax!, *in* Proceedings of the 25th SIGCSE symposium on Computer science education, Phoenix, March 10-12 1994, ACM Press, New York, pp. 56–60.

Luxton-Reilly, A., Denny, P., Kirk, D., Tempero, E. and Yu, S.-Y. (2013), On the differences between correct student solutions, *in* Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, July 1-3 2013, ACM Press, New York, USA, pp. 177–182.

Maechler, M. (2013), Package 'cluster'.
URL: `http://cran.r-project.org/web/packages/cluster/cluster.pdf` (accessed 25.09.2013)

Magenheim, J. (2005), Towards a Competence Model for Educational Standards of Informatics, *in* 8th IFIP World Conference on Computers in Education, Cape Town, July 1-7 2005, Springer, New York.

Magenheim, J., Nelles, W., Rhode, T. and Schaper, N. (2010), Towards a methodical approach for an empirically proofed competency model, *in* J. Hromkovič, R. Královič and J. Vahrenhold, eds, Teaching Fundamentals Concepts of Informatics, Vol. 5941 of *Zürich*, Jan 13-16 2010, Springer, Berlin, pp. 124–135.

Magenheim, J., Nelles, W., Rhode, T., Schaper, N., Schubert, S. and Stechert, P. (2010), Competencies for informatics systems and modeling: Results of qualitative content analysis of expert interviews, *in* IEEE Global Engineering Education Conference, Madrid, Apr 14-16 2010, IEEE Press, Los Alamitos, pp. 513–521.

Manaris, B. (2007), Dropping CS enrollments: or the emperor's new clothes?, *SIGCSE Bulletin* **39**(4), 6–10.

Maturana, H. R. and Varela, F. J. (1980), *Autopoiesis and cognition: The realization of the living*, Vol. 42 of *Boston studies in the philosophy of science*, D. Reidel Pub. Co., Dordrecht.

Mayring, P. (2010), *Qualitative Inhaltsanalyse: Grundlagen und Techniken*, Studium Paedagogik, 11th edition, Beltz, Weinheim.

McClure, J. R., Sonak, B. and Suen, H. K. (1999), Concept map assessment of classroom learning: Reliability, validity, and logistical practicality, *Journal of Research in Science Teaching* **36**(4), 475–492.

McDonald, R. P. (1965), Difficulty Factors and Non-linear Factor Analysis, *British Journal of Mathematical and Statistical Psychology* **18**(1), 11–23.

McDonald, R. P. and Ahlawat, K. S. (1974), Difficulty Factors in Binary Data, *British Journal of Mathematical and Statistical Psychology* **27**(1), 82–99.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S. and Thomas, L. (2006), A cognitive approach to identifying measurable milestones for programming skill acquisition, *in* Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, Bologna, June 26-28 2006, ACM Press, New York, pp. 182–194.

Meyer, B. (1987*a*), Eiffel: Programming for Reusability and Extendibility, *SIGPLAN Notices* **22**(2), 85–94.

Meyer, B. (1987*b*), Reusability - The Case for Object-Oriented Design, *IEEE SOFTWARE* **4**(2), 50–64.

Meyer, B. (1993), Towards an Object-Oriented Curriculum, *in* Proceedings of 11th international TOOLS conference, Santa Barbara, August 1993, Prentice Hall.

Meyer, B. (2006), Testable, Reusable Units of Cognition, *Computer* **39**(4), 20–24.

Meyer, B. (2009), *Object-oriented software construction*, 2nd edition, Prentice Hall, Upper Saddle River.

Miller, N. E. and Dollard, J. (1941), *Social Learning and Imitation*, Institute of human relations, Yale University. Institute of Human Relations.

Milne, I. and Rowe, G. (2002), Difficulties in Learning and Teaching Programming—Views of Students and Tutors, *Education and Information Technologies* **7**(1), 55–66.

Minarova, N. (2013), Programming Language Paradigms & The Main Principles of Object-Oriented Programming, *CRIS - Bulletin of the Centre for Research and Interdisciplinary Study* **2013**(2).

