IT Licentiate theses
2007-002

Students working with a Large Software System: Experiences and Understandings

JONAS BOUSTEDT

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

Students working with a Large Software System: Experiences and Understandings

BY
JONAS BOUSTEDT

May 2007

DIVISION OF SCIENTIFIC COMPUTING
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science with specialization in Computer Science Education Research at Uppsala University 2007

Students working with a Large Software System: Experiences and Understandings

Jonas Boustedt

jonas.boustedt@it.uu.se

Division of Scientific Computing
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se/

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

Abstract

This monograph describes an empirical study with the overall aim of producing insights about how students experience the subject Computer Science and its learning environments[1], particularly programming and software engineering.

The research takes a start in the students' world, from their perspective, using their stories, and hence, we have chosen a phenomenographic approach for our research. By interpreting the students' descriptions and experiences of various phenomena and situations, it is possible to gain knowledge about which different conceptions students can have and how teaching and the learning environment affect their understanding. In this study, we focus specifically on students' conceptions of aspects of object-oriented programming and their experiences of problem solving situations in connection with object-oriented system development.

The questions posed enlighten and focus on the students' conceptions of both tangible and abstract concepts; the study investigates how students experienced a task concerning development in a specific software system, how they conceived the system itself, and how the students describe the system's plugin modules. Academic education in programming deals with abstract concepts, such as interfaces in the programming language Java. Hence, one of the questions in this study is how students describe that specific abstract concept, in a context where they are conducting a realistic software engineering task.

The results show that there is a distinct variation of descriptions spanning from a concrete to-do list to a more advanced description where the interface plays a crucial role in order to produce dynamic and adaptive systems. The discussion interprets the results and suggests how we can use them in teaching to provide an extended and varied understanding, where the educational goal is to provide for and strengthen the conditions for students to be able to learn how to develop and understand advanced software.

[1] The study started as part of the research project *Learning, Learning Resources and Learning Environments in Computer Science* – a collaborative project between Uppsala University and University of Gävle (Högskolan i Gävle), funded by the Swedish Research Council 2002-2004.

Acknowledgements

I wish to thank all of my advisors, above all Michael Thuné, my inspiring main advisor who is always very interested, patient and supportive; Shirley Booth who guided me during my first two years, sharing her valuable expertise in phenomenography; and Roy Nilsson at University of Gävle, who gives me good advice using his deep commitment and knowledge in didactics.

Some persons at Uppsala University are special to me because of their devotion to educational research, and their willingness to discuss these matters with me. I wish to thank Anna Eckerdal, Anders Berglund, Liselott Dominicus van den Bussche, Mats Daniels, Arnold Pears and Mattias Wiggberg.

At my place of work, the Computer Science Department in University of Gävle, where I am teaching, only a few persons meet me in my role as a PhD student and researcher. I want to thank Eva Carling for persuading me into graduate education; Magnus Blom and Carina Pettersson for their interest; Anders Jackson for standing his roommate; Peter Walander and Tony Malmqvist for their warm personal concern; Anita Hussénius for encouraging me to make my PhD come first; Douglas Howie for his linguistic support, and Goran Milutinovic for just being Goran. Moreover, I wish to thank Thorleif Cederqvist and Lars-Göran Eriksson for guiding me through bureaucracy. I am also very grateful to the students who participated in this study.

I met Josh Tenenberg at the SIGCSE conferences in 2006 and 2007. He took the time to read a draft version of this text and we discussed it on telephone for over one hour. He was the first person outside our research group that read my work, and when he told me he liked it, he really made my day, week, and month! Thanks Josh!

The funding for my research comes from the Department of Information Technology at Uppsala University, the Swedish Research Council, and the Committee of Teacher Training at University of Gävle.

My family is the most important thing in my life and I want to thank my dear and beloved wife Ann-Louise, our wonderful children Tova, Simon and Liselotte, for all their sacrifices and support. I hope I can pay back, some way, and lay back, some day.

Contents

1 Introduction

This monograph presents a study aimed at Computer Science students at university level, who are somewhere in the middle of their studies. The purpose of the study is to gain insights into how students experience and understand object-oriented thinking and programming and some related concepts of importance.

The overall concern has to do with how we can improve teaching and learning, and how well we prepare the students for a *professional* life. Hence, one of the driving questions is what happens when students have to deal with programming in larger software systems. Our goal is that this work will help us to answer some of our questions, and that it will contribute to the didactics of Computer Science.

1.1 The research interest

I have taught object-oriented programming for years, and still I often think it is hard to explain and motivate some of the advanced concepts that are typical for the object-oriented way of thinking.

As in all learning there is some fundamental knowledge that must be gained, which involve things such as syntax, program flow, classes, objects, references, procedure calls, et cetera. It takes time and effort to learn these things and in the introductory programming course, the students are very absorbed into mastering these fundamentals. I use the term *programming in the small* (Dalbey, 1998; DeRemer, 1975) to address what beginners in programming are doing, regardless of the “objects first or procedures first” debate.

However, nowadays, in professional contexts and programming communities, it is common that software developers use integrated sub-systems to build large and complex systems, and programmers more rarely develop small and self-contained programs.

In order to design such systems or develop new applications, a comprehensive view is required. This includes an understanding of the interaction between different parts of the system, an ability to see consequences of different design decisions, a comprehension of the need of documentation for future use and an ability to interpret documentations. Besides, it is usual that companies assign new employees to work with system maintenance or development of smaller parts in existing systems.

Even if the work with maintenance and smaller parts to some extent is a limited activity, it claims high standards of the understanding of the large system's structure and mechanisms. We can designate these activities *programming in the large* (Dalbey 1998; DeRemer, 1975).

The conceptual differences between programming in the small and in the large may constitute a potential problem in the education, namely that the students, who strive for a professional career within the area of system development and programming, might not get enough opportunities to work with programming in software systems in the wider sense. This is why I want to study how the students handle programming in the large.

1.2 The research questions

How can we improve teaching in object-oriented programming with a special aim at programming in the large? Although, it is a justified and interesting question, it is too broad and unspecific to be able to answer. We can make the question operational by asking the following questions:

1. How do students experience and describe concepts that relate to programming in the large?
2. Are there typical behaviours when students face problems of this type?
3. Are there connections between conceptual understanding and the practical abilities to program in the large?
4. Are the students well prepared for working with extensive software, in other words, is the education relevant for the profession?
5. If we can find any answers to the questions above, how can we use them in our teaching?

1.3 Research approach and methods

Phenomenography (see Chapter 3) is a qualitative research approach that is well suited for this kind of empirical investigations, as it especially focuses on learning and education. A researcher who takes this approach wishes to get deeply into how people view things, the underlying causes, the nuances and the details. The ambition is that a reader who takes part of the results of a qualitative inquiry will understand the world as the participants see it, as interpreted by the researcher, and it is their view of the reality that is the research subject.

In order to provide feasible conditions for the collection of information on how students experience concepts and how they work with large object-oriented software, an experiment was prepared (see Chapter 4): a realistic development task in a realistic software system, more extensive than the programs the students have seen in their previous studies. The purpose with this experiment was to put the students in a realistic *programming in the large* problem-solving situation, where the concepts studied in a previous course would appear in a natural context. By these means, we could gather a rich amount of data by interviewing the students, and by recording their actions, both as manifested on the computer screen and as in the files stored on the computer.

- The first question about conceptions can be investigated by phenomenographic analysis of the students' descriptions of the concepts in the transcriptions of the interviews. We want to shed light upon in which qualitatively different ways the students experience the concepts *interface*[2] and *plugin*[3] in context of a *system*[4] where the first two concepts stand out as relevant. The concept "interface" is especially interesting as it is a part of the instruction of programming with Java.
- We search for evidence of the students' view of the task and their approach for solving it in the interviews and in the practical results of their work.
- It should be possible to address the connection between *understanding concepts* and *practical abilities* through a discussion about the results of the first two questions.
- The character of question 4 and 5 has a different nature compared to the first three and we will try to address them in the discussion.

1.4 Object-orientation and the Java Interface

This section provides a short introduction to object-orientation and the concepts object, class, reference and interface, and it is intended for the readers who are not already well acquainted with the subject. The purpose is to create opportunities for the reader to benefit more from the rest of this work.

Object-orientation and object-oriented programming represent a special way of thinking. In this paradigm, various systems are described in terms of interacting objects, where every object is regarded as a unit having a limited area of responsibility. Each object can offer certain services and it has a memory of its own. The information (or data) in such a system is essentially constituted and controlled by the objects themselves. This way of thinking

[2] Interface refers to a specific language construct in the programming language Java. It is further described in section 1.4.
[3] A adaptive code module that can extend an existing software
[4] The students worked with an administrative software system. See Appendix D.

goes both for systems in the real world and their reflections in a model implemented in software. The object-oriented paradigm differs considerably from, as for example, the procedural paradigm, where the systems are rather described in terms of data flows, where information is passed to and treated by various procedures.

The program code for object-oriented software (the text that the programmer writes) consists of a number of so-called *classes* and each class describes a unique type of object. A class often describes a model of some concept taken from the real world, such as a person. One of the fundamental properties of a class is the possibility to create instances (objects) of its own type. If there is a class that defines a model of a person, it is possible to create an unlimited amount of person objects, each representing an individual person of its own. Each person object holds and manages its own name, address, et cetera. When the software is executed on the computer, a number of objects are created, and taken all together; the objects and their interactions constitute the program's behaviour.

Interaction between an object (A) and another object (B) takes place by means of object A passing a *message* to object B. This message contains a request for B to execute a service (operation). In order to pass the message (call the operation) it is required that A is in contact with B through a named reference variable[5]. We can imagine a scenario where A is an object of the type *person* which has a reference variable of the type *CD_Player*, named *player*, referring to the object B, which is an object of the type *CD_Player*. In this case A can pass a message to B: *player.play()*. Using this message, object A requests object B to execute the operation *play()*, that is to say, "play a CD, please." The reference variable's type always determines the messages that are possible to pass.

The programming language Java is strongly typed, which (amongst other things) implies that a reference variable must have the same basic type as the object that it wants to refer. Consequently, a reference variable of the type *CD_Player* can only refer to objects descending from the class (type) *CD_Player*. If we assume that a new type of object is introduced, for instance an *MP3_Player*, then our *person*, from the example above, cannot use the MP3-player since the reference variable *player* only can refer to objects of the type *CD_Player*. This is true even if the desired operation, *play()*, exists also in the class *MP3_Player*.

In the programming language Java, we can still handle the need for dynamic behaviour in its strongly typed environment, for instance by using a so-called *interface*[6]. In a Java interface, a limited set of operations are specified. However, the interface omits the operations' implementations, that is to

[5] In the programming language Java, a reference variable is an *object handle* which can refer to objects.
[6] Using interfaces is one way. Another way is by using inheritance. See Appendix B.

say, their behaviour is not defined – only the operations’ specifications are included. The interface actually defines a type that can be used to declare reference variables, and such a variable can be used as a handle towards all objects which classes implement the actual interface respectively. The implementing classes are actually forced to define (implement) behaviour for all of the specified operations in the interface. We assume that we have created the interface *playable* that defines the operation *play()*, and furthermore that both classes *CD_Player* and *MP3_Player* implement the interface *playable*. Suppose that the reference variable *player* now has the interface type *playable*. Then the *person* can play both CD records and MP3 files. Moreover, in the future the same person can play all kinds of media without having to adapt to their specific technical implementations. The only requirement for this to work out well is that they are all *playable*, that is to say, they all implement the interface *playable.* The following quote expresses this line of argument as a general advice to programmers: “*Program to an interface, not an implementation*” (Gamma et al., 1995, p.18).

1.5 Outline

We have given the background of the study, and a definition of the research questions, followed by some methodological considerations, and finally a brief introduction to object-orientation. Chapter 2 gives an overview of related work and Chapter 3 describes the phenomenographic research approach. Chapter 4 describes the empirical study and its conduct. Chapters 5 to 9 present the results and supporting evidence. Chapter 10 and 11 discuss the results, their interpretations and implications for teaching, and Chapter 12 outlines the conclusions. In the appendices you can find a word list and a more elaborate and personal presentation of object-orientation. In addition, you can find some of the materials referred to in the text there.

2 Related work

This chapter describes other studies that relate to this work in various ways, how they are related and why they have relevance to this study. We have selected four themes: (1) Computer Science Education Research (CSER), (2) learning to program, (3) educating for a profession in industry, and (4) the interface concept in Java.

2.1 Computer Science Education Research

This is a study within the wide area of CSER. There are some examples of literature providing general overviews of the field of Computer Science Education Research. A number of researchers has put an effort into it and have managed to contribute with descriptions of what is going on and what has been done in this young discipline. Clancy et al. (2001), describes models and areas for such research, and Holmboe (2001), presents a research agenda. Pears and Daniels (2003) suggest a model for how such research can be achieved. Berglund, Daniels & Pears (2006), describe examples of qualitative research projects in the area.

2.1.1 Phenomenographic studies in CS

There are many possible approaches to study aspects of learning in computer science and it is important to formulate an outlook on the research and get familiar with that research tradition. This study takes mainly a phenomenographic perspective (see Chapter 3) and it is inspired by other studies that use the same research approach. These studies are good examples of how to conduct such investigations and how to interpret and discuss their results.

Shirley Booth is one of the pioneers who took a phenomenographic approach in order to tackle pedagogical issues within computer science education. In her classic work in the area (Booth, 1992), she investigated how students learn and approach programming through a study where the students' descriptions of their conception of several related phenomena and situations were analysed. The students learned a functional programming language, and it is interesting to reflect on similarities and differences compared to object-oriented programming.

Anna Eckerdal (2006) has studied how a group of students, involved in a Java programming course, in different ways experience the object-oriented concepts class and object. Her work is close to my research interest, and it has been very valuable to take part of how the students experienced this type of programming.

Anders Berglund (2005) studied how students describe concepts related to computer communication protocols. He also studied the students' activities in groups. The context was an internationally distributed project course where students from USA and Uppsala worked together in an advanced software engineering (SE) project. This setting has connections to my study as it also has complex software as a theme for the study of students' experiences of concepts.

Chris Cope (2006) investigated how students learn and experience the concept of information systems. In my study, the students worked with programming in a database system, and therefore it was interesting to see how the students in his study viewed these types of systems.

2.2 Learning to program

Even though we have chosen a phenomenographic approach on "learning to program in the large," we are interested in other perspectives to get a broader view of the area of programming education and learning. Naturally, all perspectives contribute to the picture of how learning comes about and it is important to understand how other researchers are reasoning. Moreover, how researchers try to widen their perspectives and combine their research approaches with other traditions.

2.2.1 The awareness of cultures and communities

One of the conclusions in Shirley Booth (2001a) is that there is a tendency within the universities to move away from the traditional pedagogy where knowledge is transferred from the teachers to the students. Now they approach a way of teaching where the students take an active part of their learning by working in groups and projects.

However, she notices, there is a lack of a theoretical foundation for this way of approaching learning and therefore she suggests a phenomenographic perspective, where teachers can establish their view of learning and teaching and then learn from how their students learn.

Booth also investigates various approaches to learning and learning studies. She starts from two different research approaches. Firmly rooted in the phenomenographic tradition and perspective, she discusses the possibility to combine it with a socio-cultural perspective in such a way that both traditions could benefit and learn from each other (Booth, 2001b). In this paper, she discusses how she would like to take on this new perspective and re-examine a previous study by adding new questions to it. She is interested in what it takes to enter a “datalogical” culture, and if the answers have connections to the previous results. She identifies three essential cultures: the informal (amateur) culture, the academic culture, and finally the professional culture.

An interesting observation is that advocates for the academic and professional cultures are debating the purpose, the practice and the contents of the “datalogical” education. She concludes that it seems both possible and enriching to extend the phenomenographic terminology and theoretical ground to include cultural aspects of learning. The text thoroughly covers the phenomenographic research approach. When it comes to learning in cultures, she uses the concept *Community of Practice* (CoP), coming from the socio-culturist Etienne Wenger. Then she goes about and argues for a connection between cultures and her previous results, namely the categories of description of programming and how learning how to program takes place.

The preliminary result shows that there are such connections and she tries to make a mapping between cultures (CoPs) and her categories. The students’ datalogical (cultural) identity before the studies affects their learning. At the same time, their cultural identity is influenced by the studies and of course, they are gradually incorporated in the academic culture (for better or worse).

In summary, a tenable vocabulary of concepts must be constructed and a new theory must be developed before valid conclusions can be drawn from this combined research approach. In any case, both phenomenography and socio-culturism should benefit from an exchange of ideas. A preliminary conclusion from this theoretical research attempt is that the educational institutions should consider and use the datalogical cultures that exist *outside the academia*.

Yiffat Ben-David Kolikant (2004) describes the clash between the informal culture of technology users and the academic culture and CoP. Under the assumption that the educational goal is to help the students to enter the academic CoP, she noticed that it is possible to regard a specific course as an entry point to the community of computer science practitioners. She claims that students will experience the practice of CS if they work with a certain type of assignments and when they do, they can “cross the boundaries from the User CoP”. Within concurrent programming, there is a rich set of problems to deal with and one of them is the synchronization problems. She suggests how to design an assignment that motivates the students to enter the academic culture. In the study, she follows the students and investigates how and when they cross borders.

2.2.2 A constructivist approach on learning to program

Said Hadjerrouit (1999) suggests a constructivist approach to learning object-oriented design and programming. He claims that passive learning, where knowledge is transmitted from the teacher to the learner, is not adequate for most students when it comes to the object-oriented paradigm. The concepts are far more abstract than in procedural programming.

Instead, the learners' minds must be deeply involved, and the learners themselves must construct their knowledge about object-oriented concepts, which implies that they must play an active role in the learning process. The object-oriented concepts, the programming language and the problem specific knowledge must be strongly linked together in order to create good grounds for being able to construct knowledge.

To get students involved, we must provide for *realistic and motivating problems* for them to work with. Teachers should transform passive lectures into student activities that aim towards construction of knowledge that agrees more with the expert's view on programming, such as the importance of skills in analysis, design, analogical thinking, and reflexive and critical thinking.

Hadjerrouit suggests guidelines and appropriate activities. We must know and adapt to the students prior knowledge. The concepts should be listed explicitly. Moments of reflection should be part of the activities. Examples of activities are, design by adapting existing solutions (patterns), study the Java API and find code to reuse, *study experts' solutions*, organize knowledge by similarities and differences, develop alternate solutions, and finally, reflect on the solution process.

In a more recent publication (Hadjerrouit, 2005), he brings this approach further and describes a general model for how constructivism can be applied to teaching within software engineering. He refers to some of the central figures of constructivism (Bruner, Piaget, Vygotsky and Van Glaserfeld) who claim that knowledge cannot be directly transferred to learners. Instead, learning is an active process of construction.

He suggests a set of pedagogical guidelines that should be followed: construction, cognitive skills, *authentic tasks*, related cases, cooperation, and information technology. By authentic tasks, he means assignments that build on experiences and contain relevant concepts and principles of software engineering. The students will understand and appreciate the *connection with reality* if the tasks are connected to external organisations or private companies. The tasks should contain all the relevant information that is needed to be solved, and they should have the intrinsic property of being interesting and motivating. The students should be the initiators of the assignments, and the tasks should be designed in consultation with the teachers.

2.2.3 A cognitive perspective on learning to program

Anthony Robins, Janet Rountree and Nathan Rountree (2003) have compiled a literature overview on studies of learning programming. They are mainly interested in how beginners learn to program, and they take a cognitive perspective. They notice that the research has focused on understandings and development of programs, mental models, and knowledge and skills that are required to be able to program.

They claim it takes 10 years for a novice to become an expert, and they classify five developmental levels: novice, advanced beginner, competent, skilled, and expert. Five overlapping domains are involved in learning, namely: orientation, the concept of machine, notation, structures and practical skills.

As examples of alternate methods of instruction, they mention that the students should learn to use a new vocabulary and that more weight should be put on understanding and using named patterns. A different point they have is that abstract representations not can be learned in a direct way, they can only be learned by using and working with the practical operations from which the abstractions are derived.

This way of reasoning have a direct connection to Anna Sfard's (1991) way of explaining learning within mathematics as a duality between conceptual and operational understanding, and to Orit Hazzan (2003) who uses Sfard's model applied to learning in Computer Science.

Another way to approach learning in programming is through problem solving, where the details of a programming language are introduced successively through the needs by a given problem. However, there is a complication; the students have difficulties to formulate the solutions to the problems as programs.

In the end of the literature review, the authors point out four trends within the research. The first is the tendency to divide novices from experts, focussing on the novices' shortcomings. The second trend is to *separate knowledge from strategies*, and the third trend is to distinguish between *understanding programs* and the ability to generate programs. The fourth is comparisons between object-oriented languages and procedural languages. Even though object oriented languages give a more distinct way to structure and plan programs, it is required to put a big effort on procedural aspects, especially for the weaker students.

The authors comment their review by claiming that it is more *important to study differences between efficient and inefficient novices*, rather than studying the differences between experts and novices. The focus should be set on what can promote students to perform as efficient novices. Some potential factors could be motivation, self-reliance, how students are treated, aspects of specific and generic knowledge, and finally, strategies and mental models. Strategies for how to get, gain and apply knowledge when it comes to understanding and construction of programs are critical for the learning.

Researchers should look for which strategies the effective students use and teach this to the beginners whenever possible. Teachers should give *many explicit examples of programs* under development and strategies, perhaps by writing programs together with the students during the lessons. An important question is why useful knowledge and strategies are well known, but still not are used.

2.3 Educating for a professional career in industry

One of the main issues in this study concerns "programming in the large". It deals with the question of how well prepared the students get for a profession as developers and programmers, and I have chosen to study this by analysing how students experience related concepts and how they approach a problem-solving situation.

The research interest driving our study takes its starting point in the presumption that a considerable portion of the students that follow programmes in computer science or computer engineering strives for profession in the IT industry. The education should prepare the student for a wide spectrum of professional roles. To be a good system developer or a software engineer, a comprehensive view is required, that amongst other things includes knowledge about computer systems, programming, databases, project methodology, test methods, and good treatment of customers. The object-oriented paradigm and its related system development methods deals with all of the topics mentioned above.

Knowledge in academic educational systems is traditionally specialized and divided into pure subjects, and what typically characterizes experts and researchers is the tendency to know very much about very little (deep but narrow). Perhaps this is the only way to obtain new knowledge and manage the heritage of from the past. The thought behind study programmes that prepare for a profession and therefore include several subjects is that the knowledge should be integrated within the students' minds and that is will result in competence and professionalism. However, there is a potential conflict of interest between, on one hand, the study programmes that account for the task to produce educated, skilful and professionally trained citizens, and on the other hand, the institutions role to maintain and develop the subjects and their duty as guardians of the free and independent academia.

One way out of this dilemma is to organize the educational institutions as professional schools, such as institutes of technology that specialize towards certain professions. The content of the subjects is considered from a professional perspective and is adapted and integrated to suite specific professional purposes and applications. An unconventional variant is project-based educations, or at least courses that introduce realistic or even authentic projects where the students themselves choose which knowledge are required to fulfil the commissions. For an example, see Jacceri (2001).

I have found support for my thoughts about learning for the professional life. Several studies show that the educational systems have some shortcomings in this area, and there are suggestions about how we can improve the education in this regard.

In their longitudal study, Madeleine Abrant Dahlgren, et al. (2006), investigate the transition from higher education to professional careers within social science, psychology and engineering. Earlier results had shown that not only the content, but also the socio-cultural context contributed to students' learning in various educational programmes.

There were clear differences between teacher types, teaching methods and demands on students. However, not much research had been done on transitions between academia and working life. A starting-point for their research was consequently, how participation in these *communities of practice* (CoP) changes with time. In this context, they also studied reification, that is to say, how abstract concepts are embodied in these CoP.

Their results show that the psychology programme met the demands and needs of professional life in a rational manner. This programme used a thematic structure of the content and integrated academic and professional focus. The social science programme was driven in a traditional academic way and was organized sequentially, giving generic knowledge that must be transformed in order to be used professionally. The engineering programme was also driven with an academic focus using a parallel structure. Much of what was learned within the engineering programme was characterized as knowledge that plays a ritual role for the profession.

Timothy Lethbridge studied 168 professional software developers and tried to find out how relevant their formal education had been for their professional careers. He concluded that there were important subject areas not given enough room in the education, such as project methodology, real-time systems, user interface design, maintenance, re-factoring, leadership, ethics and *professionalism*. However, chemistry and mathematical analysis were given too much emphasis according to their relevance for practising the profession. The shown results led to a revision of the educational programme in order to improve upon the indicated shortcomings.