Ministerium für Schule und Weiterbildung des Landes Nordrhein-Westfalen (2014), Kernlehrplan für die Sekundarstufe II Gymnasium/Gesamtschule in Nordrhein-Westfalen: Informatik.
URL: `http://www.schulentwicklung.nrw.de/lehrplaene/upload/klp_SII/if/KLP_GOSt_Informatik.pdf` (accessed 14.11.2014)

Mitchell, W. (2000), A paradigm shift to OOP has occurred. . .implementation to follow, *Journal of Computing Sciences in Colleges* **16**(2), 94–105.

Mitra, S. (2000), Minimally invasive education for mass computer literacy, *in* Conference on Research in Distance and Adult Learning in Asia, Hong Kong, June 21-25 2000.

Moosbrugger, H. and Frank, D. (1992), *Clusteranalytische Methoden in der Persönlichkeitsforschung: Eine anwendungsorientierte Einführung in taxometrische Klassifikationsverfahren*, Vol. 12 of *Methoden der Psychologie*, 1st edition, Huber, Bern.

Mössenböck, H. (2014), *Sprechen Sie Java? Eine Einführung in das systematische Programmieren*, 5th edition, Dpunkt, Heidelberg.

Mühling, A. and Hubwieser, P. (2012), Towards Software-supported Large Scale Assessment of Knowledge Development, *in* Proceedings of the 12th Koli Calling International Conference on Computing Education Research - Koli Calling ’12, Koli National Park, Finland, November 15–18 2012, ACM Press, New York, pp. 145–146.

Mühling, A., Hubwieser, P. and Brinda, T. (2010), Exploring teachers’ attitudes towards object oriented modelling and programming in secondary schools, *in* Proceedings of the 6th international workshop on Computing education research, Aarhus, Aug 8-11 2010, ACM Press, New York, pp. 59–68.

Mühling, A. M. (2014), Investigating Knowledge Structures in Computer Science Education, Dissertationsschrift, Technische Universität München, München.

Müller, H. and Weichert, F. (2011), *Vorkurs Informatik: Der Einstieg ins Informatikstudium*, 2nd edition, Springer Fachmedien, Wiesbaden.

Neugebauer, J., Hubwieser, P., Magenheim, J., Ohrndorf, L., Schaper, N. and Schubert, S. (2014), Measuring Student Competences in German Upper Secondary Computer Science Education, *in* Y. Gülbahar and E. Karatas, eds, Proceedings of 7th International Conference on Informatics in Schools, Vol. 8730 of *Istanbul*, Sept 22-25 2014, Springer, Heidelberg, pp. 100–111.

Novak, J. D. (2002), Meaningful learning: The essential factor for conceptual change in limited or inappropriate propositional hierarchies leading to empowerment of learners, *Science Education* **86**(4), 548–571.

Novak, J. D. and Cañas, A. J. (2008), The Theory Underlying Concept Maps and How to Construct and Use Them, Technical report, Florida Institure for Human and Machine Cognition.
URL: `http://cmap.ihmc.us/Publications/ResearchPapers/TheoryUnderlyingConceptMaps.pdf` (accessed 16.12.2014)

Novak, J. D. and Gowin, D. B. (1989), *Learning how to learn*, University Press, Cambridge.

Novick, M. R. (1966), The axioms and principal results of classical test theory, *Journal of Mathematical Psychology* **3**(1), 1–18.

Nygaard, K. (1986), Basic concepts in object oriented programming, *SIGPLAN Notices* **21**(10), 128–132.

Ozdemir, A. (2005), Analyzing Concept Maps as an Assessment (Evaluation) Tool in Teaching Mathematics, *Journal of Social Sciences* **1**(3), 141–149.

Özdemir, G. and Clark, D. B. (2007), An Overview of Conceptual Change Theories, *Eurasia Journal of Mathematics, Science & Technology Education* **3**(4), 351–361.

Park, R. E., Miller, T. R. and Col, L. (1992), Software size measurement: A framework for counting source statements, Technical report, Carnegie Mellon University, Pittsburgh.

Pea, R. D. (1986), Language-independent conceptual "bugs" in novice programming, *Journal of Educational Computing Research* **2**, 25–36.

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M. and Paterson, J. (2007), A survey of literature on the teaching of introductory programming, *in* Working group reports on ITiCSE on Innovation and technology in computer science education, Dundee, Dec 2007, ACM Press, New York, pp. 204–223.

Pedroni, M. (2009), Concepts and Tools for Teaching Programming, Dissertationsschrift, ETH Zurich, Zurich.

Pedroni, M. and Meyer, B. (2010), Object-Oriented Modeling of Object-Oriented Concepts, *in* J. Hromkovič, R. Královič and J. Vahrenhold, eds, Teaching Fundamentals Concepts of Informatics, Vol. 5941 of *Zürich*, Jan 13-16 2010, Springer, Berlin, pp. 155–169.

Pedroni, M., Oriol, M., Meyer, B. and Angerer, L. (2008), Automatic extraction of notions from course material, *in* Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, Mar 12-15 2008, ACM Press, New York, pp. 251–255.

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F. and Simmons, R. (1988), Conditions of Learning in Novices Programmers, *in* E. Soloway and J. Spohrer, eds, Studying the novice programmer, Interacting With Computers : Iwc, L. Erlbaum Associates, Mahwah, pp. 261–279.

Piaget, J. (1929), *The Child's Conception of the World*, Littlefield Adams Quality Paperbacks, Lanham, Maryland.

Piaget, J. (1952), *The origins of intelligence in children;*, International Universities Press, New York.

Pinto, Y. (2012), The efficacy of homogeneous groups in enhancing individual learning, *Journal of Education and Practice* **3**(3), 25–38.

Ponocny, I. (2001), Nonparametric goodness-of-fit tests for the rasch model, *Psychometrika* **66**(3), 437–460.

Posner, G. J., Strike, K. A., Hewson, P. W. and Gertzog, W. A. (1982), Accommodation of a scientific conception: Toward a theory of conceptual change, *Science Education* **66**(2), 211–227.

Pugh, J. R., LaLonde, W. R. and Thomas, D. A. (1987), Introducing object-oriented programming into the computer science curriculum, *in* Proceedings of the eighteenth SIGCSE technical symposium on Computer science education, St. Louis, Feb 19-20 1987, ACM Press, New York, pp. 98–102.

Putnam, R. T., Sleeman, D., Baxter, J. A. and Kuspa, L. K. (1988), A Summary of Misconceptions of High School Basic Programmers, *in* E. Soloway and J. Spohrer, eds, Studying the novice programmer, Interacting With Computers : Iwc, L. Erlbaum Associates, Mahwah, pp. 301–314.

Quibeldey-Cirkel, K. (1994), *Das Objekt, Paradigma in der Informatik*, Teubner, Stuttgart.

Raadt, M. d., Watson, R. and Toleman, M. (2005), Textbooks: under inspection, Technical report, University of Southern Queensland, Toowoomba.

Ramalingam, V., LaBelle, D. and Wiedenbeck, S. (2004), Self-efficacy and Mental Models in Learning to Program, *in* Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education, Leeds, June 28-30 2004, ACM Press, New York, pp. 171–175.

Rasch, G. (1980), *Probabilistic models for some intelligence and attainment tests*, University of Chicago Press, Chicago.

Rentsch, T. (1982), Object Oriented Programming, *SIGPLAN Notices* **17**(9), 51–57.

Renumol, V., Janakiram, D. and Jayaprakash, S. (2010), Identification of Cognitive Processes of Effective and Ineffective Students During Computer Programming, *ACM Transactions on Computing Education* **10**(3), 1–21.

Rieman, J. (1996), A field study of exploratory learning strategies, *ACM Transactions on Computing Education* **3**(3), 189–218.

Robins, A., Haden, P. and Garner, S. (2006), Problem distributions in a CS1 course, *in* D. Tolhurst and S. Mann, eds, Proceedings of the 8th Austalian conference on Computing education, Hobart, Jan 16-19 2006, Australian Computer Society, Inc, Darlinghurst, pp. 165–173.

Robins, A., Rountree, J. and Rountree, N. (2001), My program is correct but it doesnt run: A review of novice programming and a study of an introductory programming paper, Technical Report OUCS-2001-06, University of Otago.

Robins, A., Rountree, J. and Rountree, N. (2003), Learning and teaching programming: A review and discussion, *Computer science education* **13**(2), 137–172.