John Tvedt, Roseanne Tesoriero and Kevin Gary suggest a Computer Science Curriculum that in their view is *better adapted to the needs of the industry* (Tvedt, 2002). They point out that contemporary educational systems produce students with good technical skills, but unfortunately, the students lack the required practical software engineering of the profession. Their solution to this problem is their own proposed educational model, *Software Factory*. The students will learn more and consolidate more of their knowledge by applying their new skills in an authentic development environment.

It would accordingly be of advantage to the students, the teacher staff, the academic institution, and the industry. Their model has been adapted and implemented.

David Parnas (1998) claims that educational programmes within software engineering are not, and should not be, computer science programmes. He reveals an on-going a tug-of-war between different educations in the sense that the computer science educations wants to embrace the concept of software engineering and make it a part of their programmes, whilst there has evolved a new specialization within the technical educations that are entirely focused on software engineering. He points out the necessity of programmes focussing on software engineering that also follow the structure of traditional engineering education. He comes up with the following conclusions:

- Software Engineering and Computer Science are different
- Programmes within Software Engineering must be accredited and ascribed a status in level with civil engineering educations
- There is a need for new courses in SE, not combination of existing ones, such as programming courses with elements of SE
- The teaching style and the organisation of the courses must change
- Staffing of teachers is the most critical problem
- Computer Science has matured, and the numerous results allows for an education devoted to Software Engineering
- It takes a genuine commitment. Both researchers in Computer Science and staff from the traditional engineering educations must acknowledge the eligibility for treating new field seriously.

2.3.1 Apprenticeship

The idea of apprenticeship inspired the way this study was designed, both as a way to put the students in a *state of realism* when they solved the task, and as a *particular way of learning* that might be something to consider in teaching programming in the large. In the following, I will point out some related work in this area.

Lave and Wenger started from the idea to try preserving apprenticeship - the traditional and ancient way of learning (1991). They tried to investigate and explain its relation to the concept *situated learning*. From this perspective, they created a sociologic and cultural epistemological theory based upon the presumption that learning takes place in social forms.

They mean that the modern view of learning totally has left out the social aspect and that it incorrectly focuses on the individuals' learning of facts. On the contrary, learning in their view is all about a process of taking part in new cultures, *communities of practice* (CoP).

In the beginning, the learners are allowed to take limited part of the culture, which is called *legitimate peripheral participation* (LPP). Gradually, the commitment gets deeper and more complex. In their rather radical publication, they give example of five different cases of apprenticeship.

Naturally, one could argue that our systematic way of educating new generations in schools and universities could be regarded as a variant of LPP since the schools are allowed to be peripheral to the modern society's production apparatus and that the students are gradually introduced to the new culture. However, it is more likely that the students approach a more academic culture than the culture of the profession that the study programme aims for, which is certainly not in line with LPP.

Mordechai Ben-Ari (2004) examines situated learning (LPP) in context of Computer Science. A common example of learning, which can be described as LPP in CS, is the concept of Open-Source Software Development (OSSD) and especially the success story of how the operative system Linux was developed. Ben-Ari admits that this really was a case of LPP and that there was a clearly defined Community of Practise (CoP) in the project. At the same time, he argues, there are branches within CS where this form of learning would not be appropriate, especially within pure, theoretical CS (non-applied).

He concludes that LPP in its proper sense is not applicable for the entire chain of learning that must precede the high-technological knowledge that Computer Science Education aims for. Generalization and models must be utilized in order to make the education effective. On the other hand, it is possible to make use of and be inspired of LPP when the content of the education is designed and the literature is chosen. The teachers should be well aware of the different CoP that the education aims for and they should design courses that reflect authentic situations taken from these CoPs.

Ben-Ari is therefore sceptical to the effectiveness of an entire education formed as an apprenticeship. Yet, for suitable courses, he appreciates the idea of creating authentic environments and situations. This attitude was actually an inspiration for him when he designed a new course book and chose to base the entire book on authentic documents.

> "Professional programmers and software engineers rarely have the luxury of learning from textbooks. They are routinely required to work from formal definitions of protocols, interfaces, languages and architectures" (Ben-Ari, 2004).

Ben-Ari argues that learning activities must be relevant to the intended CoP and he strengthens this reasoning by referring to previous results shown by Shirley Booth (1992). Her results show that the best learning outcomes (within programming) are achieved by those students who take a structural approach, where the programming problems are interpreted in the problem domain rather than in the coding domain. Her advice to programming teachers is to design assignments that force the students to focus on the application domain.

John Dalbey describes an educational project where he used a different way of conducting teaching in a programming course (Dalbey, 1998). The pedagogic idea in this project built on learning inspired by apprenticeship; the students were supposed to learn about programming through working immediately with *far more authentic* software systems compared to the small problems used in the traditional courses. Instead of learning how to write small programs of their own, the teachers introduced the students to the software by giving them simple tasks that did not require extensive knowledge in programming, such as testing and evaluation of the software. Gradually, the students were asked to carry out programming tasks in the software.

The interesting thing in Dalbey's study is the fact that the students were provided with an authentic context and that they got the opportunity to work with *programming in the large* and could therefore study the structure and behaviour of a completely developed system.

Michael Kölling and David Barnes (2004) suggest an integrated model for teaching that combines apprenticeship with problem based learning and case studies. They describe how to do this in a first programming course in Java.

The first student activity is to be acquainted with a software system, which is a game designed by experts. The students explore the software interactively by running it and studying the code using the development tool BlueJ, and they describe the software to peer students. In the next activity, the students discuss design of alternate versions of the game and improvements to the existing software.

The discussion soon moves from details in the code into code quality and maintenance and the students develop the skill of being able to evaluate code critically. Then the students work with exercises that gradually extend the software in different ways. Finally, the students make their own versions of the game as an assignment. During this activity, tutors discuss the solutions with the students with focus on aspects such as maintenance and extendibility.

Two important properties of this way of teaching are the problem driven approach where the interesting concepts appear naturally, and the apprentice approach, that gives the learner opportunities to learn from experts and from doing small changes in the code. It is also important that the exercises and assignments are well defined and at the same time are open to variation and individual extensions. The students should take control and ownership of the tasks and the system they develop.

2.3.2 System maintenance is important

Regarding the professional learning in society, Lave and Wenger are probably correct. A newly employed will not get the same tasks as a more experienced colleague, trainee programmes are often used to introduce becoming managers, and in certain branches, the ancient apprentice model is still used. At some companies, the inexperienced are assigned to some typical beginners tasks. Unfortunately, these assignments are not always selected on basis of their suitability for learning. On the contrary, the choice can rather be based on the low status of the job.

Mirja Kajko-Mattson et al. (2002) have pursued research on education within software engineering and its relevance for the industry. In particular, they focus on software maintenance and conclude that system maintenance is a job for the beginners, whilst the experienced take care of system development. They claim that maintenance has low status and quote Gunderman:

> “Maintenance has been viewed as a second class activity, with an admixture of on-the-job training for beginners and low-status assignments for the outcast and the fallen” (Gunderman, 1988).

However, are the inexperienced able to become acquainted with the software, are they capable to get intimate knowledge of the existing source code and can they understand the underlying design of the system?

> “A trivial change of one line of code to a module implementing common functionality may alter the internal processing of the whole system” (Kajko-Mattsson, 2002).

On the contrary, they argue that software maintenance is an important business that requires high competence in form of skills and formal training. To achieve the needed competence they suggest an education in large-scale software.

> “A highly skilled maintainer is the most important organisational asset pivotal for achieving quality software, strategic for improving maintenance and development processes, essential for remaining competitive and critical for business survival. This requires that universities properly prepare students to enter the maintenance workforce and that maintenance organisations actively build and maintain their human knowledge and skill base” (Kajko-Mattsson, 2002).

2.3.3 Dialogue between university and industry

Letizia Jacceri and Sandro Morasca (2006) point out the importance of a dialogue between the industry and the educational institutions teaching SE.

In other words – there is a need for an exchange of ideas between different CoPs having a shared interest.

The authors identify five possible roles that the industry can take towards the education and that it is very important to make use of them. The role as *students* implies that the universities can arrange continuation training for the employees in the industry. The role as *alumni* (former student) facilitates direct communications channels between the companies and the universities. The role as *researchers* means that companies are interested of sharing the results of empirical research within the education and the industry. The companies can also act as more or less authentic *customers* in student projects.

Finally, the industry acts as *teachers* when its experts share their experiences with students in guest lectures or in other situations. In addition, it is my personal opinion, that the two latter roles connects to the idea of learning in apprenticeship, and that they would contribute to reinforce legitimacy for the education and help both students and teachers to gain insights in the professional view of the subject field.

Experiments with educational collaboration between industry and universities can be found at several places. For example, the CS department at University of Gävle has good experiences from a one-semester course where the first half of the course contained studies of advanced applications using Java, project methodology, and guest lectures from various consulting agencies in the IT business. The second half consisted of independent student projects with advisors from both academy and industry. The tasks were primarily authentic orders from customers from the IT-sector, but also from other enterprises.

Rayford Vaughn (2001) reports results and experiences from a similar model at Mississippi State University, where students work in authentic projects towards authentic customers in an industrial environment. Their experiences are mainly good and both the students and the customers are happy with this way of working. The *deliverables* that was the basis for examination was a conceptual model of operations, a specification of system requirements, a design document, a test plan, system documentation, and a *formal delivery to the customer*.

2.3.4 Companies' strategies for obtaining education

Eskil Ekstedt (1988) means that the early IT-companies have moved on from being manned by computer nerds that focus on technical solutions and nowadays the companies concentrate on supplying overall IT-solutions that aim to increase the efficiency within corporations. This requires knowledge that reaches far beyond the horizon of pure programming, since the developers must be familiar to the various processes in different organizations. In addition, the companies must constantly maintain their knowledge base in a never-ending process.

In these companies, the educational level is exceptionally high and they recruit staff from people with academic degrees or long professional experience. The companies use three principal recruitment strategies. The first strategy is to look for well-educated professionals, mainly engineers. The second strategy is to search for persons with good skills in programming and computers in general, but this tendency has decreased because the companies offer in-service education with extensive programming courses anyway. The third strategy is to employ young inexperienced people having an academic degree. In this case, the companies regard the education process as a filter; a person with an academic degree has proven his capacity and in addition, he contributes to the company's status. Regardless of the chosen strategy for recruitment, the internal education is very important for these companies and also the internal research and development (Ekstedt, 1988).

2.4 The interface concept

I have chosen the students descriptions of the Java interface concept as the central phenomenon in this study. I consider this concept most interesting and important when it comes to aspects of programming in the large. There is support for this view in other studies that deals with conceptual understanding of object-orientation.

Miguel-Ángel Sicilia (2006) has analysed his experiences from teaching object-oriented programming with Java during the period from 1997 until 2003. He concludes that there are problems in the understanding of the conceptual knowledge layer regardless if the teachers take a modern approach such as *objects first*, or the more traditional attitude *procedural first*.

With the conceptual knowledge layer, he considers problems within object-oriented design, in contrast to the problem of learning the specific Java syntax. He claims that when teachers use the *Unified Modelling Language* (UML) to model the design of software, the principle should be to use *instances first* and focus on groups of objects and relations between objects, and then later generalize them to classes. If teachers and novices model using class diagrams, the solutions often tend to be too abstract for a smooth transformation from the model into computer programs. Modelling with objects also gives a better understanding of what happens in run-time.

Later, when the students have conceived the fundamental principles for the object-oriented way to structure programs, it is feasible to introduce new concepts such as inheritance and interfaces, motivated by their possibility to extend, reuse and generalize the existing software. Teachers should introduce polymorphism as generalization and as a way to solve the absence of built-in generic types in Java.

Although it is possible to argue for or against them in an introductory programming course, Java interfaces are very important in order to explain one of the principal lessons of object oriented programming; the separation between specification and implementation.

> "It is difficult to provide novice students with a full understanding of the role of a professional designer, but it is at least possible to describe design situations that emphasize producing design structures with certain quality characteristics, such as reusability or minimal coupling" (Silica, 2006).

Sicilia points out that it is difficult to construct good pedagogical examples of how to use interfaces. However, he gives general guidelines and gives some examples from his own experience.

Schmolitzky (2004) argues for an introduction of Java's interfaces before starting with sub-types and inheritance. He summarizes three good reasons for his standpoint:

- To emphasize that a server's interface seen from the client's perspective should be independent of its implementation and that a built-in feature of the programming language Java supports this principle.
- To (earlier) introduce and practise the powerful concept: "*program to an interface, not an implementation*" (Gamma et al., 1995, p.18).
- To learn avoiding the common mistake to include private members of classes in the documentation since there are no private members in an interface.

In summary, he concludes, that interfaces should be introduced as soon as possible in the courses. After an evaluation of accomplished courses, there is evidence for that the students have gained a better understanding of the concept interface and that they are more confident in the use of the mechanism interface.

Friedrich Steimann, Wolf Siberski and Thomas Kühne (2003) call attention to the fact that the Java interface is often (mis)conceived as a means to utilize multiple inheritance. They claim this explains why the Java interface is used so sparsely in teaching despite of its potential to design highly independent code.

Programmers should use interfaces to declare reference variables to objects instead of using the explicit class types. Using this method, the programmer will attain the advantage of being independent of specific implementations. Thus, the dependency is limited to a mere specification, which allows a variation in how the specifications are implemented. This principle is especially important for developing frameworks and in component based design. In this context, the application programmers' classes can be compared to specially designed plugs (plugins) that must fit in the corresponding sockets. The interfaces correspond to the sockets and they specify partial types that describe some aspects that can be implemented by one ore more classes. In this way, using an interface variable, it is possible to connect to several different class instances, which all have the specified aspect. On the other hand, one class instance could be connected to several interface variables and be used from their specific aspects.

After an analysis of software, it is concluded that interfaces are not used to the expected extend. This fact is explained by the considerably large effort that is required of the programmer. Moreover, the intuitive conception of the interface is still weaker than the class concept. Therefore, they advocate a different conceptualization of the notion of interfaces.

In programming courses, the concept of *roles* should be emphasized over the idea of *natural types* since roles are possible to identify in the problem domain in the same way as natural types (classes). Roles are partial types and they have a natural connection to interfaces. The authors give a distinct method to separate the cases for when to declare variables as roles (interfaces), when it is proper to declared them as concrete types (classes) and when it is more feasible to declare them as polymorphic types through inheritance. In addition, they provide a set of rules for how code can be refactored to utilize interfaces and they give a suggestion for how to measure the soundness of the utilization of interfaces in specific software (Steimann, 2003).

3 Phenomenography

Ference Marton and Shirley Booth point out that people do things differently, and they learn to do these things in different ways; some do it worse and some do it better (1997). Phenomenography originated in educational questions of how learning comes about and how it is possible to improve the learning process. Amongst other things, Marton and Roger Säljö were interested in deep and surface approaches to learning and gave contributions to that field of research (e.g., Marton and Säljö, 1976a, and 1976b).

It gradually evolved and matured into a research tradition that concerns how different aspects of the world appear to some group of people. Essentially, the studies within this approach are explorative and use empirical data, and they all take a second order perspective on some phenomena. That is to say, the phenomenographer does not study the phenomena as what they are (the first order perspective), but the variation of what they are as experienced and expressed by people (the second order perspective). Ference Marton, one of the pioneers of phenomenography, gives the following definition of this research specialization:

> "Phenomenography is a research method adapted for mapping the qualitatively different ways in which people experience, conceptualise, perceive, and understand various aspects of, and phenomena in, the world around them" (Marton, 1986b, p.31).

Consequently, the object of study is the relation between a certain phenomenon and a group of people and the variations of the relation. It is neither the phenomenon nor the people it tries to explain; it is the group's experience of the phenomenon. The ontology of phenomenography is non-dualistic, which means that it does not separate the observer from the observed (object and subject). Marton (2000) explains it in the following way:

> "There is only one world, a really existing world, which is experienced and understood in different ways by human beings. It is simultaneously objective and subjective. An experience is a relationship between object and subject, encompassing both. The experience is as much an aspect of the object as it is of the subject" (Marton, 2000).

In this non-dualistic world, the set of different ways to experience an object is what actually constitutes it. Moreover, because the experiences all relate to this constitution, they are all logically related. A prominent feature of phenomenography, compared to other qualitative research traditions, is thereby, the way in which the results are structured.

Experiences from earlier studies had shown that different people described phenomena in only a few different ways, and that led to the fundamental epistemological assumption, namely that there are only a limited set of qualitatively distinct ways to experience and describe a phenomenon. Each qualitatively distinct way to experience forms a *category of description*. In addition, there are always a set of *logical relations* between the categories, and the logical structure in combination with the categories of description constitute the *outcome space*. In this way, the outcome space contains a rich set of information of how the phenomenon is experienced and how these experiences relate to each other.

Moreover, there is no explicit connection to the experiences of any individual person in the outcome space. Each category describes a particular way to experience a certain phenomenon, *observed in the collective*, and is thereby constituted by merged fragments of meaning found in the individual's description of the phenomenon. The collective outlook is a quality that distinguishes phenomenography from qualitative research in general, which is often described as taking the *individual's perspective* (Denzin, 1994).

Marton (2000) actually refers to the outcome space as a synonym for the phenomenon, and this emphasizes his non-dualistic view. However, there are researches from other traditions that do not appreciate this presupposed ontology. John Richardson believes that the non-dualism in phenomenographic research is problematic and advocates a closer association to the constructivist approach (Richardson, 1999, p.68).

As in its origin, the most common application for the research approach is still to study different aspects of learning and teaching. However, phenomenography is not restricted to that area only. John Bowden (2000) divides the research approach into two forms: the applied (or developmental) form, and the pure form, separated from institutional learning environments.

> "Phenomenographic research methods of data collection and analysis can be used to study a range of issues, including approaches to learning, approaches to teaching, understanding of scientific phenomena learned in school, or understanding of general issues in society unrelated to educational systems" (Bowden, 2000).

Marton and Booth emphasize that all of the frequently used terms in phenomenographic publications, such as "conceptions", "conceptualizations", "ways of understanding", "ways of comprehending", are all synonyms for "ways of experiencing". One should not understand them as referring to the internal mechanisms in the human brain. The phenomenographic researchers always allude to the experiential sense of the words, all in line with the non-dualistic approach (1997).

3.1 The research process

John Bowden (2000) outlines the phenomenographic research process as having the four stages: plan, data collection, analysis and interpretation. In all of these stages, the researcher must maintain focused on the purpose of the study. This is important to consider for obtaining trustworthy results. As in all research, it starts with a plan that defines the purpose and the strategies. Naturally, what drives the research is an underlying question that the research activity tries to answer. Students' difficulties in coping with physics gave Bowden a good reason to try to make sense of the students' understanding of important concepts in physics.

The input data for a phenomenographic investigation is essentially people's statements of experiences of a phenomenon. The predominant method for collecting this data is through interviews with people, and the researcher must select the persons carefully and consider why they are a good choice. Another issue is who the interviewer should be. The interviews pose open-ended questions that address the problem area or ask the subject to explain what the phenomena X is.

Even though the interviews should be planned on beforehand, they can take different directions and follow the spontaneous thoughts that might appear differently from case to case. The next phase is the analysis of the data, which often starts by transcribing the recorded interviews. The texts are then sought for different meanings and the contexts they appear in. Sometimes phenomenographers de-contextualize the fragments of meaning, and sometimes not. In either way, the meanings constitute a pool, from which the categories are condensed.

The categories should relate to each other logically. If not, the researcher should reconsider the data again. Section 3.2 elaborates more on the analysis process. Finally, the results should be interpreted according to the purpose of the study. In applied, developmental phenomenography, the interpretation is a natural consequence of the posed research question. If the result tells how students experience phenomena in an educational context, the teachers can use the results to enlighten their pedagogy and instruction. They can adapt their way of how they present new concepts, or they can get a better understanding of why students fail to do certain tasks.

A pure phenomenographic study, on the other hand, might have only the purpose to describe the experience of a phenomenon, without any further implications. In all cases the results of the study must be seen in the light of its purpose, and if a researcher wants to use the results in a different context, this issue must be taken in consideration.

3.2 Phenomenographic analysis of interviews

In phenomenographic analysis, the researcher refines the primary source of data by transcribing recorded interviews into a textual form where the participants' quotes are anonymous. However, it is still possible to separate individuals by using pseudo names. The next step is to search the texts for different expressions of meaning that relate to a certain phenomena.

> "Phenomenographic analysis – whether it is seen as construction or discovery – focuses on the relationship between the interviewee and the phenomenon as the transcripts reveal it" (Walsh, 2000).

These manifestations of meaning can be identified in several places and different forms in the text. Meanings are found where the interviewee explicitly describes her experience of the phenomenon as such. However, implicit descriptions can also reveal meanings, as in descriptions of how she uses the phenomenon, or which purposes, advantages or drawbacks this phenomenon brings about.

The meanings are expressed by quotes that form a large collection of further refined data. The quotes are usually de-contextualized from the text, but the reference to their context should be kept, to maintain the possibility to re-interpret their meaning. The purpose of making the de-contextualisation is to be able to, on a collective level, find qualitatively different meanings, experiences and understandings of the focused phenomena. Some meanings stand out from others, whilst some have something in common with other ways to experience.

The fragments of meaning are in this way condensed into clusters of meaning that are abstracted and outlined in categories of description. It is important to understand that the categories do not express any particular individual's understanding; rather they are the result of an analytical categorization of the meanings found on the collective level. In the process of forming categories, one should search for different dimensions in such a way that each category opens a new dimension of understanding the phenomena and its meaning. This avoids categories that are instances or variations within the same dimension.

The phenomenographic outcome space is distinguished by the categories of description and their mutual logical relations, usually the hierarchic inclusiveness which implies that the meaning of the categories include each other in the sense that a certain understanding also includes or implies a similar, more elementary understanding. As phenomenography originated in studies that in one way or another aimed to understand or improve institutionalized learning, it is reasonable to range the outcome space in a hierarchy where the quality of each category is valued by some measure of compliance to the educational goals.

> “Thus, we seek an identifiably hierarchical structure of increasing complexity, inclusivity, or specificity in the categories, according to which the quality of each one can be weighted against that of the others” (Marton & Booth, 1997, p.126).

Consider the structured and interrelated outcome space in contrast to sociological research traditions where it is usual to make categories without any requirement of internal logical relations. It is important to emphasize the clear and inevitable relation between the result and the subject field that includes or views the phenomenon. This is an argument for taking a phenomenographic perspective in educational research within a specific subject field.

On the other hand, it is vital to make clear that the analysis is not a matter of sorting the subjects’ conceptions into a predefined structure. One of the fundamental epistemological assumptions within phenomenography is the relations between the categories of description. The various ways in which a phenomenon can be experienced are logically connected to each other through the phenomenon itself and the structure of the logical relations is typically hierarchically inclusive.

Another property of the outcome space is the collective level of descriptions constituted in the categories. It is not the case that all individuals or a specific individual have a certain structure of their way to experience. Rather, the analyst tries to constitute the categories on a collective level and if successful, the well-founded categories can be structured and interrelated. This is what the phenomenographic researchers are striving to achieve.

Marton and Booth (1997) describe three principal criteria for the expected properties of an outcome space constituted of categories of description. The first criterion is that each category should have a distinct and unique relation to the phenomenon according to a distinguished way of experiencing it. This is motivated by the fact that phenomenography is a pedagogical research specialization, focused on learning, with the goal to obtain a clear picture of qualitatively distinct ways to experience phenomena that have a connection to learning.

The second criterion is that the categories must have a logical relation to each other that often is hierarchical and often is inclusive as well. From a pedagogical perspective there is a norm that defines which ways of understanding (experiencing) a certain phenomenon is preferable before others. The pedagogical goal is often that the learner should be able to experience phenomena in a more extensive, complex or specialized way and therefore a hierarchical structure of the categories is sought that corresponds to this goal.

The third criterion is that the system of categories should be as compact as possible. This implies that there should not be more categories than necessary to express the critical experiences and the differences between them. (Marton & Booth, 1997, pp.124-126).

3.2.1 A definition of inclusive categories

As discussed in the previous section, the hierarchic structure of the outcome space can be explained by inclusiveness of conceptions. Inclusiveness in this context means that a certain way to experience a phenomenon also includes another way of experiencing it.

However, the description of inclusiveness is a bit vague and we need to consider what it means to say that one way of experiencing also includes another way. I propose the following definition of inclusiveness for this study. Given that there is a category, A, that codes a particular way to experience and describe a phenomena. Then category A is included in another category, B, if their relation fulfils the following conditions:

- There is a non-contradictionary relation between category A and B, and
- The relation is of the type *B consists of A*, or *B is an augment of A*, or
- Something in *B assumes A.*

During the data analysis, we used this definition to study and establish the categories' internal relations and plausibility related to the other categories in the outcome space.

3.3 Questions of trustworthiness

As in all research, the phenomenographic researcher wants to be heard and believed by other researchers and receivers. The fundamental condition to achieve this goal is to uphold trustworthiness and to deliver credible results. In qualitative research, it is crucial to show that the chosen research methods reflect the goals for the research in a suitable manner, and to show how to use the results.

In her dissertation, Shirley Booth (1992) discusses these matters through accounting for her own long experience of programming and her familiarity and good relations with the students who participated in her study. She describes the exhaustive and open-ended interviews and declares that the transcripts are open for other researchers to read. In this research, there are no absolute truths and therefore, she claims, the researcher must argue convincingly for the chosen methods, the results, and the interpretations. This can take place in seminars, presentations and by peer-reviewed articles within the research tradition.

She explains that, due to the collective level of the results, the interviewees seldom confirm credibility; the individual's experience is not traceable, and the researcher's interpretations goes further than the individual's understanding at the time of the interview (Booth, 1992, pp.64-69, 90-92).

The phenomenographic analysis has a subjective nature since it reflects the researcher's way to discover and experience the meanings within the text material. For instance, Eleanor Walsh (2000) discusses openly and critically concerning the logical relations between categories and whether the categories are discovered or constructed and what the difference is between these approaches within the phenomenographic analysis.

Gerlese Åkerlind (2005) has analysed what is common and what is varying in the conduct of phenomenographic analysis. She pursued this by studying other researchers' descriptions of data analysis in their publications. The purpose of this investigation was to collect descriptions of the analytical processes and the methodological procedures used to ensure quality and consistency in interpretation of data, in a single place.

Åkerlind points out that there is a prevalent ignorance of the variations within phenomenography and this fact could strengthen the arguments from sceptics. In addition, there are only a few explicit descriptions of how to accomplish the phenomenographic data analysis and this is one of the reasons for the critique[7] of phenomenography. In her study of applied phenomenography, She finds a common conduct in the analytical process when it comes to keeping an open mind, suppress own preconceptions, focus on the whole, the search for variations in meanings and relations between them and the iterative process using re-structuring and tentative categories.

However, Åkerlind identifies areas within the analysis where there are variations in the practice. Some use de-contextualized fragments of meaning and others do not. The collected data is handled in different manners and the principle of letting the logical structure follow the data as close as possible is sometimes compromised by the desire to reflect the researcher's professional classifications. The collaboration with other researchers varies from individual analysis to collaborative analysis (Åkerlind, 2005).

Yvonne Lincoln and Egon Guba discuss trustworthiness within qualitative research and argue that this research, in contrast to positivistic traditions, is inevitably associated with subjective values (Lincoln and Guba, 1985, pp. 37-38). Hence, in order for the researcher to be trustworthy, in the sense that the audience thinks it is worthwhile to take part of the results, it is of great importance to communicate how the researcher is reasoning and to account for both the data and the researcher's interpretations. They point out that trustworthiness cannot be judged by the same measures as in the positivist

[7] See for example Richardson (1999) who critically scrutinizes phenomenography from a constructivist's perspective.

science tradition. Instead, they suggest four alternate terms that replace the traditional terminology.

> "The four terms 'credibility,' 'transferability,' 'dependability' and 'confirmability' are, then, the naturalist's equivalents for the conventional terms 'internal validity,' 'external validity,' 'reliability' and 'objectivity'" (Lincoln and Guba, 1985).

The *credible* researcher should persist long enough to ensure that a sufficient and unbiased amount of data is gathered. The critical aspects should be studied in depth using different angles, various data sources, methods, researchers and theories (*triangulation*). The raw data must be available for re-examinations and the informants should comment on the results.

The descriptions should be so rich and thick that someone who is interested in making a *transfer* to another context should be able to decide if that is possible or not. The data sources should be selected to maximize variations.

The *dependability* of the results can be increased if the research group is split and each sub-group deals with data independently and compare the results (*stepwise replication*), or an auditor could examine the data, the process and the produced results and then see if the conclusions are similar (*inquiry audit*).

It should be possible to conduct a *confirmability* audit trail by reviewing the raw data, the analysis and synthesis, and the documentation of the process and other documents. This ensures that the results are products of the conducted investigation and are not products of the researcher's preconceptions (Lincoln and Guba, 1985).

Mulholland and Wallace (2003) suggest a method to present results from narrative studies in a legitimate and trustworthy manner, by dividing the presentation in three stories. The first story shows the strength and credibility of the study. It contains a story told by the subject of the research – the data. The second story is told as the researcher's interpretations of the first story. The third story adds a theoretic model to the first two stories. The researcher give suggestions for how the experiences of the inquiry can be useful, for example, how they can improve educational matters for the participant, the researcher and the reader. Finally, the researchers account for how the study has influenced them selves and their view on teaching. Moreover, the first two stories can be re-read using the new perspectives gained from the third story.

We can conclude that a very important asset in obtaining trustworthiness in qualitative research is a rich set of data that can be shared with the reader in various ways, together with the researcher's interpretations and conclusions.

3.4 Will the outcome space become complete?

The outcome space of a phenomenographic study is a categorization of several descriptions of a phenomenon, regarded at a collective level. What it shows is a model of descriptions of how people experience something – a second order perspective. It does not describe what the phenomenon is in it self - a first order perspective. Gerlese Åkerlind claims that the outcome space always will be a subset of the hypothetic complete set of ways to experience a phenomenon, and that in reality; there are only more or less complete spaces. However, every single outcome space can provide an important contribution to the big picture of how certain phenomenon is experienced (Åkerlind, 2005, p.10).

The present study focuses on students' experiences of phenomena in a context that will emphasize some aspects and perhaps suppress other possible aspects. Thus, it can be expected that the conceptions will be influenced by the context – a bias that actually is intended. We know that the context probably will affect the students' descriptions simply because some aspects will not be considered as relevant in the prepared setting. We do not search for a complete picture that reflects the universal view of the phenomena. We search for the conceptions as they appear in a specific context. Hence, these circumstances must be considered when the results are interpreted.

4 Conducting the empirical study

The study reported herein, concerns how students describe their experiences of working with a realistic software engineering task. The purpose of putting the students in such a situation was to supply them with a context where a number of interesting concepts appeared naturally, and was part of the system design. In this manner, we could gather information about how students conceive concepts that we believe are important for their future professions. In addition, it is of interest how they interpret their commission, how they act during their work and if they have learnt anything by their participation.

This chapter describes the design of the empirical experiment; the people involved in it, the task, data collection and analysis.

4.1 Who are the students?

The participants in this experiment were all students at three-year study programmes at University of Gävle[8] in Sweden. The students were at their second year in either the Computer Science programme or the Computer Engineering programme. From the point of view of content, these educations are relevant to study because they both emphasize programming and system development.

Furthermore, we assume, most of the students from these programmes aim at industrial careers starting as programmers, system developers, computer engineers, or something similar. This assumption is motivated by the fact that only a few students stay in academia, and by the informal feedback that we get from alumni. Accordingly, in contrast to students from other study programmes who are forced to take a mandatory programming course, we expect that the selected students will get professional use of their knowledge and skills in programming.

Many of the courses are given to students from both study programmes, and this was the case for the course *Object-Oriented Programming I.* This course deals with the concepts of which we want to investigate the students' experiences. Earlier during the autumn semester, the chosen students had studied this course and for the majority this was their second or third course in programming. During the experiment and the data collection, many of the

[8] The Swedish name is *Högskolan i Gävle*.

participating students studied the course *Algorithms and Data Structures* in which they also used an object-oriented language for their assignments. During the first year, the contents relating to Computer Science are the same for both programmes, and it is partly true for several courses in year two and three.

The differences lie mainly in the subsidiary subjects, where the engineers study more mathematics and technical courses, such as digital design and embedded systems, whereas the Computer Science students gets a classical bachelor degree with a higher degree of freedom when it comes to the subsidiary subject, which for example could be economics, psychology or geographical information systems.

Most of the students study quite a few courses that, one way or another, contain programming. The Computer Science programme has a clear direction towards information systems with courses in data-bases, system development, system maintenance, whereas the computer engineers often choose to get deeper in for example operative systems or compiler theory.

In this type of qualitative study, based on interviews, it is appropriate to select individuals that represent different groups of students in order to get the opportunity to maximize the variation of ways to experience and describe experiences (Booth, 1992, p.58; Marton & Booth, 1997, p.124). The selection process started by giving information about the study to all of the students that completed the previously described course *Object-Oriented Programming I*, and asking them to take part of the study.

From the 32 students who completed the course, 11 students in total agreed to participate, and therefore, it was not possible to make a selection. Still, the volunteers represented both of the described study programmes. Their study achievements in the course were evenly distributed; half of them passed, and the other half passed with distinction. Hence, their ability to solve programming problems and design problems should vary, but still have a lower bound.

There were only two female students among the interviewees, and two of the participants were first generation immigrants speaking fluid Swedish. However, this approximately reflected the proportions of these groups in the class, both between males and females, as well as between native and immigrated Swedes.

To protect the participants, their real identities are not revealed in the text. Nevertheless, they are provided with fictitious names in order to give the readers a more personal impression. You are now going to be acquainted with Alf, Bea, Cia, Dan, Eva, Fia, Git, Hal, Joe, Ken and Leo[9], who so generously lent their voices to this study. We hope that you will get an understanding of the world of programming as they described it.

[9] In the text, quotes by the interviewer will be preceded by the abbreviation: *Int*.

4.2 Data collection – the experiment

The purpose of the experiment was to investigate how a group of students experience (conceive) and handle various aspects of object-oriented programming in a situation where they act as system developers committed to a realistic ("authentic") software engineering task.

What constitutes the novelty of this situation is how they should tackle a problem in a piece of software that is larger, less arranged, and more realistic, compared to the normal examples in traditional programming courses. The engagement in the work with this system gives the students and the researcher the possibility to contrast interesting aspects against a common background.

The complexity of the software environment motivated the use of abstract concepts, such as polymorphism and interfaces. Would their natural appearance stimulate to discussions with the students about abstract concepts in concrete terms, or would the system's complex environment "conceal" the abstractions? We hoped that the experiment would stimulate to a comprehensive and varied experience of different concepts. One of these concepts is the notion of plugin modules, which the system utilized extensively.

The Java interface is an example of a concept full of nuances; people probably experience (conceive) it in many different ways. A course that involves object-orientation often treats the interface concept theoretically, and it can be problematic to supply the students with a larger context that they can relate to the concept. Our intention is to focus on the understanding and use of interface in a situation where students are working with relatively extensive and complex software. In this context, the students cannot avoid to relate to the interface concept, one way or another, since the software mainly was designed using interfaces as a bearing idea.

Hence, the purpose of this study is not only to understand how students experience and describe certain concepts alone; it is also how the students experience their purpose and role in a situated context. Through the problem-solving situation, the students should be stimulated to try to grasp an entire object-oriented system where the use of abstract concepts such as interface is an important part of the construction and functionality of the system.

4.2.1 Description of the system

Exclusively for this investigation, we designed a flexible software system with a dynamic behaviour that taken together with its structural complexity motivated a consistent use of the Java interface.

The architecture constitutes a framework[10] that builds on a fundamental design principle for how to handle new functionality dynamically through an extensive use of various interfaces. Partly, the system consists of a server (software on a machine) that is connected to a database, which contains administrative information about courses and students in a school. An unlimited amount of users can connect themselves to the server through a client software.

The principal idea of the design of the system is that the client software should be kept as independent from the server software as possible, which implies that both parts should be robust to changes. A potential change introduced to the server software, should not impose a corresponding change in the client software, even if its run-time behaviour could change drastically.

As mentioned earlier, Java interfaces are used frequently in the software, and the theme of purpose in common is to separate the implementation from the specification. The following text will give some examples of how interfaces were used in the system and explain the underlying ideas.

The first type of use of interfaces defines the possible communication between the client and the server parts of the software, that is, a specification of a number of operations that the server offers. Such an interface used together with Java's Remote Method Invocation protocol (RMI), enables that calls, to any of the operations specified by the interface, can be initiated (invoced) from the client machine, and that they are actually executed on the machine that runs the server software.

The second way to use interfaces is partly to reduce the dependency to specific implementations, and partly to delimit the clients' authority to affect data objects. It is important to consider data integrity of objects and the idea is that entities that derive from the database should be created exclusively by the code that handles the database. No other part of the system should be able to access that code, due to object data consistency. For example, by giving public access to the interface *Person*, but only giving a restricted (private) availability to the implementing class *PersonImpl*, it is secured that only the owner of the class can create its object, since interfaces cannot be instantiated. The creation of data objects is thereby well defined and strictly localized. On the other hand, all parts of the system are free to *use* the created objects. An essential property for a person object is to define its identity[11] only in connection with its creation, and to inhibit all attempts to change it later on. This technique prevents unauthorized production of fake persons, and it assures that every person object is reflected by the database.

[10] A framework is an underlying software system that offers developers the possibility to use its components, functionalities and strategies

[11] For example the Swedish "Birth Number," the British "National Insurance Number," or the "Social Security Number" used in the USA.

The third way of how interfaces are used in the system is to achieve polymorphism and dynamic behaviour. When the client software begins to execute, it connects to the server and fetches a number of objects from it. The common aspect of the plugin objects is that they all implement the interface *PluginPanel*, and that they all implement user interfaces for various use cases of the client program. The client side is not aware of the concrete type of the received objects; rather, it regards them only as *PluginPanel*-types through using the associated interface. In this manner, it is possible to handle completely new use cases to the system by designing new plugins. These can be added to the system without having to change or affect the client software at all – only its appearance to the user is affected. There is no need to recompile the client software, and there is no need at all to restart the client programs – it is even possible to add new functionality during operation.

The students' task was to use the dynamic properties of the system in order to introduce new functionality to the system. To be specific, they should design a new plugin that was supposed to handle registration of students on a certain course instance.

4.2.2 Carrying through the experiment

The data collection was accomplished during three months from December 2003 until February 2004, and for this sole purpose, a particular office room was reserved throughout the whole period.

The workroom was prepared and furnished with a swivel chair and a desk on which there was a computer display and a keyboard. The computer unit was placed behind a bookshelf in order to reduce the background noise and to conceal the tangle of cables that connected the computer's graphic card, its sound card, the videocassette recorder, the minidisc recorder and the microphone.

Thus, the videocassette recorder could capture anything displayed on the computer screen and record all sounds in the room during the experiment. In the room, there were also materials at hand, such as a writing-pad, pencils, and technical literature on Java. The room was also equipped with an extra chair for the interviewer.

When the students, the subjects of the experiment, had presented themselves at the intended office, they were informed about the experiment and its character of a role-play where they were expected to act as a newly employed developer (programmer) at an IT-company. Furthermore, they were informed that they were about to accomplish a software development task at the computer and that they could spend at most two hours.

In addition, this would be followed by an interview that would take about one hour. Each student was provided with two help vouchers that could be used to get help from a senior colleague (acted by the experiment leader). The purposes of this procedure was to prevent students from being totally stuck, at the same time prevent them to ask questions to soon and encourage them to keep going on their own as far as possible.

One of the task's major difficulties to the students was to understand what the mission was all about, and hence, it was very important that the student were not prepared in any way. Therefore, the student was asked to be discrete and not reveal to other participants what was going to happen during the experiment.

After the information, the role-play started as soon as the student had taken place in front of the desk and the experiment leader had started the recording equipment. A letter from the manager, that informed the newly employed about the present situation, lay on the desk. Apparently, a senior programmer had taken ill and the work he was presently doing had to be completed as soon as possible.

In the letter there were instructions for how to get on, starting by finding a document on the computer that described the project and the system in detail. This documentation also made clear what was already implemented and what remained to be accomplished before the software was complete. Appendix D contains parts of this technical documentation.

The video recordings give a precise picture of what activities took place on the computer and in which order they were executed. This information will not be further analysed in the present study, but we hope to be able to process it in a continued study.

4.3 Data collection – the interviews

Immediately after the role-play and the work with the task, the experiment leader made an interview with the student. At that time, all of the circumstances surrounding the experiment were still fresh in the students' memory.

During the two hours of hard labour, there were many opportunities for the student to encounter different concepts, such as interface. It was hoped that the student had noticed them and perhaps even reflected upon them. Since the purpose was to investigate how the students described their experience of these concepts in connection to a situation of programming or problem-solving, it should be appropriate to accomplish the interview as soon after the process as possible.

The interviews were semi-structured and based on a few prepared themes and questions. In this interview technique, it is vital for the interviewer to be sensitive to the interviewee and to come up with follow-up questions in response to answers, and it is not possible to make a detailed plan for this (Marton & Booth, 1997, pp. 129-132). The interviewer must be prepared to rearrange the order of the questions and to catch the student's spontaneous reflections. Appendix C describes the planned themes and questions.

4.3.1 Doing the interviews

When the student had accomplished the imposed programming task, the interview was carried out at once. There was a predetermined time limit of two hours for the mission, and therefore it was interrupted even if it was not fully completed. In the end, it turned out that all students used their full two hours, and that some of them would like to continue until completion. In order to maintain the context from the job, the interview was carried out at the very same office where they had worked, and the computer was kept on as a resource that could be utilized during the interview when the student wanted to show or discuss something, such as source code and other documents.

The interviews were fairly extensive and their length varied between 45 and 60 minutes. Only the interviewer and one interviewee participated in the interview, and the conversation was recorded on a minidisc recorder. As described above, the interviews were semi-structured, and in most cases, the interviews took detours towards topics that were not planned at beforehand. Sometimes the discussions took interesting turns, and sometimes it led to dead-ends that treated irrelevant matters.

The interviews were thereby very dynamic in the sense that they did not cover exactly the same questions; however, all interviews covered the prepared set of themes. During the interviews, the students expressed that it had been fun and stimulating to do the assignment, but it required much of hard work.

4.3.2 Transcription

The interviews were recorded on minidisks. However, the recordings themselves are not suitable to use when the researcher wants to analyse what the students said in the interviews, at least not in the phenomenographic research tradition where it is preferred to analyse data in a textual form. For this purpose, a transcriber must first listen to the recordings carefully, and then transcribe them into text as faithfully to the original as ever possible. In this study, the interviews contained many technical terms that were possibly hard to interpret for a person who was not familiar with the technical knowledge. Hence, I decided to transcribe the interviews myself, and in return, I became familiar with their contents before reading them.

The transcriber aimed at representing the linguistic expressions that came out in the interviews by imitating sounds and using spoken language as extensively as possible. Hence, there are often grammatical errors, logical errors, and incomplete sentences in the transcribed quotes.

4.4 Analysis of the collected data

The most clearly defined questions in this study are the ones that illuminate the students' experiencing of various concepts; both commonly used terms within object-orientation, as well as concepts that were specific for the students' mission. The phenomenographic analysis is well suited for this type of questions, and how this analysis was accomplished is described in section 4.4.2.

When it comes to the analysis of the students' course of action and which strategies they used, it was not feasible to use a phenomenographic approach. A process like the one the students have lived through during the experiment consisted of a long series of thoughts and actions, which not in any way could be described as a phenomenon. This analysis was therefore achieved by a scrutiny of the collected data and the reasonable conclusions that could be made from it.

4.4.1 Expected results

One of the purposes of this study was to establish an image of the qualitatively different ways of how the students experienced different concepts, related to the commission in which they were involved. The selected concepts were: (1) the commonly used concept *interface* in Java, (2) the concept *plugin* (not covered in programming courses), and (3) the specific *system* that the students tried to complete.

The primary results of the phenomenographic analyses are expected to be outcome spaces consisting of categories of description and their interrelations. These results will then be further analysed, interpreted and discussed and we hope they will contribute to a deeper understanding of how students experience these concepts and perhaps why they have these ways to experience. The results and the study per se could also promote for a discussion on how teachers could use the results when they teach the concepts or other similar concepts within the subject area.

4.4.2 Conducting the analysis

The first step of the analysis was to read all of the transcribed interviews two times to get an apprehension of the whole context, and thus, to get a broader perspective. During the reading, all text sections that related to the selected concept, interface for instance, was marked.

The next step was to collect all of the marked text sections and copy them into a separate document, and then we imported the document to the computer based analysis tool Atlas.ti (Muhr, 2004). This tool did not automatically analyse the data in any sense; however, it made the work with the text easy. It enabled us to browse through the text, to add comments, and to mark those quotes that in some sense ascribed a meaning to the phenomena in question. One or more labels to identify the interpretations of the underlying type of meaning then coded each marked quote, and it was convenient to use the software to consider the data from several perspectives. For instance, it was easy to find and overview all quotes coded with a certain label. On a higher level it was possible to study the various codes of meaning through an alternative view, where the labels was represented as graphical symbols, structured as nodes in a graph.

In addition, the software allowed us, to insert logical relations between the nodes in the graph, such as “depends of,” “is an,” and “is part of.” The various codes of meaning were analysed from a perspective of finding qualitative similarities and differences between them, and hence, different clusters of meanings were condensed. Before these groups were eventually transferred into categories of description, they were further scrutinized using another perspective, namely the requirement that each category should open up a new dimension in the phenomenographic outcome space.

In the notion of “qualitatively distinct categories,” it is understood that each category should open up a new dimension of ways to understand the phenomena at question (Marton & Booth, 1997). That is to say, that it is not desirable to have categories that constitute values along the same dimension; The new category should rather describe a dimension of meaning that expresses something qualitatively different from the other categories.

To illustrate this, we can imagine that we are analysing transcripts of interviews about different persons’ experiences of food. We find many interesting utterances, such as: “I think pizza tastes nice,” “fish are repulsive,” “potatoes are nutritious,” and “fast food is no good for you.” In this case ”tastes nice” and ”are repulsive” could be regarded as values along the dimension taste/smell, whilst the values represented by ”nutritious” and ”no good” could reside along the dimension wholesomeness. When we establish the categories of description, we use these dimensions as a starting point rather than from the individual values of meaning found in the text. In this manner we present our results, namely that food is described as “something that has taste and smell” and “something that affects our bodies and wellbeing”.

The analysis process included the reciprocal relations between the categories, and the preliminary categories, synthesized after the early text analysis, and now regarded from the relational perspective, was now arranged in a logical structure based on two criteria.

Criteria for the logical structure:

- One criterion is based on an evaluation of the categories’ compliance to the educational goals. The objective of the education that the students are undergoing is to obtain competence within the subject; and therefore, experts in the subject, can accordingly estimate and compare the categories.

- The categories are related to each other hierarchically. Inclusiveness and dependence are examples of such relations. See section 3.2.1.

Seen from a phenomenographic perspective, the outcome of the analysis is potentially successful if the same structure is obtained when both of these criteria are applied.

5 Descriptions of the concept Interface

This chapter deals with how the students described the interface concept in Java. As accounted for in the previous chapter, we interviewed the students after their work with an extensive piece of software, and we did a phenomenographic analysis of the transcribed interviews, which lead to a categorisation of the ascribed meanings of interfaces, identified in the text.

The result – the outcome space – is a structure that shows the qualitatively distinct ways in which the students described their experiences of interfaces. There has been much consideration about the categories' names, their character, and their internal relations such as hierarchic inclusiveness. This section outlines the categories with summary descriptions in textual and tabular form (see Table 1). The following sections give a more exhaustive description of the categories and their relations.

Table 1. The descriptions of the concept Interface – the outcome space

Category	**How the concept Interface is described** (its meaning, and purpose)
1 To-do list	The Interface is a text, in form of a to-do list, that tells the programmer what to do; what operations he or she should write. It is an uncompleted program, skeleton code, or a template, to start from when a new class should be written.
2 Content declaration, specification of operations	The Interface is certainly defined by a text; however, the text constitutes an abstract "thing" that can be bound to a class by referring to the interface's name. The class is thereby obliged to have implementations for all the operations specified by the interface. In this way, the interface becomes a forcing contract. The programmer must implement the interface, and the interface has a name.
3 Data type and reference	The interface is a data type for reference variables and thereby indirectly for objects. The interface, defined by text, has a name for the data type it represents. The data type can be used to create variables that can handle those objects that fit the content declaration. This is an expression of a meaningful relation between interface, class and object.
4 Open connection	The interface is an open connection to new and unknown objects. The purpose of interface type handles is that they represent an open connection towards arbitrary objects that implement the same interface. Hence, the same handle can connect to several objects, defined by different classes. According to the descriptions, it is possible to replace object and use different object types without having to change the rest of the software. Using interfaces allow objects to communicate with each other even though they are "strangers".