Rosson, M. B. and Alpert, S. R. (1990), The cognitive consequences of object-oriented design, *Human-Computer Interaction* **5**(4), 345–379.

Rost, J. (2004), *Lehrbuch Testtheorie - Testkonstruktion*, Psychologie Lehrbuch, 2nd edition, Huber, Bern.

Rubin, J. (1967), Optimal classification into groups: An approach for solving the taxonomy problem, *Journal of Theoretical Biology* **15**(1), 103–144.

Ruiz-Primo, M. A. (2000), On the use of concept maps as an assessment tool in science: What we have learned so far, **2**(1), 29–52.

Ruiz-Primo, M. A., Schultz, S. E., Li, M. and Shavelson, R. J. (2001), Comparison of the reliability and validity of scores from two concept-mapping techniques, *Journal of Research in Science Teaching* **38**(2), 260–278.

Ruiz-Primo, M. A. and Shavelson, R. J. (1996), Problems and issues in the use of concept maps in science assessment, *Journal of Research in Science Teaching* **33**(6), 569–600.

Ruiz-Primo, M. A., Shavelson, R. J. and Schultz, S. E. (1998), On The Validity Of Concept Map-Base Assessment Interpretations: An Experiment Testing The Assumption Of Hierarchical Concept Maps In Science, Technical Report 455, University of California, Los Angeles.

Rumbaugh, J. (1991), *Object-oriented modeling and design*, Prentice Hall, Englewood Cliffs, N.J.

Saeli, M., Perrenet, J., Wim M. G. Jochems and Zwaneveld, B. (2011), Teaching programming in secondary school: A pedagogical content knowledge perspective, *Informatics in Education* **10**(1), 73–88.

Sajaniemi, J., Kuittinen, M. and Tikansalo, T. (2008), A study of the development of students' visualizations of program state during an elementary object-oriented programming course, *Journal on Educational Resources in Computing* **7**(4), 1–31.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L. and Zander, C. (2008), Student understanding of object-oriented programming as expressed in concept maps, *in* Proceedings of the 39th SIGCSE technical symposium on Computer science education, Portland, Mar 12-15 2008, ACM Press, New York, pp. 332–336.

Sanders, K. and Thomas, L. (2007), Checklists for grading object-oriented CS1 programs: concepts and misconceptions, *in* Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, Dundee, Dec 2007, ACM Press, New York, pp. 166–170.

Sattar, A. and Lorenzen, T. (2009), Teach Alice programming to non-majors, *SIGCSE Bulletin* **41**(2), 118–121.

Schubert, S. and Stechert, P. (2010), Competence Model Research on Informatics System Application, *in* Proceedings of the IFIP Conference New developments in ICT and Education, Amiens, June 2010, pp. 28–30.

Schulte, C. (2013), Reflections on the Role of Programming in Primary and Secondary Computing Education, *in* Proceedings of the 8th Workshop in Primary and Secondary Computing Education, Aarhus, Nov 11-13 2013, ACM, New York, pp. 17–24.

Schulte, C. and Magenheim, J. (2005), Novices' expectations and prior knowledge of software development: results of a study with high school students, *in* Proceedings of the first international workshop on Computing education research, Seattle, Oct 1-2 2005, ACM Press, New York, pp. 143–153.

Schunk, D. H. (2011), *Learning theories: An educational perspective*, 6th edition, Pearson, Boston.

Schwill, A. (1994), Fundamental ideas of computer science, *Bulletin-European Association for Theoretical Computer Science* **53**, 274–297.

Sebesta, R. W., Mukherjee, S. and Bhattacharjee, A. K. (2013), *Concepts of programming languages*, 10th edition, Pearson, Boston.

Sedgewick, R. and Wayne, K. D. (2008), *Introduction to programming in Java: An interdisciplinary approach*, Pearson Addison-Wesley, Boston.

Sethi, R. (2003), *Programming Languages: Concepts and Constructs*, 2nd edition, Addison-Wesley, Boston.

Shavelson, R. and Ruiz-Primo, M. (2005), On the psychomentrics of assessing science understanding, *in* J. J. Mintzes, ed., Assessing science understanding, Educational psychology, Elsevier, Amsterdam, pp. 304–341.