The analysis of the data has divided the conceptions of the Java interface, as expressed in the interviews, into four qualitatively separated categories. The least advanced category, "*interface as a to-do list*," describes a way of experiencing where the interface is a text that programmers can copy and start out from, when they are going to write code of their own. Partly, the interface is described as supports for the memory, and partly, as a framework to further continue to build code on.

Compared to the first category, the second category, "*interface as a content declaration*," expresses a more advanced and abstract understanding of the interface concept. The interface is described as a contract, between actors, that defines what the code must comply with, regarding its form and partially its contents. In this view it is understood that a class and its objects can have a relation to an interface in the sense that the class, that through declaration syntax commits itself to implement an interface, guarantees that the specified operations always are accessible in its objects. A metaphor for this could be an article (the class) in the supermarket; the third pie soup can on fourth shelf for instance (the object), having a label with a printed declaration of ingredients (the interface).

The third category, "*interface as a data type*," is characterized by the experience of the interface as a named data type that can be used to declare reference variables; and, the reference variables are used as handles to objects. The interface is described as a specific data type that, as for classes and primitive types such as integers and floats, is intended for a particular type of data. The declared variables of the interface type can handle precisely those objects which fulfils the content declaration, i.e., the instances of classes that is declared as implementations of the interface.

The fourth category that expresses the most advanced understanding, "*interface as an open connection*", the interface is described as something that lets objects communicate when the program is executed, in spite of the fact that they are unfamiliar to each other. Reference variables of interface type are described as connectors to arbitrary objects, if only they fulfil the stipulated "contract"; and this is expressed as polymorphism. In this manner, new objects can be inserted into the program that handles them "as is", without any changes in the existing source code; the program does not even have to be restarted. Hence, this facilitates for a convenient maintenance and further development of the software. Thanks to the separation of specification and implementation, the new objects can be introduced – even in run-time. The category represents a comprehensive view that includes all the the other categories' perspectives. In addition, the fourth category considers what is going on in the interface connections between the objects in run-time, and the positive consequences they have for the functionality and system maintenance.

The following sections elaborate and motivate the categories of description further, and they show how the categories relate to each other. I the discussion (Chapter 10) we return the categories and discuss other aspects as for example what was in focus when the various meanings appeared.

5.1 Interface is described as a to-do list

The first category ascribes the interface a meaning as a textual tool for programmers themselves to use as a to-do list and as a foundation to write code upon. The interface is a text file that helps the programmers to achieve their goals faster in a programming situation because it is a skeleton, scaffolding, or a framework that is the beginning of the program. It is something to start out from and continue when they write the code for their Java class.

In practise, you copy the textual contents of the interface, the text file, into another text file that is supposed to contain the complete source code, i.e., the class you are going to create. By doing it in this way, the programmer avoids to type in a number of necessary code lines, which makes it easier, and will in addition get them into his or her program without any misspellings. This pragmatic way of seeing interfaces – that they in practise are skeleton codes to copy and paste from – appears in some of the interviews. The following excerpt from the interview with Cia exemplifies that view:

> **Cia**: […] "So, then I copied the interface, and I removed what was only the interface, so I only had left what one must have in the class. And then I built from that."

Alf was asked to describe how he worked with programming assignments in the courses he had studied. In his answer, he described text editors, how he compiles his classes, how files are put in various directories, and then he starts talking about interfaces:

> **Alf**: […] "Now there are those half-finished programs there, which you continue to implement. Eh, what's the name? Interfaces and you know, to write methods and such. It is kind of, only to implement them and trial and error. Compile loads of times before one sees that it works."

One of the meanings that Alf described was that the interfaces are half-completed programs that one should use to write complete code. There was a vaguely outlined understanding that the meaning of interfaces was to know what one should write, and he accentuated this understanding later in the interview:

> **Int**: "What is the purpose of having interfaces?"
> **Alf**: "Well, but it is terrific to have interfaces, isn't it? I think so; it is only because when you start, you know, you think a little about what should I have? I shall have a client. What should it be able to do? And you write what

it should be able to do, and then it becomes very good. It is only to implement all of that and not forget anything, and then you can add more if you want and you avoid writing a lot… avoid writing a lot? Or once you have thought something through, you don't have to think a lot about what you should have."
Int: "So, interface is kind of a list of what to…"
Alf: "Yes, kind of, something to… prepare for something else."

Here, Alf expresses the conception that interface is something that you can use in a planning stage, in order to remember what has been worked out later. The meaning "to-do list" gets a deeper meaning, as something that carries ideas, about how someone has thought out how things should be done in the future, to somebody who is going to do it. Later in the interview, interfaces were discussed again:

Int: "In 'algorithms and data structures', how far have you reached in that course?"
Alf: "Eh, until, let's see, eh, what have we…? We have done this about linked lists, and now we are doing a linked list again, but not the one that we should do on our own… Gosh what is it? We are implementing a queue for, well, anything… for queues, for heaps."
Int: "Do you use interfaces then, somewhere?"
Alf: "Yes, everywhere, all the time! I really think it is terrific, I do, but it's just that there are so many of them."

Obviously, Alf thinks that the teachers in the course Algorithms and Data structures have used interfaces to communicate, to the students, which operations the various objects (abstract data types) must have. In this case, the teachers have provided the students with interfaces (to-do lists) which have made an impression on the student's understanding of interfaces.

The fact that the teachers' purpose probably was something quite different is not apparent in this interview. On the contrary, Alf expresses an understanding that the program does not actually *use* the interfaces. When Alf was asked to describe the software, he suddenly recalled that there were interfaces there:

Alf: "… but… I thought there was something else, those interfaces, the ones that you don't have at all."

The interpretation of this way to see interfaces is that they are understood as something that you do not use in the software; they are only used when you do the programming, merely as a tool that helps you to know what to write. Bea explains interfaces in a similar way:

Int: "I asked you during…, when you were working, what an interface is. Can you repeat that, can you elaborate on that?"

> **Bea**: “Well, you know, a trunk or something, or what should I say, that contains the method names and what they should return and the values they should receive. So that you later should not miss some method, or something to make sure that all are there. Well, you make an interface then, before you start with the actual implementation in order to, eh, well, get a structure of it all, to make it easier to, if you are many that work, or something to make it possible for all to work towards the same, well, eh, I don’t know really…”

Bea says that the interface contains an enumeration of method names (operations), what input these operations should receive and what output they must return. The enumeration is there in order to make you not forget any of the operations, and to enable you to communicate this to others; thus, more people could work with “the same”. In Bea’s description, we can se a nuanced understanding and a slight shift towards the way to experience interfaces as described by the second category of description.

5.2 Interface is described as a declaration of contents

The second description category of interface is constituted by expressions that take a different perspective, opening a new dimension, and at the same time, it includes the meaning of the first category. The second category describes interfaces as something more than merely the concrete text, or to-do list, that tells you what operations you must remember to write. In addition, the interface is understood as a content declaration that can be applied to classes[12], that is to say, a formal specification for how something should be constituted; something that offers and guarantees a certain protocol of functionality.

The interface is not only described as a text, it is also thought of and described as an abstraction; it has a name, and it symbolizes something. Hence, it is ascribed a meaning that reaches further than the textual properties. The experiences that form this category can be summarized by saying that the interface is a forcing contract that the programmer agrees to comply with by implementing the interface in a class.

When it is time to transform the written source code into executable code for the machine, the complier will verify whether the class conforms to the contract, or not, and if the compiler does not approve, it will not generate any code. Hence, the interface is not only a textual to-do list, it is a specification of how new code should be constituted; which operations that must be defined by the new code. And as Fia describes below, the programmer chooses to accept the interface, the contract, when he or she types a special

[12] This takes place in the class definition where the content declaration (interface), I, is tied to the class, X, through opening the class definition with the text statement:

```
class X implements I
```

keyword in the class definition, which consolidates the relation between the interface and the class in its declaration.

> **Int**: “The interface is used…, you said that one should implement an interface. Does that suggest that one should write a class when one implements?”
> **Fia**: “Yees…, I think it does.”
> **Int**: “And then, can you see it somewhere that you have…, that I choose to implement an interface?”
> **Fia**: “Yes, ‘implements’, very logical!”

The programmer binds a class to an interface by explicitly typing in the name of the class, the keyword “implements,” and the name of the interface in the first text line of the class definition, and thereby the programmer is committed to fulfil the agreement; that certain named operations are available in the produced code. The declaration is forcing since the compiler will not accept the class unless it implements all of the specified operations. The interface hereby represents an abstraction in the form of a contract that is defined by the interface, is fulfilled by a class, and is finally to be verified by a third party, the compiler. Eva has something to say about this:

> **Int:** “So, in this case for instance, was there something that forced you to write certain methods?”
> **Eva:** “Mmm…, yes, the interfaces.”
> **Int:** “Can you tell me about that?”
> **Eva:** “Well, I did not look so much at the interface itself, I looked in the other plugins instead. And I used, as in…, changed these implementations of the interface.”

Or as Cia puts it:

> **Cia**: “An interface is kind of something that tells you what you should use or have to use to make it work.”

In the following quote, Cia expresses her experience of an interface as a “content declaration” that in a summary form tells a programmer what can be done with a class that implements that particular interface:

> **Int**: “Do you think that your own understanding of what an interface is in any way made it easier for you to get on the track to solve the problem?”
> **Cia**: “Yes, because then I could find out how I could use it to communicate with the client, or work with the client. Otherwise, one would have to go through the client code, and see, ‘well I need that one, and that one’. That would take more time, but it would work, you know.”

This category of description includes the first category since the content declaration certainly is described in a text file, and the content declaration could be regarded as a to-do list as well. In the first category, the name of the interface is not relevant; however, the text file's name has some importance. In the second category, the interface's name is significant because the programmer chooses to implement a specific named interface. The notion of implementation is in this context, a relation between the interface and the class, which precisely means to fulfil the contract. Whenever the contract is complied with, we can always expect that the specified operations are accessible in the objects of the implementing classes. The following excerpt shows that Leo understands the interface as expressing something that the other parts of the program are expecting. To make it work, he must implement the interface according to the contract.

> **Int**: "And the Java Interface, do you get the hang of that? Could you for instance explain this 'PluginPanel', what's the big idea of that interface, in your opinion?"
> **Leo**: "Well, I guess it is because…, other ways it could go wrong when you don't have the proper functions and all that. I rather must, kind of, bring in what the program expects to come."
> **Int**: "So, if you are going to write a new of these tab panes, what would you do with this interface then?"
> **Leo**: "Then I would implement it and write code for the functions."

5.3 Interface is described as a data type

The third category describes interfaces in terms of references, reference variables, and data types. Unlike the first two categories, this category takes the point of view from the client side, that is, the part of a code that uses another code or components ready to use. Here is a short introduction to the background. If you plan to use an object in your program, you must choose which individual object you want to handle and its type.

This works since every object has a unique reference (address or handle), and these references can be stored in reference variables. These variables can refer to various individual objects, one at a time; however, in Java, the condition is that the types of the object and the variable are compatible. A class represents a type and in order to handle its instances (objects), you have to use a reference variable of the same or a compatible type. This implies that the code, that wants to communicate with an object of type X (class X), must use a compatible handle. The obvious type for the handle would be the actual class X, but a compatible type is sufficient. What is a compatible type then?

In this category, the students describe how a class that implements an interface is compatible with the interface. The transcriptions revealed an awareness of the fact that an interface is a data type, and that it, just like the class type, can be used to declare reference variables. In the source code of the system, there is an interface called *Registration,* and Fia recalls it when she expresses her understanding of interfaces used to create variables:

> **Fia**: “Well, yes, you can use that ‘Registration’ as a reference!”

In the interviews, there are descriptions of how these reference variables – declared by an interface – can be used to handle (refer to or point at) those objects which classes implement the same interface that declared the variable’s type. Bea describes the interface as a reference variable’s type, and accordingly, how that variable can point to compatible objects:

> **Int**: “So, how can it be that it still can display your program, or your class as one of these tab panes?”
> **Bea**: “Oh, well, but yes…, yes, because it knows what a ‘PluginPanel’ is, I suppose, or what the interface is. It can have a pointer of the type ‘PluginPanel’ and then it can point to all that implement it, ‘PluginPanel’, and therefore it can put it up…, although it does not know what a ‘StudentPluginPanel’ is, it can put it up anyway.”

If you use this type of variable, the only thing you can do with the object it refers to, is to call those operations that the interface specify – even if the object itself offers many more operations. This implies that we can use the unchanged client code generically, to handle various objects that stem from different classes that represent disparate implementations of the same interface. Bea expresses this in a less complicated way by saying that the variable just does not care about how the object implements the interface:

> **Int**: “What can it do with that then?”
> **Bea**: “Well, it can do all of the methods that are defined in ’PluginPanel’.”
> **Int**: “So, you did not include that in your description of interfaces, because it is another aspect of interfaces, that there can be different implementations behind the same interface, or?”
> **Bea**: “Yes, but that is…, because it…, well, it does not care about how they are implemented, only the…”

In this category, there is a connection between the meaning of what the word “interface” represents and what the concept represents in Java. The interface seems to be a border or an intersection between the client code that uses objects, and the objects that comply with the interface. The objects can behave as they please, defined by the code in their class, as long as they provide implementations for the specified operations.

This category includes the second category because the descriptions include names for the interfaces, and in order to use it as a type and as a compatible type, someone must have implemented the class and have used a compiler to verify the content declaration contract. Another difference compared with the first two categories is that the interface’s name constitutes more than a just an identifier. In this category, it has become a type. Ken expresses that he in fact regards the interface as a data type and that an implementing class is compatible with that type:

> **Ken**: "And, once I had realized it, that I had to create a completely new class, of this specific type, plugin, well 'PluginPanel'…"

Later, Ken describes how he looked for an existing implementation, available to start out from when solving the problem. He mentions a plugin that an interface described (formally), but here he also connects the type to an implementation, that is, a class:

> **Ken**: "… and then I started to look around in the other panels and I noticed that this panel, 'CourseInstances', that, that it probably was pretty well suited for this kind of plugin. But of course, that you had to modify it, and that was what I was trying to do now."

When Eva describes how the system uses the so-called plugin modules, she points out how the client side only depends on the interface:

> **Int**: "So, in other words, you mean that one who uses these… the client program, what does it have to know about plugin?"
> **Eva**: "It must have the interface."
> **Int**: "Only?"
> **Eva**: "Yes."

5.4 Interface is described as an open connection

The fourth category reflects descriptions of the interface concept that have a more profound meaning than in the previous categories. In addition to the descriptions of interfaces as types for variables, the descriptions emphasize that these reference variables can refer to arbitrary, concrete objects, if their classes implement the interface.

The criterion is that the implementing methods have exactly the same signatures as the interface prescribes, while the program code inside the methods, their "inner life", is not part of the agreement. This implies a degree of freedom for the programmer to decide how to define the functions, and the programmer can even provide several alternating versions of implementations.

Thus, an interface can have *any* implementation; one or more programmers can write different versions, and they are *exchangeable*. The understanding described in the third category is a prerequisite for the understanding in this category; a common variable type that can refer to all of the varying implementations is required. Now we come back to Fia, who may unfold the thoughts that she indicated earlier in the description of Category 3:

> **Int**: “Speaking of interfaces, how would you like to describe what an interface is and what use one can get from it?”
> **Fia**: “Well, yes, what should I say? An interface is something that one can implement and use for several different implementations, one can say. Well, what should one say? It is a bit like a template one could say. Of course, you cannot use a pure interface to run something; you always have to implement an interface. That is to get hold of the functions that you want in the program. They are in the interface, you know. And then you have to redefine them in the application that you are doing.”
> **Int**: “Why should you have an interface? Why can’t you write the application directly, so to speak, without the interface?”
> **Fia**: “Well, yes, but then it isn’t certain that the applications support each other, that they become alike. When you make a program from scratch, you have decided what it should look like and what should be included in it, and then it is better to write an interface for that and you don’t have to make the implementation at the same time.”
> […]
> **Fia**: “Well, yes, you can use that ‘Registration’ as a reference!”
> **Int**: “Yes, that’s right, so, that’s one way to use it!”
> **Fia**: “Yes, sure, that is what you can do, and you have to point at an object.”
> **Int**: “Yes, exactly, the one that implements it?”
> **Fia**: “That implements the particular interface.”
> **Int**: “If you think about the reference variable, which are the concrete objects it can point to?”
> **Fia**: “It can point at all those objects that implement the interface.”
> **Int**: “Ok, so it doesn’t matter which the implementing class is, if only it implements that interface?”
> **Fia**: “Yes.”
> **Int**: “Do you know of any fancy word?”
> **Fia**: “Polymorphism.”
> **Int**: “So, polymorphism, what does that mean to you?”
> **Fia**: “It means that you can call a function without knowing what it really does. You only know that it is there, and then what you call is executed. That’s polymorphism to me!”
> **Int**: “Can you get any practical advantages from that, so what interpretations can you make from that?”
> **Fia**: “Well, you can do… The advantage is that you can add more classes afterwards without any need to remake the program and compile the rest of the program. You only have to add what you need, if you have to expand the program for instance.”

The benefits of having this property in a software is described as that it is possible to exchange objects with new modified ones later, and that there is no need to re-design or re-compile the existing program. This was fundamental for the application that the students worked with, since it used plugin modules and enabled an independency at the client side. One effect of this was that the users on the client side did not have to re-install the software if the developers upgraded the system. In fact, they did not even have to restart it. In the following quote, Eva says something that tells us about her insights of plugins:

> **Int**: "About this plugin thing, what does it mean to you?"
> **Eva**: "That you extend the program…, enlarging it with more functions without actually making changes in old functions."

A similar way to experience interfaces is expressed more elaborately by Joe, who explicitly connects the concepts "interface" and "plugin" to each other. Moreover, he is capable of making clear how he values things, and what his opinions are concerning interfaces:

> **Int**: "… interface, what is that?"
> **Joe**: "Interface, that is, ha, ha…, by having an interface here, the client doesn't have to know really, what, what it gets from the server. Because all these plugins implement the same interface, then … then there are a certain number of methods defined that always are included in the interface, that you just simply can call. But you want to know…, should I explain what an interface is…? It is, well, I really know what it is, kind of."
> **Int**: "Yes, but I thought that it was a good explanation of what an interface is. So, the client doesn't know exactly what the objects are then – is that what you mean?"
> **Joe**: "Yes, exactly."
> **Int**: "Behind the interface, but it knows that it always can call these?"
> **Joe**: "Yes."
> **Int**: "What's the big deal about that then, or what is it that could be good?"
> **Joe**: "In this case, it is super smart, it is really a proof of that it is smart because, ehh, i is only to create a new plugin and implement the interface, and it will work, painlessly!"
> **Int**: "Hm, and if you for instance imagine that…, you said that this software was intended for teachers, and 500 teachers are sitting here at the university, and there is this server running, and you want to add a new function. What do you have to do then, really?"
> **Joe**: "Well, it is only at one place in the server where you need to build this new plugin class, and you have to go to the database and specify who should have access to it as well, I guess. And then it will happen automatically, that all 500 clients, or all who should have access to it, will get it. You don't have to update each client."

Joe seems to think of the system from a run-time perspective when he answers the questions, and he can account for what happens to the object when the computer executes the program. From this point of view, he can describe the advantages of using interfaces when he explains how the server sends objects to the clients, and that the client software can use the objects without knowing anything more than the interfaces they implement. He realizes that this implies, for him as a programmer, that he can modify the server system without any need for re-installations of the software at the user’s computers, and that this works thanks to polymorphism through interfaces.

A characteristic insight that distinguishes Category 4 from Category 3, is that there is no need to re-compile the client code when reference variables of interface type refer to newly implemented objects, provided that they are compatible. Static bindings between types require re-compilation after changes in the code because it is already determined at compile time, which explicit object types that the client refers and calls.

However, the descriptions in Category 4 reveal an understanding of *dynamic binding*, a crucial property of Java. When reference variables in Java refer to objects, as in the case for interface variables, run-time mechanisms determine which methods to bind to method calls. The interface type contains no information about where an object’s methods “are”. The advantage of this is consequently that one part of a program can create objects of which the explicit type is unknown to another part of the program. Nevertheless, the unaware part can still handle the objects, and can make use of the operations that the interface specifies.

Thus, without having to re-compile or restart it, a program in operation can handle new object types. It is thereby possible to design a system that has a weak dependency between different parts of the software. The system that the students met utilized this fundamental idea when the server distributed new types of objects to the clients.

Moreover, the students could see yet another aspect, namely, that the system was designed for being distributed on several physical machines. The server part was located on a dedicated server machine, and the client part was supposed to be distributed to several user machines. The server machine would hand out a number of objects to the client machines that would use them without knowing the concrete types.

The experiences of interfaces in this category are that it is a means to achieve the behaviour described above. The descriptions explain that it is possible to create a new class that implements a certain interface, and that the server can distribute the corresponding objects to the clients in the system, where they will appear to the users. In fact, this is a paraphrase of the students’ task in the role-play, but there was no explicit description of how this works or how they could achieve it.

In particular, there was nothing in the written text about the mechanisms of Java interfaces. The descriptions concern courses of events and mechanisms in the run-time dimension, and they reveal an overall view on the interfaces’ range of application with a clear connection to the software system that the students worked with before the interview.

There is an obvious connection between the concepts interface, class, object, and plugin in this category. This insight is the first that comes up to discussion in the interview with Cia, and this indicates that some of the participants have learned something from the experiment, and that they are aware of this learning:

> **Int**: “You have been sitting here for two hours and three minutes, and have been working so hard that the sweat was dripping. And we are having a small interview afterwards to find out how you experienced this.”
> **Cia**: “To see if one has learnt something.”
> **Int**: “Yes, we can start with that. Have you learnt something, from this?”
> **Cia**: “Yes, well, it is this about plugins, I guess, how you can use these in a good way.”
> **Int**: “Can you describe what it is?”
> **Cia**: “It is this kind of small programs, you know, that you insert. So that you don’t have to recompile the main program. Rather, it is just this little program. That’s a good thing to know. I hadn’t tried that before.”
> **Int**: “Did you see any description of that in the documentation?”
> **Cia**: “Well, there was one of these plugin classes…, interface that was not implemented.”
> **Int**: “Can you see any use of not have to recompile the client program?”
> **Cia**: “Yes, if you are adding certain stuff, sort of, if you have made a save function that you didn’t have before. Then it is just to send the plugin instead.”

In the following excerpt, Eva connects the interface concept to something forcing, and she comes into plugins when she describes interface, and then she continues with a description of how a new plugin can be made from an existing one. Finally, she returns to the purpose of using interfaces and plugins in the system; that the plugins can be inserted transparently as additional programs, or as a replacement for other plugins:

> **Int**: “So, in this case, for instance, there was something that forced you to write certain methods?”
> **Eva**: “Mmm, the interfaces, yes.”
> **Int**: “Can you tell me about that?”
> **Eva**: “Well, I didn’t look so close at the interface it self, I looked at the other plugins instead. And used as in…, changed these implementations of the interface.”
> **Int**: “Can you explain how you view this, about what an interface is, and for what it can be used?”
> **Eva**: “An interface is, you know, that you specify which methods a, well a class should have, an object, or what should I say? Well, yes, it can be used very favourable when you write plugins, you know.”
> **Int**: “Because…?”
> **Eva**: “Because then the plugins will get the functionality that they are supposed to have. It can do these things. So that, the program itself, that runs the plugins can send it…, the same thing to all plugins and get the same things in return as well.”

Thanks to their property as constituting a common data type that programmers can handle in a polymorphic way, the use of interfaces enables the interchangeability of objects – the plugins. In addition, a consequence is that the specific implementations of the plugins are irrelevant to the structure:

> **Eva**: "An advantage of interfaces…, well, if you should not use it, then the…, well, you know, it…, the advantage is also to treat all of the plugins as one plugin. You don't have to treat them as their particular implementations."

In summary, we conclude that the fourth category expresses a deeper understanding of the interface concept, where the understanding of interface relates to something that lies far beyond the circumstances of writing code for a certain class (the first and second categories).

Thanks to the requirements imposed on all classes that implement a specific interface (the second category), it is possible to handle all of their descending objects with variables of interface type (the third category). Hence, it is possible to achieve a structural property of a software system that, within certain limits, enables and allows dynamic changes.

This facilitates conditions for a system that is capable of, and prepared for, development and maintenance. The fourth category expresses an ability to discern different aspects of interfaces simultaneously, and this way to understand synthesizes all of the approaches described by the other categories. This argument makes it seem reasonable to claim that Category 4 includes all of the previous categories.

6 Descriptions of the concept plugin

This chapter presents the students' experiences of plugins as they appeared in the interviews. As accounted for in the previous chapter concerning interfaces, the result of the analysis is an outcome space that consists of description categories.

From the point of view of an educational context, the notion "plugin" differs from the concept "interface" in the sense that the former represents informal knowledge, while the latter is part of the formal studies. It is not to expect that the Computer Science students should have a profound understanding for the "plugin" concept, because it is not (a salient) part of the terminology used in course literature, nor is it part of the undergraduate programming courses as an explicit term or technique.

It is possible that the design courses, on the more advanced level, use the concept informally, but then only in classroom discussions, and not in lecture notes. Hence, it is possible that the concept has no meaning at all for some students. Certainly, a quick search on the Internet resulted in many hits on "plugin", but the connection to academic education was very modest. Instead, we find the concept in various concrete contexts where it is possible to expand existing software with additional modules or extensions that often have the common name "plugin". It is therefore most improbable that the students' conceptions of plugins, as expressed in the interviews, would originate from teaching. Rather, the students established it during the experiment or it was already part of their previous experiences, or a combination of the two.

Evidently, for most of the students who took part in the experiment, the concept plugin was very vague or completely unknown before the they started to work with the task, and for some of them, even after the two hours of work, the term was still very hard to define. This was the case for Dan:

> **Int**: "For example, there was a concept there – plugin. Did you get a feeling for what that was?"
> **Dan:** "No, I don't think so. I read about it, and read again, and there was texts and stuff. But I never really understood what the particular word plugin meant."
> **Int**: "Have you encountered that word somewhere else?"
> **Dan:** "Plugin, I don't know. No, I cannot say so straight off, no, not that I know of. I might have come across it, but not as I recall."
> **Int**: "Can you make a guess about what it could be, what it sounds to be?"

Dan: “A plug, a plug in, well, I don’t know. You have a *plug* and then *in*, I don’t know.”

However, other students describe how they already had a good understanding of the concept as something they brought with them from the world outside the university, connected to their experiences of computers and programming. Here Eva and Joe confirm that they had a preconception about plugins before they participated in the experiment:

Int: “Then, did you have a full understanding of what a plugin was, you know, what was the meaning of a plugin?”
Eva: “Yes, because I already knew that from before.”

Int: “And the concept plugin itself, is that something that you had experience of, that ‘aha, a plugin’, did you know what it was?”
Joe: “Well, kind of, it usually…, yes, I did.”
Int: “I don’t know, maybe I have misused the word plugin in this case? This is what I have called it.”
Joe: “Well, yes, but I felt it was logical that it was called plugin. I thought so.”

As we could see already in Chapter 5, the plugin concept appears in different contexts in the interview excerpts. The analysis of interview data resulted in two qualitatively distinct description categories, accounted for in Table 2. Because of its absence in the education, it happened that students could not describe the concept plugin – as in Dan’s statement above – in other terms than the literal meaning of the word. Descriptions that we classified as guesses, or as having tautological features, did not form a category of their own, because they did not actually describe the phenomenon. Nevertheless, it was a very interesting result; that there are individuals who did not succeed to obtain any “proper” understanding of the concept, despite their hard work during the experiment.

Table 2. The descriptions of the plugin concept – the outcome space

Category	**How the concept plugin was described**
1 Small program	A plugin is described as a small program that contains what should be done; it is the code for a “tab sheet” – a part of a graphical user interface that is responsible some operations. The concept is described as a tangible implementation.
2 Part of extensible structure – a concept	A plugin is described as a small program – a module – that can be inserted or removed to a program. It is part of a conceptual model that enables the system behave in an adaptive and dynamic manner. The remaining parts of the system do not have to be re-compiled, or restarted. The notion of plugin is described on a conceptual, abstract, level.

6.1 Plugin is described as a small program

The characteristic features of this category are that it describes the concept plugin as a specific type of classes that are part of the graphical user interface. They all work as a “tab sheet” that displays an input form, in which the user can type in the intended information, and in addition, the “tab sheet” can display output data. This was exactly what was going to be developed by the students – an additional “tab sheet” that could handle students’ registrations on courses. In Alf’s description, it is evident that the plugin classes were something different compared to the other classes, and that their task was to handle user input data and pass it on to the system:

> **Int**: “All right then, if we look at the actual program, well, you have noticed that not all code is, kind of, in the same class. There is not only one class, but there are plenty of them, you know. Do you have an idea of which classes that deals with what? How the structure of the classes looks like, kind of?”
> **Alf**: “Eh, what classes, OK, these plugin classes, they receive everything and later calls others to register, but… thought there was something else, those interfaces, those that you don’t have at all. And then it was this thing, that there was no kind of, that there was no sort of ‘putting up class’, so that you know, kind of, that here’s a central class that puts it all together. But that would not be smooth[13], would it?”

Alf also describes his experience of how there was no central class that “puts it all together” – that, if so, it would be easier to understand, but on the other hand, it would not be a smooth and easy solution. It seems like Alf conceives the system as constituted by parts that call each other, however, when he uses the term plugin during the interview, he do not associate it with their origin and how they come to appear in the graphical user interface:

> **Int**: “From where do these plugins come, how does it work?”
> **Alf**: “What do you mean by that?”

In Ken’s description, we can clearly see how he feels uncertain when it comes to plugins. He probably did not bring with him any preconceptions about the term to the experiment; rather, he made up his understanding during the work. At first, he thought that everything in the system was already well worked-out and complete, and that the only thing he had to do was to pick out which of the existing classes he should use. Later, he understood that he had to program a new plugin class of his own. Then, to get started, he “borrowed” code from a similar class. However, in his descriptions, the concept plugin itself appears rather vague.

[13] This is a translation from the Swedish word “smidigt” that means something like “come in handy”, convenient, smart, flexible, and smooth.

Int: "… could you explain what the big idea is, what really is the meaning of a plugin?"
Ken: "The big idea of a plugin?"
Int: "What do you consider special about the concept plugin? Do you get any associations, or a sense of what it could be?"
Ken: "In this case, everything seemed to be complete. The only thing that was not finished was that they did not have this graphical interface to do this particular registration. Therefore, a plugin, I guess, is a kind of, this is a long shot again, you know, that you try to get at functions that are not available. I mean, they are available, but not available to use."

Hal describes that it took him a while to understand how it all worked. He had to wait until he executed the program and he could actually see the "tab sheets" in the graphical user interface. Only then did he apprehend the meaning of the term plugin:

Int: "This concept 'plugin', did you understand what was meant by that, at once?"
Hal: "Well, yes, I guess it took some…, I mean, perhaps I did not understand it right away as I was reading it, rather it was when I started the implementation, when I saw the various tab sheets in from of me, the panels. Then I understood the purpose, and why it was in that way."

In our interpretation of this quote, the tactile input given by the trial run of the software allowed him to make the connection between the "tab sheets" and the plugin concept. The formal descriptions of the concepts in the documentation were too abstract to make this clear.

6.2 Plugin is described as part of a conceptual model

In addition to the previously discussed way to experience, the descriptions of plugins comprise the importance they have on the system's internal structure and functionality. Eva describes how it is possible to augment the system in a way that does not affect the parts that already exists. Of course, this affects the appearance of the system's "whole", but none of the "old parts". In principle, the plugins build up the entire system, and if they should all be removed, there would be nothing left:

Int: "About this plugin thing, what does it mean to you?"
Eva: "That you extend the program…, enlarging it with more functions without actually making changes in old functions."
Int: "What…, do you know what…, in this program, if you should remove all the plugins, what could this program do then?"
Eva: "Well, it's built upon plugins, so if you should start it without those plugins, there would be nothing. Because all the things there were plugins, you know."

> **Int**: “Hmm, I think the only things you could do is more or less logging in and logging out. Eh, if you imagine…, in relation to this system with plugins, eh, do you see an advantage of having it built up in this way, or, what advantages do you see then?”
> **Eva**: “I see one advantage, that…, as in this situation, that someone else could get familiar with how to write a plugin, and really wouldn’t have to bother about the rest of the code.”

Eva describes how a programmer, who is about to make a new plugin, do not have to bother about the rest of the system – it is sufficient to know how to write the plugin. Hence, the advantage of having a system that utilizes plugins is the independency between the system’s parts.

The descriptions that form the present category concerns how the system, in a formal sense, is independent of how the specific plugin behave, which implies that the system’s code is independent and thereby does not require any re-programming or re-compilation. Eva describes yet another advantage, namely that there is no need to shut down the server when new plugins are installed, which makes the system free from operational disturbances during an upgrade – a profound insight.

> **Int**: [*Describes a system in operation with many users*] “With this system, do you see any advantages there, have you thought about that?”
> **Eva**: “Yes, exactly, that you can activate new plugins without restarting the program, the main program. Otherwise, it could get a bit awkward if there were many users working with it, and then you are putting in a new plugin.”

Cia tells us how the plugins are loaded to the clients from the server (see page 68). In the following, she describes that the client is unaware of which specific plugins it receives; it only knows that they are objects of plugin type. If a programmer wants to add something new to the system, she or he only has to write and compile a new plugin, and install it in the system.

> **Int**: “Yes, we can start with that. Have you learnt something, from this?”
> **Cia**: “Yes, well, it is this about plugins, I guess, how you can use these in a good way.”
> **Int**: “Can you describe what it is?”
> **Cia**: “It is this kind of small programs, you know, that you insert. So that you don’t have to recompile the main program. Rather, it is just this little program. That’s a good thing to know about. I hadn’t tried that before.”
> …
> **Int**: “Can you see any use of not having to recompile the client program?”
> **Cia**: “Yes, if you are adding certain stuff, sort of, if you have made a save function that you didn’t have before. Then it is just to send the plugin instead.”
> …

Int: "So how can it come that…, if one of those 'UserAdminPlugin' comes from the server to the client, how does the client itself regard the plugin that is coming?"
Cia: "Well, it regards it as a quite normal program…, or one of these polymorph…"
Int: "Ok, so polymorphism…?"
Cia: "Well, you don't know if it is, kind of…, which plugin it is, rather 'he' only knows that it is a plugin that 'he' can use."

Git follows the same line of argument as Cia and Eva, when she describes how the development of new plugins can be separated from the system software, and that the new plugins can be introduced without system stops. She compares it with the alternative – which would force the customer to wait for two weeks during the system upgrade:

Int: "Is there anything special that you think you have learnt?"
Git: "Eh, well, I think this plugin part was interesting, just because I like it when it's… Well partly to separate the developing of it, then if you want to release this to some client that you have created it for, and then you only want to extend it with further functionality… That it should be easy and that you don't have to, kind of, that you don't get entangled, kind of. Well, that you don't have to take back the entire package, and then they would have to wait for two weeks while I'm making something new. That I can simply put in something extra, and then it works right away, kind of, without having them to shut down the application, kind of…"

Now, we let Leo give the concluding evidence of this category of description. He describes his way to see plugins, what he had learnt, and he tells the interviewer that he never had thought of the possibility of doing it in this way before. By saying that, he refers to the dynamic way to handle new objects (plugins) in the software. He describes how a system administrator can install a new plugin, by adding a file and type in its path to the administrator's "tab sheet". Apparently, when he says that the plugin design was "smart", he believes the described advantages are important and meaningful.

Int: "Perhaps you could describe what you have accomplished?"
Leo: "Well, it didn't become more than, an…, I haven't managed to get anything out from the database yet. But I have figured out how it works with plugins and all that. So, there is not so very much left to do really. If only you figure out how it works with the database and those things."
Int: "This thing about plugins, could you explain?"
Leo: "Well, it was that you…, well you could simply, you only had to compile a file, and then the administrator, I have it here…, then he could type in the path to the file, and then it appeared in the program, a new tab sheet."
…
Int: "Do you think you have learnt something today?"

> **Leo**: “I think it was very smart to use plugins in this way. I have never thought of that you can do it. You can add a file, and then it turns up in the program without recompiling it.”
> **Int**: “What advantages could one get from that?”
> **Leo**: “Well if you define and have this on results instead, you know. I don’t know exactly how it works, this with the server/client, but perhaps that was intended. Then you can simply update the server and then the clients get updated right away. And the clients don’t have to…, you don’t have to distribute the program again and they don’t have to reinstall what they should have, because the upgrade takes place automatically.”

The data analysis revealed several points in common between the description categories of the two concepts interface and plugin, and the students’ work with the system was the context that connects them. The relation between the most advanced categories, respectively, are obvious, and they are in a way two faces of the same coin.

7 Descriptions of the system

This chapter deals with how the students describe their experiences of the system they were working with. For most of the students, the programming task was a big challenge. The documentation of the administrative software system “StudAdmin” was brief, and the description of the task was even shorter, and the explicit description for how to do it was almost insufficient. When the student had managed to install the project on the computer, it was only to discover that the numerous source code files were scattered into several directories in a tree structure.

Considering a person that wants to get things in order, and understand what, where and how to get things done, it is reasonable to suppose that it is significant to realise the meaning of it all, the why. In order to get an opinion and see the meaning, it is probably important to create a picture, or model, for how the system is constructed. We wanted to find out in which ways the students considered and explained the system. We were interested in what they had managed to get out of the documentation, the source code and its comments, and the trial runs.

Naturally, there were many circumstances that influenced the students’ interpretations of the system, for example their previous experiences and knowledge about programming, design and information systems. While some of the students had experiences from the IT business, and some had been active as hobbyists, others had obtained their experiences from the university courses only. How did the students experience this IT system, after two hours of labour with reading its documentation, dealing with its software, and running it?

The starting-point for the analysis was the built-in complexity in the system’s design and technical construction, and it was mainly in this dimension of complexity we interpreted and compared different expressions of meaning.

The analysis of the interviews and the identified ways to describe the system resulted in three qualitatively different categories of description. Each category represents a particular quality in various expressions of meaning in the students’ descriptions and that way it groups a set of statements from the interviews.

The categories are logically associated to each other in a hierarchy, based on inclusivity, where each category on the “higher level,” includes or presumes the underlying ones. At the same time, they are distinct in the sense that each category opens a new dimension; a new way of seeing that is not present in the underlying categories. Table 3. shows the results of the analysis.

Table 3. The outcome space of the students descriptions of the software system that was the subject of their work during the experiment.

Category	**How the system was described**
1 What the system can do	The system is what it can do and what its purpose is; a unity that provides a number of services that someone outside can use, e.g., register students to courses. The descriptions have an operational association to the system.
2 How its collaborating parts do it	The system is a unity that provides services, and it is constituted by logically or physically separated parts, that collaborate and delegate tasks between them. In the descriptions, there are often associations between the system's parts and the corresponding source code. The descriptions have a structural association to the system.
3 How it can do new things	The system is a multi-user system with different levels of authorization. The design is "interesting" and "smart" because of the lightweight clients and the dynamic plugin loading. The descriptions associate technical solutions and strategies (framework) that go beyond the "user's view" or the functionality of the software itself. The way to structure its parts is important.

The following sections allow the reader to take part of the students' own voices. We elaborate on each description category by giving examples of the students' descriptions of the system, and our interpretations.

7.1 The system is described in terms of what it can do

During the interviews, the interviewer asked the students to describe the system they had worked with. One way to interpret the question and a corresponding way to describe the system was to focus on what the system could "do" or what one could "do" with the system.

This category of description synthesizes the various expressions of meaning which associates towards how the system appears to a user, and this especially involves what the user can do with it. The term "system" is not associated with inner structure and properties; rather "the outer" – what you see or what you do with the system, what discerns it from the surrounding world. This operational way of describing the system is a means to abstract the inner complexity of the system, into a "black box". To encircle and delimit the system from its surroundings by describing the general, the common and the obvious, seems to be a natural way to start a description (see Figure 1).

In the following quote, Bea describes the system by explaining its purpose and use:

Int: "How would you like to describe this system?"

Bea: “How do you mean, the actual…?”
Int: “Yes your conception of the system, what it is.”
Bea: “Eh, mm, well a system where you can register students and courses and connect relations between the students and the courses, which courses they take. What was I saying? And then this about instances, that there are different, well that the courses run at different occasions and concurrent…, at different speeds and such. Well…”

The students also speculate about where the system is supposed to be used. Alf describes an outer context where “the program” has its place and its tasks:

Int: “Let’s continue with this. Could you try to describe this system?”
Alf: “The system, the program then, how, what you are supposed to do, or?”
Int: “No, describe what kind of program it is, what it does and how it works.”
Alf: “A program, kind of, that you could use in a university. And add new students and set up which courses that exist and then type in which students are taking which courses. Hm, then it does no more.”

The fact that the term “program” is used to describe the complex structure of components that are parts of the system, emphasize the aspect “what the system does”. In our opinion, the term “program” has more of an operational character, rather than a structural. In addition, it indicates a view that refers to a unity, rather than to a integrated whole.

Teachers are supposed to use the system, Joe guesses. Using it, they could administrate their courses and keep track of the students who follow the courses. He also gives his opinions of the suitability of the system:

Int: “Could you try to give a description of the system in your own words, how it works and how it is built up…, in your opinion?”
Joe: “Yes, ok, it’s…, I suppose it is teachers who are going to use the system. You can set up your courses, eh, and create own instances of courses and register students to these courses. But you cannot put in grades or something. So, eh, but you can keep track of which students that are taking ones courses, and which courses you have. It would require some more information in order to be really good, and that. Anyway, I guess a teacher is the one who should use it, not the entire school. It is more for a teacher sort of, it feels like that. Otherwise, it would be very much.”

The interviewee Dan describes the system by referring to the kind of information that was to read in the documentation. Moreover, he describes what the program displayed. When he mentions “a third tab sheet”, he alludes to the various “tab sheets” in the graphical user interface.

Int: “Can you try to describe this system. The system that you have been working with and studied now; what is it all about?”
Dan: “It’s a database system, with students, courses, among other things, I guess. And I should be able to see which… I could see which students and

which courses, you know. And then there was something 'course', a third tab sheet that said 'course'… I don't remember what it said. Those three."

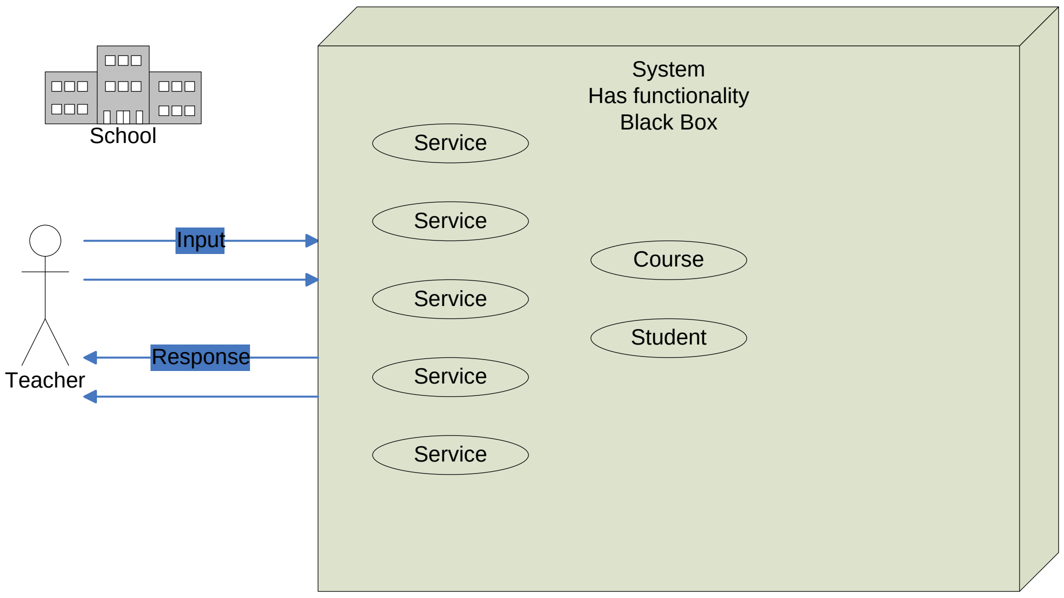


Figure 1. You can do things with the system, which in this sense "is what it does". Teachers can record the students who follow their courses, for instance.

In this situation in the interview, Dan seems to be focused on his own idea of what the system could or should do, but he does not pay attention to *how* it is done. The interviewer hints that there is a possibility to get different functionality depending of who you are as user of the system, but instead Dan focuses on what functions that should be useful to a student, such as the ability to see who the teacher at a specific course is:

Int: "It said something about that there were different user categories. One could log in as 'root' or as 'admin' or as 'user'. And you have logged in as 'user' now, and you can see that these buttons are greyed, so you can only look at…"
Dan: "Yes, exactly. But if I should go in here as a student, then it would be interesting to see which courses…, as I was saying. And in the same time, I would like to see which teachers that are giving the courses. That is actually very important to me, if I want to apply for a course. Because I think, it is so different with different teachers that I've had, and therefore it is important, for my part, to see which teachers I have."

The documentation at hand during the experiment described the system's purpose and possibilities, fairly well. All of the participants in the study read the documents that described the system, and should have made a clear picture of what a user of the system could do. The descriptions that belong to this category do therefore not necessarily reflect a deep understanding of the system, obtained from activities concerning the system's design and source code. They could rather be a depiction of the general description in the technical documentation. Nevertheless, it is possible to describe and conceive the system in this way, and still have different, perhaps more advanced, ways to understand it.

For the students, it seemed to be very important to get a basic understanding of how the system acted towards the surrounding world before they could continue. They needed something to act from – a context and an outline of the system. In spite of the fact that the documentation describes the system, the majority tried to run the software in order to understand how it behaved "for real". Perhaps the urge for an operational understanding is a first fundamental step towards an understanding at a different level.

7.2 The system is described as integrated parts

In the second category, the students describe the system partly from the perspective that it performs services to users, as in the first category; however, what the descriptions emphasize is the system's internal structure, and therefore the focus changes from the surrounding world into the interior. The system is described as a client/server system consisting of relatively independent parts: a client, a server, and a database, which are logically or physically separated from each other (see Figure 2). There seems to be a clear model that shapes the parts and ascribes them with meaning. The parts are separate, but still they can communicate and thereby share and delegate the workload. In the following, Fia describes two aspects of the system in one sentence:

> **Int**: "How would you like to describe this system, what is your image of it?"
> **Fia**: "I understand it as a client/server system that should, well, take care of administrative tasks concerning students and courses."

The first aspect is what the system can do, which belongs to the first category. However, the second aspect introduces something new; the system is descried as a client/server system, in other words, she describes the internal organization of the system, and which parts it consists of. Later, the interviewer asked Fia to tell more about how the system was constructed, and then she came back to the client and server parts again:

> **Int**: "Do you have any idea about how it is built up?"
> **Fia**: "That, well anyway, it was built up by a client part and a server part that will…, then, with the client part you should be able to call the parts on the server, so that they will be independent of each other, and then you have the database that is connected too. I guess the intention is that the small part is the client part, and that the big one is the server part."

An interesting contradiction takes place when Fia describes how the parts use each other when the client calls upon the server, and in the same sentence, she says that the parts should be independent of each other. A reasonable interpretation of what she says is that the one part should not have to concern *what* the other part will *do* when it calls it. In other words, that the client can rely on the specification of the operation it calls on the server. For example, this would enable developers to work on different parts of the system independently, without consequences for other parts of the source code.

This has implications for how to compile the system's source code; it should be easy to compile the files, belonging to a certain part of the system, separately. This is actually the case for the designer has organized the source code for this system. The directory structure separates the client code in one sub tree, and the server in a parallel tree.

Fia also discerns the database as a part of the system. She describes it as a separate part that connects to the rest of the system. In addition, Fia suggests that the parts should have different sizes, which implies that the client mostly should delegate tasks to the server's more extensive software instead of doing them itself.