Shrager, J. and Klahr, D. (1986), Instructionless learning about a complex device: the paradigm and observations, *International Journal of Man-Machine Studies* **25**(2), 153–189.

Sousa, D. A. (2006), *How the brain learns*, 3rd edition, Corwin Press, Thousand Oaks.

Spohrer, J. C. and Soloway, E. (1986), Novice mistakes: are the folk wisdoms correct?, *COMMUNICATIONS OF THE ACM* **29**(7), 624–632.

Ständige Kultuministerkonferenz (2004), Einheitliche Prüfungsanforderungen Informatik.
URL: `http://www.kmk.org/fileadmin/veroeffentlichungen_beschluesse/1989/1989_12_01-EPA-Informatik.pdf` (accessed 14.11.2014)

Steinert, M. (2010), Lernzielstrukturen im Informatikunterricht, Habilitationsschrift, TU München, München.

Strobl, C. (2010), *Das Rasch-Modell: Eine verständliche Einführung für Studium und Praxis*, Vol. 2 of *Sozialwissenschaftliche Forschungsmethoden*, Hampp, Mering.

Stroustrup, B. (1988), What is object-oriented programming, *IEEE SOFTWARE* **5**(3), 10–20.

Stroustrup, B. (1994), *The design and evolution of C++*, Addison-Wesley, Reading.

Stroustrup, B. (2003), *The C plus plus programming language*, 3rd edition, Addison-Wesley, Boston.

Sweller, J. (1988), Cognitive Load During Problem Solving: Effects on Learning., *Cognitive Science* **12**(2), 257–285.

Sweller, J. (1989), Cognitive technology: Some procedures for facilitating learning and problem solving in mathematics and science, *Journal of Educational Psychology* **81**(4), 457–466.

Sweller, J., van Merrienboer, J. and Paas, F. (1998), Cognitive Architecture and Instructional Design, *Educational Psychology Review* **10**(3), 251–296.

Technische Univerisät München (2014), Curriculum for the Bachelor's degree.
URL: `http://www.in.tum.de/fuer-studierende/bachelor-studiengaenge/informatik/studienplan/studienbeginn-ab-ws-20122013.html` (accessed 14.11.2014)

Temte, M. C. (1991), Let's Begin Introducing the Object-oriented Paradigm, *in* Proceedings of the Twenty-second SIGCSE Technical Symposium on Computer Science Education, San Antonio, Mar 7-8 1991, ACM Press, New York, pp. 73–77.

Tew, A. E. and Guzdial, M. (2010), Developing a validated assessment of fundamental CS1 concepts, *in* Proceedings of the 41st ACM technical symposium on Computer science education, Milwaukee, Mar 10-13 2010, ACM Press, New York, pp. 97–101.

The CSTA Standards Task Force (2011), CSTA K-12 Computer Science Standards, Technical report, Computer Science Teacher Association.
URL: `http://csta.acm.org/Curriculum/sub/K12Standards.html` (accessed 15.12.2014)

The Royal Society (2012), Shut down or restart? The way forward for computing in UK schools.
URL: `https://royalsociety.org/~/media/education/computing-in-schools/2012-01-12-computing-in-schools.pdf` (accessed 16.12.2014)

Truong, N., Roe, P. and Bancroft, P. (2004), Static analysis of students' Java programs, *in* Proceedings of the 6th conference on Australasian computing education, Dunedin, Jan 2004, Australian Computer Society, Inc, Darlinghurst, pp. 317–325.

Turkle, S. and Papert, S. (1990), Epistemological Pluralism: Styles and Voices within the Computer Culture, *Signs: Journal of Women in Culture and Society* **16**(1), 128–157.

van Roy, P., Armstrong, J., Flatt, M. and Magnusson, B. (2003), The role of language paradigms in teaching programming, *in* Proceedings of the 34th SIGCSE technical symposium on Computer science education, Reno, February 19-23 2003, ACM Press, New York, pp. 269–270.

Ventura, P. R. (2005), Identifying predictors of success for an objects-first CS1, *Computer science education* **15**(3), 223–243.