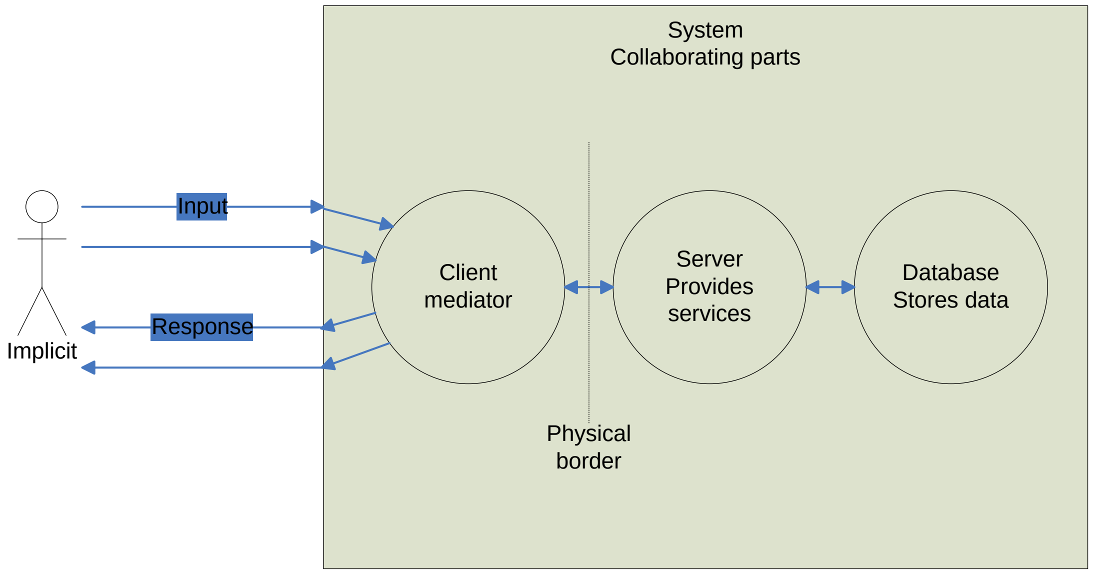


Figure 2. The system is described as a structure consisting of collaborating parts.

Later, when Fia tells what she has learnt, she returns to the system's division in parts and points out how they "work towards each other". We can also see how Fia describes her preconception that the server and the client can be separated in the room and be connected through a network. Hence, the system is something that is not restricted to a certain place; on the contrary, it can be distributed over long distances:

> **Int**: ”Have you learnt something today?”
> **Fia**: "Yes, I think so. First of all, I have learnt how to do this kind of experiments. Then I think I have come to somewhat of a deeper insight into how

client/server systems work, actually. It was just more that you… previously, you knew that the client was at one place, and the server was at another place, but now I think I have more understanding of how they work with one another. Therefore, it was good for me, this exercise. I think.”

Bea describes her learning from working with the task similar to Fia’s way of putting it. In addition, she thinks the way to divide the system is “smart”, and she will bring this in mind if she is going to build a large system sometime:

> **Int**: “Do you think you have learnt something about the system, or is there something that you think you can make use of yourself? Is there something in the system itself that you think you have learnt from, is there?”
> **Bea**: “Well, this thing with the division in server and client, and such, that’s smart, isn’t it? So, that’s perhaps one thing that you will think about some time if you are making a large…, a large program in the future, but…, well…, and then there are many interfaces and all is built on, upon them, and I suppose that is something that you will learn sometime…, to start out from them. Especially, if you are working in a group or something, it can be useful to know how it works.”

7.3 The system is described as adaptable and dynamic

In the quote that finished the previous section, Bea seems to connect the system’s parts to interfaces when she describes that there were many (Java) interfaces and that everything built upon them. As we have already discussed in Chapter 5, interfaces can facilitate a division of a large system into separate and independent parts. Bea’s way to think of interfaces and parts approaches the description category elaborated on in this section.

This category introduces a new meaning – the system’s dynamic nature, and the smartness and usefulness of that property, which Bea also appreciates in the quote. In the descriptions belonging to this category, there are examples of opinions that indicate self-confidence, and an intimate relation to programming.

This way to describe the system embraces the other two categories, however it is distinguished from them because the descriptions gives evidence of deeper insights about detailed solutions, such as how the system uses “thin clients” and plugin modules, and how these are loaded from the server to the clients. The software for the clients, providing the graphical user interface, is described by Cia as something that does not contain very much itself; however, when it is started, it can request components from another program, the server:

> **Int**: “Did you think it was confusing that the classes was placed on the server side, so to speak?”
> **Cia**: “No, not really, because the client itself only had the ‘login’ and the ‘logout’. But then, all of these ‘tabs’ are plugins, as I get it. Therefore, it gets everything from the server, kind of, through the client node.”

And Git confirms that this feature is the interesting part of the system:

> **Int**: “I was thinking…, could you describe…, you can have one minute to describe this system, how it is built, what you can recall from it?”
> **Git**: “Eh, hm, the whole application is supposed to be a client/server business where you have a…, working with a database, now this was, I guess a, what’s the name, Microsoft’s own thing?”
> **Int**: “Access?”
> **Git**: “An Access database, with some set up tables. The one that concerned this was the registration part, which you had to work with. But now there was, there was you know, in order to set up students and courses and course instances, and whatever. And then, I guess, the interesting thing was to set up, eh, these, I don’t know the name, tab sheets, in order to put in windows in the application, where you could load up new, eh, new tabs so to speak, or plugins, in runtime. Eh, and what should I say? It was supposed that you could store students and courses and stuff there, and handle the relations between them. That’s it, kind of.”

This quote gives implications to believe that those who can describe the system in this manner really have devoted themselves to understanding the code, and have created a good model for their understanding of what really happens. The concept “plugin” is related to a situation where there are several clients involved, and that the users can have different levels of authorization that could affect the functionality. This means that a user’s client program, only receives the particular plugins that the user is authorized to use, and furthermore, that this facilitates changes and extensions to the system:

> **Int**: “If I ask you to describe the system as such…, essentially you have already done that, but could you give a short description, and if you can see any particular advantages or disadvantages?”
> **Hal**: “No, you mean what the system is all about, do you?”
> **Int**: “Partly, what the system is all about, and if you can see any particular advantages or drawbacks, in a multi user point of view …”
> **Hal**: “Well, it is a tool for administration of courses and students in a school. And it is very user friendly, both for the end user and for the programmer. If the programmer wants to add a panel, a new function, it was very easy. You log in as ‘root’ and you decide which users that should have access to it. You can create new users as well, with new levels of security, I guess, and which plugin panels that should be displayed. And this makes it very extendible, polymorph. And as yet, I have not seen any drawbacks.”

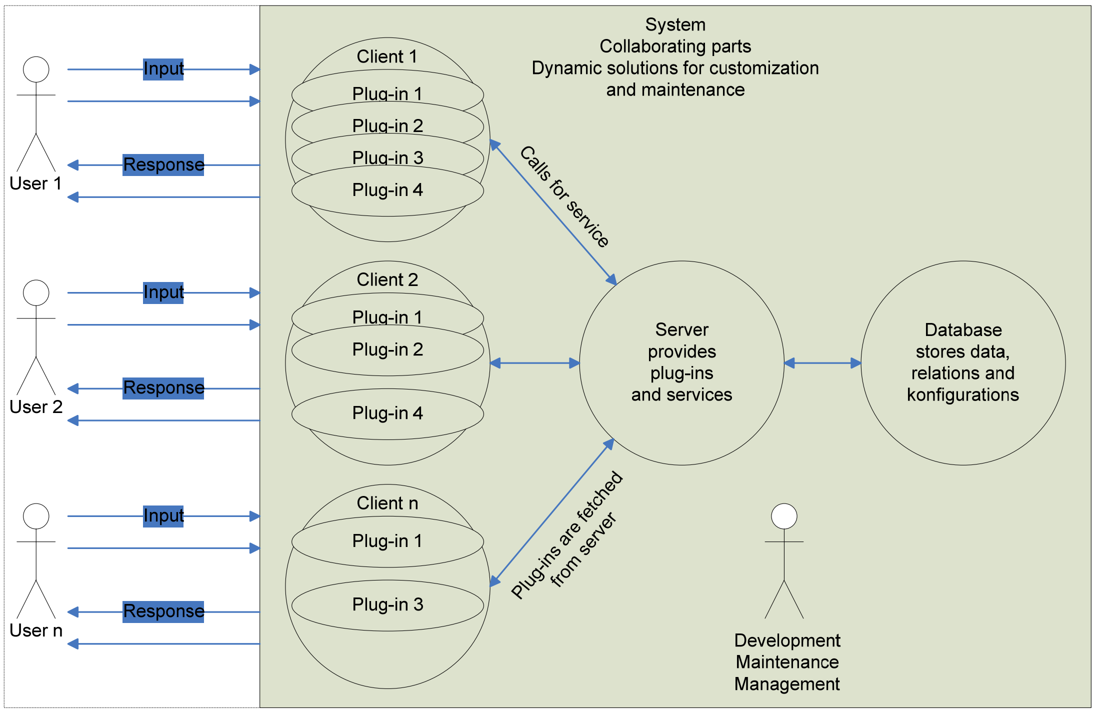


Figure 3. This is an image how the system is described in the third category.

These descriptions unveil an image of comprehensions that reaches far beyond understanding the system's functions, the operations the users need, or the individual programmer's situation – a new dimension that has room for circumstances that apply to the developers and maintainers during the system's entire life cycle.

Indeed, the students describe the system from a technical point of view, but at the same time, the technical solution has an inherent meaning that affects the surrounding circumstances. In this context, the students touch, explicitly or implicitly, aspects of the system's maintenance, such as how convenient it is to manage upgrades and additional modules, or the administration of user authorities.

Figure 3 shows several user roles and we can see that they have different configurations of plugins. Some of these actors could be a system administrator with high authority that allows for modifications in the system configuration. In the following quote, Hal descries how to go about if the system needs an upgrade:

> **Hal**: "You only have to make the changes in the server… There was a client application, you know, and that connected to the server. And I guess it is from there it gets its information and that was also where you should register the new plugin panels. Therefore, you only have to change the actual server, and that's almost kind of an requirement, these days. In a large school…, to run around to each of the clients…"

According to Hal, it is sufficient to put in a new or modified plugin on the server, which makes the work easy for the administrator. He indicates that this is a “must” these days. We interpret this as if he considers the system modern. Joe describes this aspect of the system in a similar way:

> **Int**: “A little bit more technical about how the system is built here, please.”
> **Joe**: “Well, this is a client. Ant then there is a server, also, running. The client asks the server for… plugins, when it starts … according to this interface, so the client…”
> …
> **Joe**: “Well, it is only at one place in the server where you need to build this new plugin class, and you have to go to the database and specify who should have access to it as well, I guess. And then it will happen automatically, that all 500 clients, or all who should have access to it, will get it. You don’t have to update each client.”

Again, we saw a description of how the clients request a set of plugins from the server, and hence, the changes imposed on the server will automatically be transferred to the clients. To Leo, this was something new. From his own learning perspective, he explains that this is what he learnt from his work with the system, and that he realizes that it is smart to use plugins in this manner:

> **Int**: “That sounds object-oriented, doesn’t it? Dou you think that you have learnt something today?”
> **Leo**: “I think it was very smart with plugins in this way. I have never thought that you can do it. You can add a file, and then it turns up in the program without recompiling it.”
> **Int**: “What advantages could one get from that?”
> **Leo**: […] “Then you can simply update the server and then the clients get updated right away. And the clients don’t have to…, you don’t have to distribute the program again and they don’t have to reinstall what they should have, because the upgrade takes place automatically.”

8 The outcome of the assignment

This chapter presents a compilation of the students' descriptions of how they approached the task and their course of action, as expressed in the interviews. After an analysis of these descriptions, and the evidences left behind in the computer's file system, we suggest characteristic features of the students' ways to handle the specific assignment in the experiment. The last section in this chapter identifies and describes three qualitatively distinct problem-solving types: "hands off," "waterfall," and "prototype". The result is not an outcome from a phenomenographic analysis and the types are not description categories in that sense. However, the phenomenographic approach inspired the way to do the analysis in the sense that we searched for the students' way to describe how they worked, and the meaning they saw.

8.1 Traces left by the participants and their view

The students got started with their mission without any major difficulties, with few exceptions. In one case, the student required a little guidance to get started with finding the paths in the computer's file system. After they got started, most of the students worked intensively and independently for two hours. Occasionally, during their work, a "curious colleague" came in to the "office" and asked the "newly employed" to explain what he or she was doing. The underlying thought behind that was to record descriptions of approaches and ways to see the mission and the system under construction. The curious colleague and the interviewer were the same person, who could use the collected information as an inspiring input to the following interview.

When the time was up eventually, some of the students were very close to being finished with their assigned task, and at least, most of the students had understood their mission and had started to work with the actual programming routine. However, others had not quite understood the task in a proper way, or could not find out how to get started with the problem in practise.

When the experiment and the interview were finished, we filed the student's entire project environment in a single Java Archive file (JAR) file, and if the student left any notes or sketches, we filed them too. The filed material has been analysed to make clear which activities that the students carried through during their work, and how they planned their solutions. The archived file systems contained much valuable information to study, and naturally, it was in some cases possible to give the students executables a trial run in order to see how far they had come. Their source code files reveal what they had accomplished, and to some extent, how they had done it. It was possible to see whether they had complied the system and their source code or not, and what information that was registered in the database.

In Table 4, we have compiled notes of what the students told us about their way to tackle the problem and what information we could gather from their produced materials.

Table 4. Notes of the students' descriptions of how they got started, and notes on what was found when their file systems were analyzed.

Person	**Summary of the student's description of the approach.**	**Conclusion drawn from evidence in the file system and trial runs whenever possible.**
Alf	Got stuck on jar and script files. Read docs and ran the program but was unsure if he was supposed to write a class of his own. Got help with this, and then he could get on with the job, and he created a "working" plugin.	Has compiled, deployed, and tested the system. Has done the HTML docs. Has created "MyOwnPluginPanel" by copying "CourseInstancePluginPanel". Has not inserted much code, but have changed two labels. Has compiled and inserted his plugin, and it is visible. Not complete.
Bea	Was confused at first and needed to run the program to see how it worked. Did the HTML docs, and found and understood the interface "PluginPanel". Created an empty class, copied the interface and parts from similar plugins.	Has compiled, deployed, and tested the system. Has done the HTML docs. Has created a class "StudentCoursePluginPanel" that implements the interface with dummies and some copied code from other plugins. Has compiled and inserted her plugin, ant it is visible. Not complete
Cia	Understood the code by tracing a client's calls to the server's methods, and located where to put her code. Started from an interface to cut down on the coding, but did not make any *dummies*. Never compiled or ran anything.	Has not compiled, deployed, or tested the system. Has not done the HTML docs. Has created a class "RegstudcoursePluginPanel" and has written lots of code. Has designed layout for the user interface. Has not compiled her klass. Not complete.
Dan	Struggeled to grasp the task and repeatedly read the documents, but was stuck on what a "registration" involved. Looked in the code. Never realized the ready to use operations.	Has compiled, deployed, and tested the system. Has done the HTML docs. Has not created any code or file. Not complete.
Eva	Ran the program once, and read the documentation to get an idea of how it worked and what to do. Focused on GUI layout, begun to write method stubs in her class, checked with other plugins, and filled in code gradually. Wrote lots before she tested her plugin.	Has compiled, deployed, and tested the system. Has not done the HTML docs. Has created "RegisterAdminPluginPanel". Has not compiled or deployed her own class. Compile errors occur. Has coded much and have combo boxes for student and courses. Not complete.

Fia	Wanted an overview of the system and what was already there. Ran it, checked the code, focused on the task, and returned to the documentation. Asked for help on where to write the code. Got frustrated and could not get started in time.	Has compiled, deployed, and tested the system. Has not done the HTML docs. Has not created any class or file. Not complete.
Git	Read the documents. Unpacked and checked the project's tree structure. Finally ran the program after some fuss. Missed a graphical model for the system. Misunderstood the HTML docs.	Has compiled, deployed, and tested the system. Has not done the HTML docs. Has created a class "RegistrationPluginPanel". Has inserted her class in the system and it shows, but is still only a copy of "CourseInstances". Not complete.
Hal	Read everything and focused on what was mentioned about database tables to register students. Ran the program. Started from an existing class and begun to design the user interface.	Has compiled, deployed, and tested the system. Has not done the HTML docs. Has created a class "RegisterStudentPlugin". The class is put in the wrong directory and is not possible to compile. Has sketched the layout in the comments. Not complete.
Joe	Read the documents and did the HTML docs. Found out how to run and see the program, played around in it, and understood plugins. Copied an existing plugin and registered it in the database. Started with designing the user interface and then examined how he could use the service object. Copied much code from other plugins.	Has compiled, deployed, and tested the system. Has done the HTML docs. Before but not after his class. Has created and compiled a class "RegisterStudentPluginPanel". Has used combo boxes for students and course instances. The plugin is introduced in the system and it is possible to register and deregister students at course instances. Complete.
Ken	Skimmed through the documents, explored the file structure, tried to get something started, and runs the program. Understood that a "tab sheet" was missing. Needed to see the database and its relations. Found a class to build on and thought about the GUI design, which components he needed. Did not try to compile his class.	Has compiled, deployed, and tested the system. Has done the HTML docs. Has created a class "RegistrationPluginPanel" and has started to code. Compile errors occur. Has sketched the layout in the comments. Not complete.
Leo	Read and sorted out what was relevant. Relied on what "was already there". Compiled and ran the program. Did the HTML docs and looked in it. Searched files using "find". Copied and started from another plugin, but removed lots of its code.	Has compiled, deployed, and tested the system. Has done the HTML docs. Has created a class "CourseMembersPluginPanel". Has inserted the plugin in the system. It loads and is visible, but it has no functionality. Not complete.

Table 5 shows the results of the analysis of the students' traces in the file system in a summarized form. It shows the progression in time, starting on the left hand side with how the students examined the software, followed by what contributions the students made, and the right hand side shows information about the new plugin. 'Y' denotes "Yes", 'N' means "No" and '-'

implies that this was not an option due to earlier decisions, actions or omissions.

Table 5. Conclusions of activities and results after an analysis of the traces that was left by the students in the file system on the computer they used

	Examining software N – Took notes on paper C – Compiled the software X – Deployed and executed D – Documentation in HTML				**Contribution** F – Created source file S – GUI-sketch in file P – GUI-sketch on paper C – Wrote code				**The new Plugin** C – Compiles OK R – Installed and running W – Works as it should		
Who	**N**	**C**	**X**	**D**	**F**	**S**	**P**	**C**	**C**	**R**	**W**
Alf [a]	Y	Y	Y	Y	Y[1]	N	N	N	Y	Y	N
Bea	N	Y	Y	Y	Y[2]	N	N	Y	Y	Y	N
Cia	Y	N	-	N	Y[1]	Y	Y	Y	N[3,5]	-	-
Dan	Y	Y	Y	Y	N	N	N	-	-	-	-
Eva	N	Y	Y	N	Y[1]	N	N	Y	N[3,6]	-	-
Fia[f]	Y	Y	Y	N	N	N	N	-	-	-	-
Git	N	Y	Y	N	Y[1]	N	N	N	Y	Y	N
Hal	Y	Y	Y	N	Y[1,4]	Y	Y	Y	N[4,5]	-	-
Joe	Y	Y	Y	Y	Y[1]	N	Y	Y	Y	Y	Y
Ken	Y	Y	Y	Y	Y[1]	Y	Y	Y	N[3,6]	-	-
Leo	Y	Y	Y	Y	Y[1]	Y	Y	Y	Y	Y	N

[a] Asked for help and was led to the directory that contained the source code for the plugin classes.
[f] Asked for help about where the *registration of a student to a course instance* really should take place.
[1] Copied a file with a similar implementation of a PluginPanel.
[2] Copied from a file containing the interface.
[3] There were compile errors.
[4] The source file was placed in the wrong directory.
[5] Never tried to compile the own file.
[6] Did try to compile the new file.

8.2 Types of problem solvers

In combination with the students' own description of their work, the analysis of what the students had done on the computers, gave a basis for a tentative characterisation of qualitatively different ways to tackle the problem, and we distinguish three problem-solving types:

Hands off

At first, both Dan and Fia read the documentation closely. Then, they gave the program a trial run, and then again, they returned to the documentation, but they could not figure out what to do or how to get started. They did not take any initiatives to *do* something concrete. For instance, they did not experiment with the code at all. The situation seemed to be a blockage for them, which partly paralyzed them. Perhaps they were too cautious and insecure to dare doing it or perhaps they had insufficient knowledge and training.

Waterfall
Cia, Eva, Hal, and Ken all wrote code, however, it seems they preferred to finish all of the coding work before they would (or could) compile and trial run the software – one sequential step at the time, as using the Waterfall model. Three of them sketched a design of the graphical layout as a comment in the corresponding source file (see Appendix E).

Our analysis shows that their code was not entirely correct, and hence, it generated quite a few compiler errors. It is remarkable how Cia never even tried to compile her code, not even the existing code she was provided with in the beginning, and therefore she could never execute the program. Hal could not compile his code because he put it in the wrong directory. Eva and Ken tried to compile their code, and they were in this way, in “dialogue” with the compiler.

Regarding the nature of the task’s requirements, writing lots of program code, more or less from scratch, was not an efficient method, and none of these students could execute their code when the time was up. None of them was close to succeed; however, they made a serious attempt.

Prototype
Alf, Git, Joe, and Leo got started by selecting a plugin module similar to the one they were supposed to develop, and then they copied its source code into a new file. Then they could easily make a prototype for their plugin by doing some simple changes, such as changing the name of the class, and then they could install it in the system.

Bea took a slightly different approach. Instead, she copied the specified operations from an interface and she was thereby forced to write a minimal implementation, using dummies, before she could test her plugin. Having this as a starting point, they could continue with the development and continuously see the effects of changes in their code by running it (see Appendix E). In this form of iterative development, the programmer gradually builds up the new code using already working code as scaffolding. This approach was successful indeed, but in the end, it turned out that Joe was the only participant in the experiment who managed to develop a functioning and complete plugin that complied with all of the requirements.

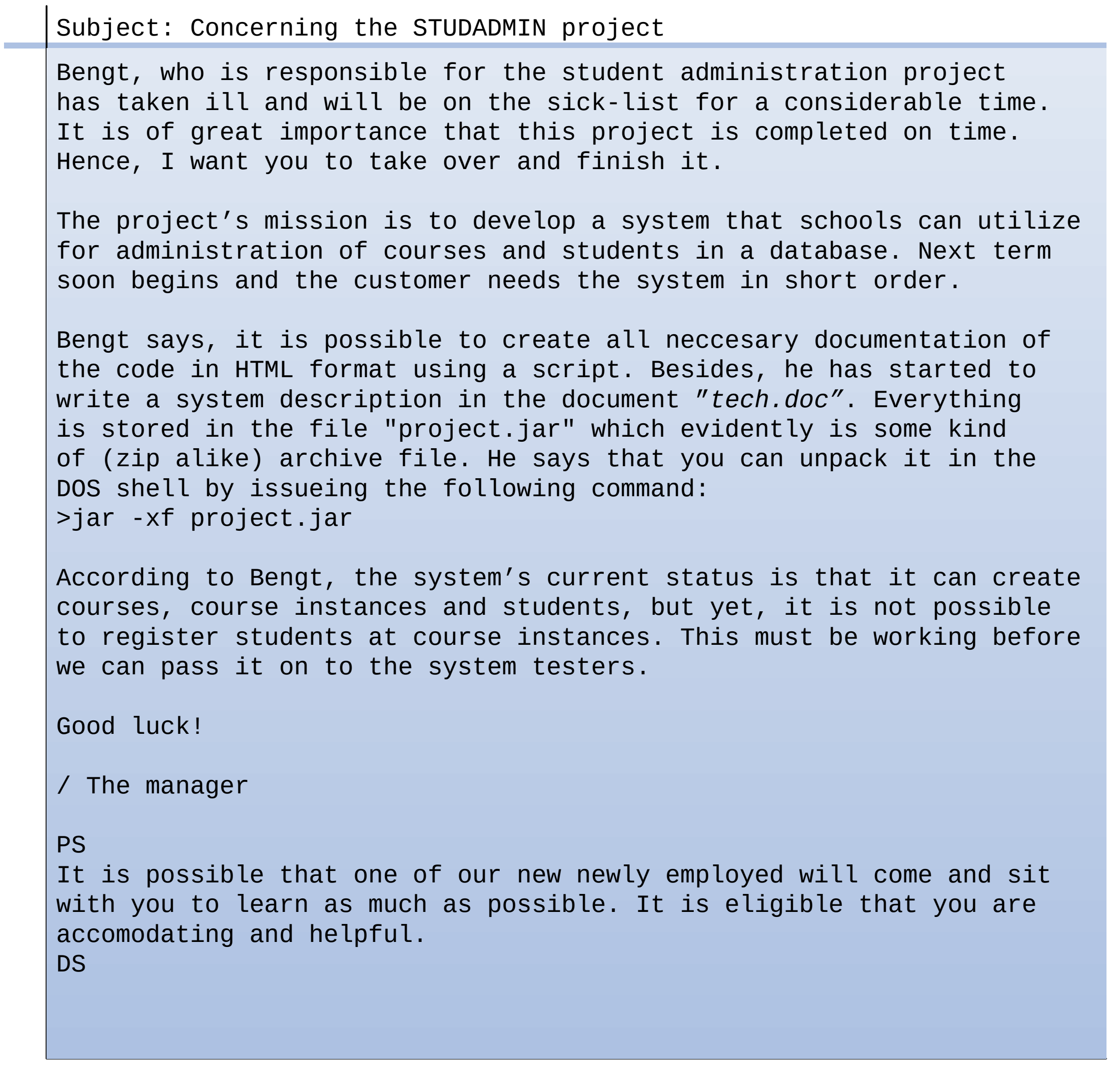
This approach required an understanding of the polymorphic aspect of the plugins, and that it was the use of interfaces that made it possible. Since all plugins in the system implemented the same interface, “PluginPanel”, the students could copy any of the existing plugins, and then they could change the implementation as they pleased.

9 Stories about the assignment

The interviews with the students contain many interesting stories that can bring valuable information to those who are interested in didactic matters in Computer Science. One of the posed research questions concerns how the students handle the unfamiliar situation exposed to them, and the specific problems that might appear. The transcripts reveal what the students thought they did, and how they approached the task. However, it is also interesting to consider what they thought and felt, and why they did things in a certain manner.

This kind of information is difficult to handle with a phenomenographic approach, as there is no obvious way to analyse the data. Still it is valuable, and hence, inspired by a "narrative approach", we have tried to find an easy to grasp way of structuring the evidence the students have provided. An appropriate way to present the data that preserves the spirit and tension from the students own words, was to structure their stories in a summarizing "story" arranged as logical sequence of themes. The story takes its start in how the students experienced to get started with the project, and it continues to describe the progress of the students' work. In the end, the story tells what emotions the students have felt, and what they believe they have learned from participation in this experience. Essentially, the story is data driven through the students' own voices, but our descriptions and interpretations set the course.

When the students had made themselves comfortable in the "office", the researcher gave only a very short introduction to the general conditions of the role-play and the experiment, and it contained no detailed description of what the students, acting as employed programmers, were supposed to carry out during their commission. This was the situation for the "recently employed", who now was alone in the office. On the desk, someone had left a note that read: "c:\jobs". This message pointed out a directory in the computer's file system. The employee explored this directory, only to discover yet another message in the text file "info_boss". In this file, "the letter from the manager," there was a clearly formulated mission: "our expert has taken ill," "complete the system in his place!" The letter was written in an abrupt manner, and between lines, you could sense some impatience and tension – the customer wanted to get the ordered system, and it was a great hurry. The letter described how to get started with the task, and obviously, the manager was not an expert in the field himself, because he only passed on what the expert had told, through the letter. Among other things, it referred to the system’s technical documentation (see Figure 4).



```
Subject: Concerning the STUDADMIN project

Bengt, who is responsible for the student administration project
has taken ill and will be on the sick-list for a considerable time.
It is of great importance that this project is completed on time.
Hence, I want you to take over and finish it.

The project’s mission is to develop a system that schools can utilize
for administration of courses and students in a database. Next term
soon begins and the customer needs the system in short order.

Bengt says, it is possible to create all neccesary documentation of
the code in HTML format using a script. Besides, he has started to
write a system description in the document ”tech.doc”. Everything
is stored in the file "project.jar" which evidently is some kind
of (zip alike) archive file. He says that you can unpack it in the
DOS shell by issueing the following command:
>jar -xf project.jar

According to Bengt, the system’s current status is that it can create
courses, course instances and students, but yet, it is not possible
to register students at course instances. This must be working before
we can pass it on to the system testers.

Good luck!

/ The manager

PS
It is possible that one of our new newly employed will come and sit
with you to learn as much as possible. It is eligible that you are
accomodating and helpful.
DS
```

Figure 4. The letter from the manager to the “newly employed programmer” came straight to the point. This is a translation from the Swedish original.

9.1 To get started

The manager’s letter gave a short outline of the system. It described the first required step to get started; namely, to unpack a file archive, and hence, make all the documentation of the system and its source code available on the computer. This seems to be a simple task for a Computer Science student, who should be familiar with file archives. Nevertheless, the extraordinary in this situation was the used archive format, namely the Java platform's JAR[14] format. In contrast to other usual archive formats, such as the well-known ZIP format, the JAR format was not supported through the operative system's graphical interface or any similar built-in application. In order to unpack the JAR archive, the students had to start the command based DOS shell, and not all of them were well acquainted with this environment. For Alf, it was problematic to handle JAR files in this way, and this was the first issue he brought up to discussion:

> **Int**: "First, 'like this', you said."
> **Alf**: "Well, no, but the first thing is when you came here, and it was JAR files, you know. I would sit down and read about JAR files first, how you do it and where are they put up. It was written, it was, but then I would, sort of, really have gotten in to everything that was written in these documents and read through everything, really, and then reflect about it, and then I would sit down and start to think about what I should do with the code, and the program and all that."

Not knowing how to manage the JAR files was very frustrating to Alf, and he was stuck on this first stage. Later during his work, the problems reoccurred because in many of its parts, the technical documentation referred to JAR files. For Alf, it was not sufficient to get the literal command that would unpack the archive, as the letter described it. It seems as though he wanted to get a deeper understanding for how the JAR archives and the corresponding commands worked, and therefore he could not accept to follow the instructions, passively. Without this obstacle, Alf argues that he would have been able to be more acquainted with the documentation, and as he continues, he explains that the mere fact that things are written literally is not enough. He would rather learn by experimenting and practicing by doing JAR files of his own:

> **Int**: "Hm, do you think that we should make it clearer in any way, for the future? Should there be an appendix about JAR files, or?"
> **Alf**: "Well, yes, I think that one should read about JAR files before this, because that is not the essential thing to know about, is it?"
> **Int**: "No."
> **Alf**: "The essential thing is to find, or kind of, be able to continue, and you should know what JAR files are, how you create them, and where they are put, and… because you spend unnecessary time on that, kind of."
> **Int**: "That is to read in the text, for sure, but you could have an…"
> **Alf**: "It says pretty much, kind of, but still. I would have tried to make my own JAR files, to see what…, where they go, what will happen."

[14] The Java Archive, JAR, software is included in the Java development kit, and among other things, it is used to assemble a "program" that consist of several class files and libraries into a single "executable" file. It is possible to use it as general archive software.

Cia was not stuck on the JAR files, and she seems to have gone straight forward after having read the manager's letter. She describes a top-down approach to get her understanding. Obviously, she had no difficulties with unpacking the archive, and to get started with reading the documentation and then to examine the file structure:

> **Int**: "Can you point out, in some way, what it was that immediately made you realize what you were going to do?"
> **Cia**: "Well, I guess it was when you read this first 'info boss' thing, that you were supposed to make a plugin. Then, first, you kind of, had to look at the big picture, what it was and then break it down into steps."
> **Int**: "And what made you realize how you should do it, eventually?"
> **Cia**: "Well, that was after I had gone through these files and read through, and looked how they communicate with one another, and that. In order to, kind of, be able to build something of my own later that could communicate with the others, without colliding, for instance."

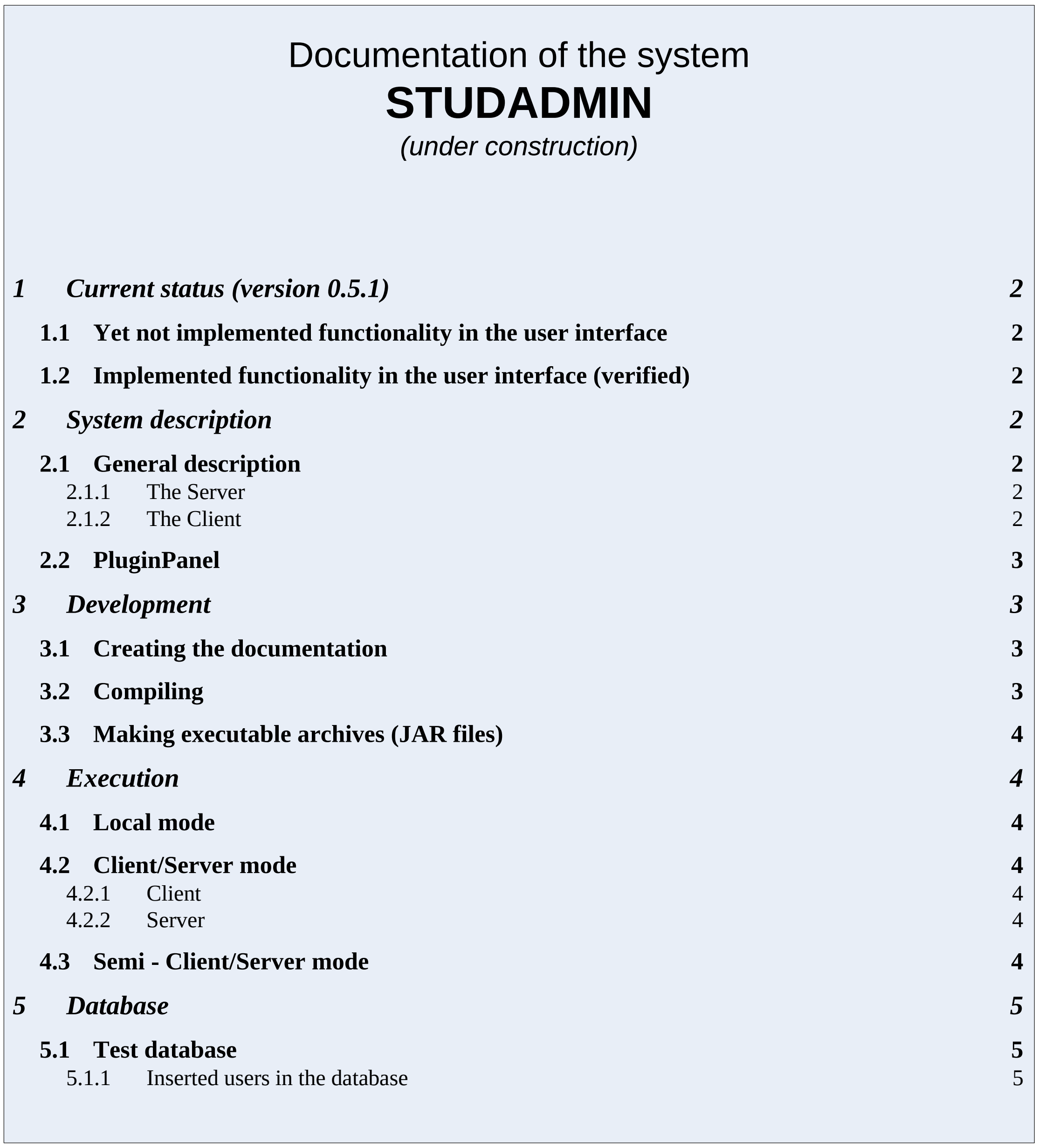
9.2 About reading the documentation

When the archive was unpacked, the JAR software created a directory tree in the computers file system. The directory contained many files, including the technical documentation, heaps of source code files, and other resources. At the top level, there was a directory named "project", and this folder contained the technical documentation of the system, in a file named "tech.doc". This document described the system, "STUDADM," on a conceptual level (see Figure 5). It pointed out some principal features, as for example the mechanisms that enabled a dynamic handling of new functions in the system. It described the development environment in terms of how the source code files were structured, how to compile the software, how to generate various versions of the system, and how to execute the programs.

The documentation did not describe the specific task that the students were supposed to do. On the other hand, it described the task indirectly, as it accounted both for the implemented functionality, and for what remained to do. Neither did the document point out the places where the students could put their additions and changes to the system.

Eventually, when the picture of the system and the mission became clearer to the students, the complexity of "the whole" appeared as well. As Alf puts it, the system was "large" and there were "files all over the place". He experiences the project as closer to a "real situation" compared with the ordinary programming assignments; however he believes, it is probably even worse in reality:

Int: … "But, eh, what was your first impression when you got started here and begun to read the documentation?"
Alf: "Well, I don't know, it was big, kind of… big. It was, you know… files everywhere and, yes, larger than I had expected, ha, ha, yes, that's how it was. But I realize that it's like this in reality… that it isn't sort of tiny and fiddeling… that it is even worse in reality for sure."



Documentation of the system
STUDADMIN
(under construction)

Figure 5. The table of contents gives an overview of the documentation. This is a translation from the Swedish original. Appendix D contains the full documentation.

All of the participants read the technical documentation of the system; however, they read it in different ways. One way to read it was to read everything through, from cover to cover, and several times in some cases. It seems as though Dan did not feel comfortable enough to leave the documentation and turn his focus towards other activities. Evidently, he had a hard time to capture the information in the document, and he had to reread what he already had read several times:

> **Int**: "But you also said that you read through the entire documentation…, I believe you said three times?"
> **Dan:** "Yes, and then there was a fourth time, to be honest."
> **Int**: "Wasn't there something special that you thought that 'I'm picking this thing now'?"
> **Dan:** "No, first of all I thought about what it said at the beginning, that you should be able to add and remove, you know. I guess, I thought that was a problem, and I kept that in mind because I wrote down some notes on a paper."

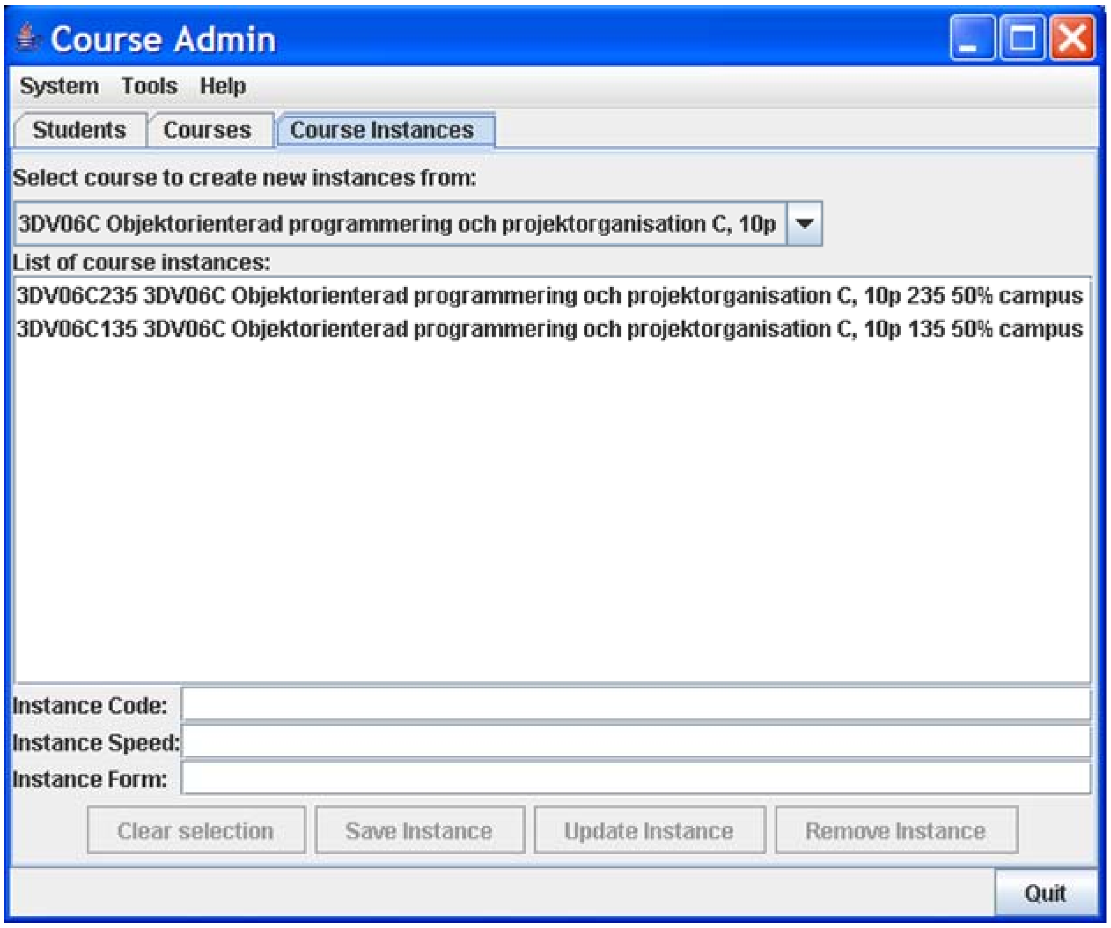
The other way was to read the document gradually, one piece at the time, and alternate reading with other actions, such as for example, looking around in the source code files, trying to compile the software, or executing the system.

> **Int**: "Do you think that the documentation…, I mean, there was a letter from the boss and also some 'tech.doc', or something. Do you think that is enough for a person of normal intelligence to get started?"
> **Bea**: "Yes, it should I guess, perhaps it…, well, if you are used to programming and have done this several times, then it is surely enough. I guess it does. But I am not so experienced, and having to read, kind of, that a long text, and divided, and… there is no real explanation, or, well, I don't know."
> **Int**: "How would you describe that document, what sort of document is it in your opinion?"
> **Bea**: "Eh, well, an explanation of how this system works, how all the parts collaborate and after all, perhaps…, well, now I don't know really…"

The students faced a task that in itself was rather narrow and did not require any advanced skills or knowledge for the actual programming. However, the complexity of the system and its context, forced them to get the big picture of how it all worked before they were able to find the proper place to write the code in. They also had to inform themselves about which system operations they could use. As expressed by Alf, this situation was confusing:

> **Int**: "How…, do you remember when you got on track with the solution, I mean how. When did you get a feeling that 'now I roughly know what to do'?"
> **Alf**: "I got that when I started reading, I guess. When I started reading that… 'tech' or whatever its name was. Here, kind of, ok, it isn't this, and it is that, so, and this, when you read this 'create and remove, register courses and course instances', ok, but not that I should write a class of my own. I didn't really know if there was something that you could alter so it takes a course instance, or if you had to write one yourself. And then I started, kind of, to 'am I really going to write a tab sheet of my own?', kind of, 'are you supposed to make changes in the GUI?', ha, ha".

We deliberately designed the task in a goal-referenced manner in order to create conditions for a variation in the ways to solve it. The task was to finish the software by adding a missing function to the system. On the computer screen, this would appear to the users as an additional form, or tab sheet[15], in the program's graphical user interface. There were already several tab sheets that could manage information of students and courses, and in the new sheet, the users should be able to connect registered students with registered course instances (see Figure 6).



Select course to create new instances from:

3DV06C Objektorienterad programmering och projektorganisation C, 10p

List of course instances:

3DV06C235 3DV06C Objektorienterad programmering och projektorganisation C, 10p 235 50% campus
3DV06C135 3DV06C Objektorienterad programmering och projektorganisation C, 10p 135 50% campus

Instance Code:
Instance Speed:
Instance Form:

Figure 6. The graphical user interface consists of pages, or sheets. Here it displays "Course Instances". A user can select a different sheet by clicking on its tab.

9.3 Descriptions of the task

It is interesting to see how the students conceived the task. Some focus on the "thing" they should make (the plugin) and what features it should have, while others tends to see it more from the user's point of view. Here are some of the students' descriptions of what they believed were their mission:

[15] Med skärmbild menas här en sida, eller ett formulär som kan innehålla textfält, knappar och liknande kontroller.

Int: “Could you describe your task, please?”

Bea: “Eh, yeah, to insert a new of these plugin panels, see, then you can register students at certain course instances. I guess you should continue to work on what was already there.”

Cia: “Well it was to make a plugin that registers students at course instances. And I had planned that I…, that you choose a course instance and then you type in the student’s identity number, and take that and compare it with the first name and the surname, to make sure that it’s the right student and the right course.”

Eva: “To write a plugin that should link courses and students, course instances and students together.”

Fia: “The idea was to complete the program into something working. Well, client/server-ish, that is. Then there’s something missing for registrations that existed on the server side, but not on the client side. And that’s what I should add to it.”

Git: […] “Well…, first I read this ‘boss note’ where he said what you should do. And, well, there was nothing exciting in particular, I guess, apart from that you should fix and store a student and a course together, kind of.” […]

Hal: […] “why, that was what he had written at the end, the boss, that you should add students and course instances, and then it was described in greater detail in the document description, create and remove registrations. That’s what the task was all about.”

9.4 The need to give the application a trial run

It turned out that most of the students had a great need of a tangible experience of how the application behaved when they executed it on the computer. For that reason, the majority soon focused on how they could run the program. It is true that the technical documentation described how to achieve this, but it was rather a complicated recipe.

First, the student had to compile the scattered source code files, and for this purpose, there was a DOS command file[16] prepared in one of the directories. Nevertheless, that was not enough; in order to run the system, the student must first pack the compiled class files into JAR file archives, and then distribute these packages to the directories that contained the start scripts. The student should accomplish this through issuing a specific command file that would make different versions of the JAR files and then copy them to

[16] The script file *“compile.bat”* compiles all of the source code files and puts the compiled class files in a parallel tree structure.

three separate directories. One directory for the server's package and start script, another for the client respectively, and finally a test directory that could run both the server and the client on the local host. All together, this meant that the students had to find out which of the system's parts that they should start, and the names of the command files. Most of them realized it was easier to use the test version on the local host, but some used the live version with separate server and client processes. Bea describes her urge to see the program in action, and her difficulties to get it running:

> **Int**: "But if I put it like this then, what did you try to focus on first, and what did you try to focus on then, and what did you try to focus on then, do you remember some chain of…?"
> **Bea**: "Well, yes, at first I wanted to see, get the program running and see what it did and how it looked, so that you could get some picture of how it worked. So, well, I compiled and tried to find… so I could run it, you know, and then I produced the html documentation of it, and then I didn't look in it so close in the beginning, ha, ha."
> **Int**: "How…, so the focus was on how you could get the program going. You kind of kept that in your awareness that 'how can I get this running?'."
> **Bea**: "Yes, because it said so…, that it should be able to start, so then, yes, to get some kind of general view, that's what I wanted."
> **Int**: "Was it hard to get it running?"
> **Bea**: "Eh, it said, it said how to do it, but you read all to fast and don't think about looking for it, that 'deploy'…, that one I missed the first time and I thought 'why doesn't anything happen, why can't I start it?', but then…"

Although there was a relatively high threshold to cross over to get the program running, most students managed to do it eventually, and that emphasizes its importance to the students. Let us hear Hal's opinion:

> **Int**: "Is my understanding correct, that you felt it was important for you to trial run the application?"
> **Hal**: "Yes, you have to do that, because it is important to know how it works too, and not just the code, but you want to see the results, what happens when I do something. I think that's important."

There was an exception to this rule, however. Cia never even tried to start the program. Instead, she focused on the source code files, and through them, she tried to understand how the system worked. She found the place to put the new source code file and started the coding by copying code from a similar file:

> **Int**: "This system… you haven't tried to run it, have you?"
> **Cia**: "No."
> **Int**: "Could I ask why?"
> **Cia**: "I read the code instead."
> **Int**: "So you rely on that it works, kind of?"

> **Cia**: “Yes, more or less. If I have had more time, I would have tried to run it.”

How can it be that it seems to have been of such great importance to try out the program before the programming begun – was it mere curiosity? Naturally, it could be a matter of poor documentation of the system and what remained to be done, but we could also speculate about our curiosity in the tangible aspects of the world that surrounds us.

In the world of computers and their user interfaces, designers have made an effort to meet the want for recognition, and they imitate the reality by using metaphors. It is true, that the written documentation formally described the system, but there were only a few pictures showing the design of the user interface, and therefore the experience of doing the trial run could probably help the students understand better. One important “tangible” feature of the program, perhaps hard to learn by reading, was the tabbed panes, which allowed users to select different use cases.

After running the program, the students could return to the documents and try to understand the task better. Fia describes how she wanted to get a picture of it by both reading the documents and running the program:

> **Int**: “Once you had tried to run the program and had made an opinion of how it worked. What came in focus then?”
> **Fia**: “Well, when I had got it started, then it came in focus, what I really should do. Then I hade made myself a picture of how it looked like. Then I went back to the documentation to see what really should be done…, and, that’s it. First, I wanted to create a picture of what it looked like, and then you find out more about what needs to be done. I had already read the documentation before, but that was also just to form an opinion.”