Verguts, T. and De Boeck, P. (2000), A note on the Martin-Löf test for unidimensionality, *Methods of Psychological Research Online* **5**(1), 77–82.
URL: `http://www.dgps.de/fachgruppen/methoden/mpr-online/issue9/` (accessed 16.12.2014)

Verhelst, N. D. (2008), An Efficient MCMC Algorithm to Sample Binary Matrices with Fixed Marginals, *Psychometrika* **73**(4), 705–728.

Vihavainen, A., Paksula, M. and Luukkainen, M. (2011), Extreme apprenticeship method in teaching programming for beginners, *in* Proceedings of the 42nd ACM technical symposium on Computer science education, Dallas, March 9-12 2011, ACM Press, New York, pp. 93–98.

Vihavainen, A., Vikberg, T., Luukkainen, M. and Kurhila, J. (2013), Massive increase in eager TAs: experiences from extreme apprenticeship-based CS1, *in* Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, July 1-3 2013, ACM Press, New York, USA, pp. 123–128.

Vihavainen, A., Vikberg, T., Luukkainen, M. and Pärtel, M. (2013), Scaffolding students' learning using test my code, *in* Proceedings of the 18th ACM conference on Innovation and technology in computer science education, Canterbury, England, July 1-3 2013, ACM Press, New York, USA, pp. 117–122.

Vujošević-Janičić, M. and Tošić, D. (2008), The Role of Programming Paradigms in the First Programming Courses, *The Teaching of Mathematics* **XI**(2), 63–83.

Vygotsky, L. (1962), *Thought and language.*, M.I.T. Press Massachusetts Institute of Technology, Cambridge.

Wegner, P. (1989), Conceptual evolution of object-oriented programming, Technical Report CS-89-48, Brown University Department of Computer Science, Providence.

Wegner, P. (1990), Concepts and paradigms of object-oriented programming, *SIGPLAN OOPS Messenger* **1**(1), 7–87.

Weinert, F. E. (2001), Concept of competence: A conceptual clarification, *in* D. S. Rychen and L. H. Salganik, eds, Defining and selecting key competencies, Hogrefe & Huber, Seattle, pp. 45–65.

Wigfield, A., Cambria, J. and Eccles, J. S. (2012), Motivation in Education, *in* R. M. Ryan, ed., The Oxford handbook of human motivation, Oxford University Press, New York, pp. 463–478.

Wilson, B. C. and Shrock, S. (2001), Contributing to success in an introductory computer science course: a study of twelve factors, *in* Proceedings of the 32nd SIGCSE technical symposium on Computer science education, Charlotte, Feb 21-25 2001, ACM Press, New York, pp. 184–188.

Winslow, L. E. (1996), Programming pedagogy—a psychological overview, *SIGCSE Bulletin* **28**(3), 17–22.

Wirth, N. (2002*a*), Pascal and Its Successors, *in* M. Broy and E. Denert, eds, Software Pioneers, Springer-Verlag, New York, pp. 108–119.

Wirth, N. (2002*b*), The Programming Language Pascal, *in* M. Broy and E. Denert, eds, Software Pioneers, Springer-Verlag, New York, pp. 121–148.

Wren, A. (2007), Relationships for object-oriented programming languages, Technical Report UCAM-CL-TR-702, University of Cambridge, Computer Laboratory.

Yin, Y., Vanides, J., Ruiz-Primo, M. A., Ayala, C. C. and Shavelson, R. J. (2005), Comparison of two concept-mapping techniques: Implications for scoring, interpretation, and use, *Journal of Research in Science Teaching* **42**(2), 166–184.

Zimmerman, B. J. (2000*a*), Attaining Self-Regulation: A Social Cognitive Perspective, *in* M. Boekaerts, P. R. Pintrich and M. Zeidner, eds, Handbook of Self-Regulation, Academic Press, San Diego, pp. 13–39.

Zimmerman, B. J. (2000*b*), Self-Efficacy: An Essential Motive to Learn, *Contemporary Educational Psychology* **25**, 82–91.

Zuse, H. (1999), Geschichte der Programmiersprachen, Technical Report 1999-1, Technische Universität Berlin, Berlin.

List of Tables

List of Figures