Ken describes it would have been impossible for him to solve the problem without the possibility to run the program and see its intended functionality. However, he appreciated the comments in the source code that sketched the graphical layout for each plugin:

> **Int**: “Did you feel that it was important to get the application running?”
> **Ken**: “The full application, you know, to get the application running whether it is finished or not, you get a good overview of what the person really is trying to do. And, had I not been able to start the application, one could say it would have been the end of it, because I mean, if I’m going to look through all of the code, in order to understand what the program is all about – it would have been impossible. Certainly there was in some of this codes…, there was in these panels…, they had done a…, they had sketched, I liked that, thought that you get a picture of how this particular panel is supposed to look like, then if it looks like this it’s something completely different.”

9.5 Some created a documentation of the source code

In the technical document, there were instructions for how the students could make an automated documentation of all the system's source codes in HTML format, using the utility program "JavaDoc". It creates a tree structure of hyperlinked HTML files that describes each class and interface in detail, and it documents them by listing their methods' signatures together with comments imported from the source code files.

This would provide the students with a documentation of the system's source codes similar to the well-known Java API reference that many teachers and students use as an important help in their programming courses. The procedure of doing this was simplified by a easy to use script, and it is remarkable how often the students seemed to miss this information. Or did they deliberately ignore it? Alf reveals his reluctant attitude to this:

> **Int**: "This html documentation… you did not like the idea of doing that, in my experience."
> **Alf**: "Well, yes, I'm kind of completely anti to that. I don't know. I never use to do any sort of…, never go into sort of. Not really… it's kind of really neat. We never did any of these ourselves, we didn't – in algorithms."

However, as Bea confirmed, some of the students thought it was valuable, or at least, they did not neglect to do it:

> **Int**: "And then later, how has focus changed during the process here?
> **Bea**: Eh, well, then I got in and read the documentation, you know, the one that I had created, and watched a bit how they looked like, and when I found the classes there, that it was built on. Then I opened those java files to see what the code looked like, and tried to get something sensible out of that."
> **Int**: "Could you make any sense?"
> **Bea**: "Ha, ha, that's the question, isn't it. Well it took a while, I guess, but yes, you saw how the interface, 'plugin panel', looked like and how, well, 'students plugin' panel for one thing, which had implemented that, how they had built up the methods, and to get some help, anyway."

Ken told how he associated the term "documentation" with something he would have to write himself – not as something that he could use. Could this be an effect of how teachers use the word in their assignment instructions – "hand in well documented code"? In the interview, he suddenly realized the intended meaning of it:

> **Ken**: "Do you know what I'm laughing about now? If you had ran this in the beginning then I think that you had got started much faster, I think so. But in most cases when you think about documentation… it's true that documentation is there for, if this kind of situations would arise. Someone gets ill and you have to complete this work, someone else must take over. And therefore the meaning with a documentation is that the next person should read though

the documentation and understand all this. But I didn't think when I started, instead I thought that when I'm done with this I will make a documentation in order for other people to understand what I have written. Not that I should understand what other people have written."

9.6 Strategies to get along with the coding

The following subsections tell stories about some typical situations and strategies for how the students got along with the task.

9.6.1 About getting stuck

The participants knew they could get help, however, they had to pay for it with one of their help vouchers, and as they possessed only two of them, they had to be economical. Otherwise, no one would help them when they really needed it. The limited means for help prevented the students' asking for help due to convenience reasons. Nevertheless, Alf, Dan and Fia really got stuck, and without any hints, they could not get on with the task. One of the major difficulties was to find the very place where they were supposed to insert their code. Were they supposed to write code in an existing class' source file, or should they make a new file, and in that case, where should they put that file, they asked themselves.

Int: "You took out a help note. Before that, did you sit there feeling unhappy for a long time?"
Fia: "Not very long."
Int: "Lucky, I came at that time. Your question, if I remember rightly, you knew that you should register students, but it was not obvious where you should do it, and how you should do it?"
Fia: "Yes, exactly, that was my question."

9.6.2 Delimitation and trust

Some of the students were confused and bothered by the huge file structure, and the numerous source files that were scattered in many directories. It was a question of finding the right places to work. Leo described how he limited his focus only to the parts he absolutely needed to use, and he trusted in the other parts' abilities to make the whole system work.

Int: "Can you describe the system's structure; do you have a grasp of how it all sticks together and how it is divided?"
Leo: "Divided, you mean the files, the codes so to speak, or?"
Int: "Well, you can take the classes as a starting point for example."

Leo: “Well, actually, I did not have time to look so much at that, instead it was…, well I can…, it was the documentation. I didn’t look at the other classes, see. I only looked in the files that were relevant for what I was supposed to do, but in the documentation, it said that it was divided in server and client and such. I don’t know if I looked so much at that really, I focused at the task and what was supposed to be done. Then I didn’t care about…, I relied on that he had done his job and that is would work. It was kind of in this way I thought on...”
Int: “But you focussed on what must be done, what is expected from me?”
Leo: “Yes, I checked what the task was and then what was relevant, those files that were relevant, I looked in these and didn’t care about the rest.”

9.6.3 To study and copy similar files

A strategy that turned out to be effective in order to get going quickly, was to take a look in a source file that was similar to the one the students should develop, and then reuse parts of that code. The looks and functionality of the various tabbed panes that could be selected in the program’s user interface were similar to each other, and consequently, the underlying source codes were also similar with respect to their fundamental structure. Hence, the students could learn and utilize much from them.

Int: “What did the focus look like then, when you came there?”
Bea: “Eh, I, well right then when I realized that I should make a class entirely of my own, then I started to look and compare with the others, you know, that also are these kind of implemented ‘plugin panels’, and checked how they were planned. I guess it was, well, to get some idea from them, how it should look like.”
Eva: […] “However, it can be a help, to look at other plugins. I did that very much.”
Eva: “Well, I didn’t look so close at the interface it self, I looked at the other plugins instead. And used them as in…, changed these implementations of the interface.”

However, they could go even further with this approach than just looking in files and copying parts. With some swift changes, the students could use a similar source file in its whole, and then it could be a prototype for the new plugin. Ken used this approach:

Ken: “And, once I had realized it, that I had to create a completely new class, of this specific type, plugin, well ‘PluginPanel’, then I started to think about what it should look like, graphically. Because I think it is easier to view things graphical before I start to get on with the work. And then I started to look around in the other panels and I noticed that this panel, ‘CourseInstances’, that, that it probably was pretty well suited for this kind of plugin. But of course, that you had to modify it, and that was what I was trying to do now.”

> **Int**: “So, you took the code as a starting point?”
> **Ken**: “I started from the code in ‘CourseInstances’. Because, really, it’s best to keep with the design that the person has tried to make, you know, tried to accomplish, instead of that you yourself come up with a completely different design. Then they would not fit in this particular program, at all. But I was really on the way there.” […]

Joe did the same thing, but as he describes it, he actually used the prototype, “as is”, when he installed it as a plugin:

> **Int**: […] “Joe, what came in focus when you started, what happened?”
> **Joe**: “The first thing I did? Well, I read through the document, I did. Tried to understand something at all. Find out how you start and see the program, overall. Then I went around in the program…, what it looked like, and then I understood this thing, that is was plugins. From the server you got these tabs, if you implement this ‘plugin panel’ interface. So, I copied a plugin that already existed. And it said here that you should put it in the database, so I found that table in the database. And added the one I created in test, to see that it showed up and to understand how it worked. Then I started to do the real thing, kind of. I started with the user interface. Didn’t think about the other stuff at all, actually. Only the interface until it was finished and I got it working. Then I started to look at how you do, really, to use the service object.”

What Joe described was a simple and wise way to produce an executable prototype version, simply by first copying a file, then renaming some names, and finally registering it. After this it was possible to test run the “new” class. However, he needed a good understanding of the whole to realize which file he should copy and where it was. This turned out to be a major obstacle for Fia, who had trouble to find the right paths in the file structure, and because of this, she never got time to write or copy anything at all:

> **Int**: “If we take a look at writing code. You never got on to write any code. Can you tell me something? Do you have any principals or some particular way or method for writing code?”
> **Fia**: “Copy as much as possible! That is my philosophy, and you change whatever needs to be changed. The most of the classes and interfaces are actually pretty much the same, so, that’s how I usually do it.”
> **Int**: “Could you refer to that as building skeletons that you later get back to and change?”
> **Fia**: “Well, to *use* skeleton code maybe, or alter what is needed in an existing program that looks similar.”

The act of copying sometimes seems a bit shameful. Teachers constantly remind the students that they must not copy their classmates’ code when they do their assignments. In addition, there is a continuous debate about plagiarism on the Internet, going on. It is a bit surprising though, that Hal expresses this feeling in this situation. He said that it perhaps was the wrong way to do it, and that it is easy to make mistakes. Indirectly, he motivates the copying with his purpose to extend the code until it fulfils the goals. He also describes that there are methods, ready to use, that he can call on the server side:

> **Hal**: “Yeah, right, well it is kind of the wrong way to go, I guess. It is easy to make mistakes, but… I took a copy of your ‘CourseInstancePluginPanel’ and was going to make a new panel for registering students, and what I intended to do was to extend it with an extra ‘combo box’ where you first choose the courses and then there is one where you choose the course instances. And then also to extend it with a list. First you have a list where you can see the registered students for the course instance you have picked, and then all the students in a list beneath. And then you can mark them in the list in the bottom, and add them to the selected course. And that’s what I was doing in this ‘plugin panel’, I tried to fix it. Because I saw in this interface ‘service.java’, there was already complete…, well, the method names were there, ‘findStudentsTaking’ some course, so when I choose that combo box, I can update the list with the proper students for the selected course. And that’s what I was changing in these action listeners for these combo boxes. Well, that’s about it.”

Unfortunately, Hal put his copy in the wrong directory, and hence he never managed to run, or even compile, his new plugin class.

9.6.4 To compile and test ones code

Most of those students, who managed to create a source code file for their plugin, tried to compile it and tried out if it was possible to get it working in the system. Appendix E shows screen shots of the students’ plugins as they appear in the user interface.

A strategy that provided the students with the opportunity to be able to compile and test run the system at all times, was to build a skeleton, or scaffolding, using method stubs. Eva described her view of stubs and their purpose:

> **Int**: “Good, then we are going to look at the actual process. Your focus, kind of, if you can remember and recall what immediately came into your focus?”
> **Eva**: “After I had a bit of an understanding about how I should build it, you mean? Well, the first thing I started to do really was to think about what the user interface should look like. Because when I had thought, when you had thought it out, kind of, how the program should work, and wrote in a way that it at least *is* an interface, that perhaps not does anything, then you can insert, well, so that it isn’t just stubs anymore.”
> **Int**: “Stubs, what is that?”
> **Eva**: “Well, methods without…, without code.”
> **Int**: “Explain!”
> **Eva**: “Em, you could say like…, if the button ’add’ should run a method that, you know, adds a student to a course instance, but you don’t… If you know

> that it's going to be used and it's called 'add', you can write the name of that method, but not write any complete code, just a little bit, whatever you can come up with at the time. And then when everything is finished…, if you want it to happen…, I mean that should add that student, *then* you can start thinking about it."
> **Int**: "What advantages do you see, working like that?"
> **Eva**: "Advantages, well, I haven't reflected about that, really. It's how I usually do."
> **Int**: "There must be a reason."
> **Eva**: "Hm, well, it's that you can test your methods, I guess. At once when you have written them"
> **Int**: "You can simply test your program."
> **Eva**: "Mm, to test if it crashes totally or just a little bit."

Most of the students strived to compile and run the code, but there were two exceptions from the mainstream. First, Cia, who did not compile the original code in the first place, and neither did she compile her own code later on, even though she explained how easy it would be to get it running:

> **Int**: "But then there were four, five, six methods that you should write, I suppose. Let's say that you wanted to try to run the program. Then, how much would you really have to implement in that…, in those methods."
> **Cia**: "Well, you need that 'getFocus'. The client calls 'getFocus' and that you know, so I took that from this 'CourseInstance'. In fact, I have kind of, built the skeleton that makes it able to work."

Then, Ken, who explained that he could not compile his code due to all of the things he had to finish first:

> **Ken**: […] "And then I haven't got a clue whether it should be compiled or not."
> **Int**: "You didn't try to compile it?"
> **Ken**: "I didn't even try to compile, 'cause I wasn't even half ways through. 'Cause once I had got it all together about this design stuff, how I had imagined it to look like… If I had come to that, I would have to, for each…, if I had selected a course, I would have to fetch all students that were registered on that course, and put them in this particular list. And then I had imagined that student…, a combo box with the available students…"

Working with gradually evolving prototypes seemed to be an effective and popular method, but it did not come natural for everyone.

9.7 How the situation was experienced

All of the students described how the scheduled time was too short in order to finish the entire mission, and hence they could feel that they worked under pressure. Apart from this sensation, there were other experiences of the lived situation, such as satisfactions, frustrations and disturbances.

9.7.1 Satisfactory, fun and interesting

Most of the students appreciated to participate in the study and said it was worthwhile, which in fact is a noteworthy opinion, as the experiment and the following interview occupied almost three hours of their time. Eva, Joe and Cia thought it was joyful:

> **Int**: “What do you think about getting into this situation?”
> **Eva**: “What I think about getting into this situation? I think it’s fun, but that’s because I’m interested in programming. Well, and it depends… If it feels completely impossible, or not. Because then it is not always as fun.”
>
> **Int**: “Was it fun?”
> **Joe**: “Yes, I think it was groovy. Really fun.”
>
> **Int**: “Hey, now we have been here for three hours. Perhaps I should apologise for saying it, but I have enjoyed watching your working. Do you think it has been worthwhile?”
> **Cia**: “Well, I guess it was fun to do this kind of thing.”

Fia and Hal expressed spontaneously, it was interesting to try working with the mission:

> **Int**: “Is there something you would like to add?”
> **Fia**: “No, I don’t think so. I just think it was interesting to see how something like this works.”
>
> **Int**: “Anything to add?”
> **Hal**: “No, well, that it was an interesting task to try out.”

When we asked Eva if she believed she could get into a similar situation in the future, her answer revealed a hopeful attitude to the prospects in her coming profession, and hence, the task was worthwhile for her.

> **Eva**: “I have never been in live situations in this way, but, well, it feels realistic. It could happen, I guess. I have not been out there yet and worked in this…, well, in this field. But, if it’s like this it would…, it probably will get rather fun.”

9.7.2 Not knowing what to do was frustrating

On the other hand, it was not very joyful for those who did not get started with the programming. Dan described how he felt overstrained, pressed and blocked, and that was not good for his self-esteem:

> **Int**: "If we return to this situation, you said that one feeling you had was stress, or what did you say, you felt…?"
> **Dan:** "I felt, sort of, that here I am in front of the camera, and perhaps I have to achieve something. But, one should not feel that way, 'casue that's what you said, just do your best, right?"
> **Int**: "I guess one could say so indeed."
> **Dan:** "Well, so I thought about it, but it didn't become…, I don't grasp it, but I kind of know…, I was kind of blocked."
> **Int**: "Can you perceive or remember other feelings?"
> **Dan:** "Yes, at the same time as you feel stress, you feel a bit bad when you don't cope with it. It doesn't connect, it feels worthless, you know."
> **Int**: "It feels a bit, what should you call it, feelings of low self-confidence then?"
> **Dan:** "Yes, in this situation, you know. In this situation, I felt low self-confidence in the situation."

Fia's experiences are similar to what Dan described. She did not feel comfortable with the programming language Java, and she described her frustration about the "hard to grasp" system that made her return to the documentation several times, to find something to focus:

> **Int**: "If we may look into your emotions during the time here, could you find some descriptive words for various moods that you felt?"
> **Fia**: "Well, you got a bit frustrated for a while before you had built yourself this conception about what you really should do when you saw how *large* the program really was. That was a bit hard to take in, and you haven't been doing java programming for a while. Forgot a bit how everything looks like and then you feel some frustration there, that it… [sigh] was quite much to deal with at the same time. So, but, well, I went back to the documentation a number of times and simply tried to limit myself to what I should do."

9.7.3 Not being left alone was annoying

It can be very difficult to be acquainted to another programmer's code, and it requires an ability to screen out the surroundings and become absorbed in the work, and for this, some people need solitude. During their work, we did not leave the students alone at all times, and this disturbed and embarrassed some of them, Bea in particular:

> **Int**: "How did you feel about this?"

> **Bea**: “How, well it…, I’m very uncomfortable about sitting like this. I prefer to sit at home where I calmly can check through things without having the feeling that someone is watching you. I get nervous and I don’t know what I’m doing, ha, ha, sort of. That’s how I feel, but, well, it’s a bit difficult to get into something that another person has done, it is. Before you get a general view, it’s tough.”

Now and then, a person came in to the office and started to ask questions to the student. This was the “newly employed” (curious colleague), mentioned in the manager’s letter, who wanted to know what was presently going on in the project. The purpose with this was to capture more data to the enquiry, and to start a dialogue that also could inspire the following interview. In despite of the good intentions, Bea felt that “looking over her shoulder” was very disturbing:

> **Int**: “Ok, that was about focus. If we talk about the emotional then… if you relate to your own feelings, starting from when you came here, and, well, until now, how have you…, could you describe them?”
> **Bea**: “Eh, well, I was pretty nervous when I came here; one thought ‘what is this, is it huge?’ and you have no idea about what you are supposed to do. And then you sit down and start reading and realise that it *is* huge and, ha, ha, I still don’t know what I should do. And, well, and then one have you sitting there, and that doesn’t help very much, ha, ha. Then you get even more…, ’cause I really have a hard time when somebody is standing behind my back and is looking at what you are doing. And, eh, it works better when you get out through the door, sort of.”

Eva had the same experience as Bea and described it as a general problem. However, she saw an advantage in making a break:

> **Int**: “How did you feel when I came in, and when I left?”
> **Eva**: “Mm, well, it don’t get at all as…, I can’t concentrate so very well when I’m not left alone, getting really disturbed then. I could just as well quit instead – for a while. But it can be a good thing to take a break too.”

9.7.4 Not being able to finish the task was frustrating

Only Joe completed his mission on time. Most of the others wanted to continue after the stipulated deadline. The following quotes from Ken’s interview exemplify this, as they describe how devoted he was to the task and how disappointed he was when he could not complete the job and see the result of it:

> **Ken**: “The only thing that you kind of, you get a bit disappointed of now, is that you have devoted yourself to a problem and now I have to finish this, you know, I will have to quit without having solved the problem. And then

you get frustrated. I mean, you want to see what it will look like when it's finished."
Int: "You're the kind of person that wants to finish things?"
Ken: "I'm not a person that gives up in a hurry. I happens that I sit up all night, you know, just to solve it, and…, yes."

9.8 What the students thought they had learnt

All experiences involve learning in some form, and one cannot escape from it; learning is something that constantly takes place. Nevertheless, this assumption does not imply that the individual necessarily is aware of her learning. Due to our attention's selective nature, we can choose, consciously or not, what we want to perceive. This selection process is not accidental and there is naturally some kind of priority of what is desirable to experience, or what we acknowledge to see. Hence, some form of valuation takes place when we describe experiences and learning from some situation. The interviewer asked the students[17] to describe what they had learnt from their participation in the experiment. Four identified aspects of the situated learning in the experiment have grouped the descriptions in a summarized form.

Personal aspects and reflections

Dan: "Good to be exposed to a shock, perhaps it will turn out better next time"
Eva: "To search more careful before one starts"
Fia: "How to do this kind of experiments"
Joe: "Never give up!"
Ken: "To use 'JavaDoc' in the future, and use good method names that associate"

The reality for developers

Bea: "Much larger than anything else"
Hal: "That this is a realistic scenario"

Technical aspects, the separation of clients and the server

Bea: "The separation between client and server"
Fia: "How client/server works; before this, it was only the different locations"

[17] We never asked Alf this question in the interview.

Technical aspects, the dynamic nature of the system's design

Bea: "Many interfaces that everything is based on"
Cia: "This about plugins, how they are used in a good way"
Git: "Plugin, to develop independently, to build out with further functionality"
Joe: "To send classes between client and server"
Leo: "Plugin is smart"

One kind of experienced learning concerned the technical solutions in the system, and another type of learning was the experiences of the situation and that it perhaps resembled the future as professionals. The third kind of learning is a reflective personal introspection where conclusions are made about own reactions and behaviour, and perhaps about how to change the future behaviour.

10 Discussion

In this chapter, I will bring out and discuss some different aspects of the conducted research. Partly I want to widen the perspective and try to interpret the results, and partly I want to share my personal reflections with the reader. Section 10.1 gives a short summary of my interpretation of the results and in Section 10.2 continues the discussion with an attempt to widen the results from the descriptions of the Java Interface. Section 10.3 treats the structure of the outcome space regarding the Java interface, from different angles.

In this mainly phenomenographic study, we certainly have heard individual statements from different persons. However, the study uses quotes primarily to give evidence for the categories of description on a collective level. Many utterances from the students do not belong to the core of the study, but still they are valuable and worth to consider if we want to gain a better insight into the students' individual situations. Hence, we have reserved space for "the voice of the individual" in section 10.5.

Chapter 8 described how the students approached their task. An interesting reference is Shirley Booth's dissertation and her analysis of some students' approaches, in a study on how students learn programming (Booth, 1992). Section 10.6 discusses the similarities and differences between the studies.

10.1 An interpretation of the results

My phenomenographic results concerning concepts reflect that the more advanced description categories connect to each other. The uniting link is the expressed overall view. The "high quality" descriptions of the concepts interface, plugin, and system, all express an integrated understanding for important principles within object-oriented programming. These descriptions elucidate a purpose of using interfaces and plugins in the system; the purpose is to get an adaptable and smart system that facilitates for the work with maintenance, installations and further development. In this manner, there is a clear connection to the professional reality and its perspectives.

While the more advanced categories converge, the less advanced categories diverge from each other and get more specific and concrete. An imaginary student, hypothetically equipped with only the concrete way to understand, would not have the apprehensive perspective that ties the concepts together. This understanding is more fragmented and it does not comprise a professional point of view! I claim this is one of the obstacles for a student to experience a concept in a more advanced way. Largely, the wholeness point of view is what carries on the understanding. Influences from other concepts and situations widen the perspectives and help to lift the understanding to a higher quality. For a beginner, who cannot relate the phenomenon to other situations and experiences, it is more difficult to abstract and realize general advantages.

10.2 Widening the perspectives – a further interpretation

The phenomenographic terminology distinguishes the following two aspects of experiencing: “the referential aspect” and “the structural aspect” (Marton & Booth, 1997, pp.86-88). The description categories in an outcome space give expressions for various ways of describing and understanding the meaning aspect of a concept, that is, descriptions of what the concept means per se; denoted as the referential aspect of experiencing. The structural aspect of experiencing a phenomenon alludes to what persons focus on while they describe it, and comprises both the phenomenon’s internal and external horizon. The phenomenon’s contour and its inner parts constitute the internal horizon, while the external horizon includes the entire surrounding context that makes the background of the particular way to experience.

The concept “interface” is particularly interesting as it has an immediate connection to teaching object-oriented programming. Since the first analysis focused on the referential aspect and I could see that the collected data contained untreated and valuable information, I wish to bring forward and discuss what the students discerned and focused in the background, based on interpretations of interview data, implications of the description categories, and my own understanding and experiences from the experiment. Table 6 shows elements of the structural aspect of the interface concept related to the description categories. We identified the following elements that illuminate the external horizon of experiencing the interface concept:

- How the students see the usefulness of the interface as an artefact in the programming process, or in a wider perspective of software development.
- The immediate or indirect associations to various phases in a software development process that the categories imply. One way to define a distinction is to divide the software development process into five phases: “design-time,” “implementation-time,” “compile-time,” “run-time,” and “the future”.
- The actors, artefacts and roles that the students associate in connection to the various ways they see the meaning in interface, and the context to which they belong.

In the first category in Table 1 (page 40), focus is set on that the programmer should solve a task by writing code in a textual form. The interface can provide help to do this easier as it constitutes a to-do list and a text to start from (a skeleton). There is a relation between text files, in the sense that the students focus on writing a class in a specific text file with help from existing code from the text file that defines the interface. From a time point of view, the students focus on the short-term goal (now), with their attentions entirely bound to the phase where you type in code (implementation-time).

Table 6. An extension of the outcome space

The referential aspect of how the Java interface was described. See Table 1.	The structural aspect – what was focused on and discerned in the surrounding situation		
	Advantages and benefits for a programming situation	The time perspective	Actors, artefacts and roles
To-do list for operations.	A convenient way to write code, copy and paste. The interface can be used as a template when a class is about to be coded. To know what to write.	Implementation time; here and now.	Focus is on *me* and my program, the class. The interface plays the role of being a text. The editor.
Declaration of contents, specification of operations.	Correct code: the implementations of the operations are verified. The specifikation and implementation are separated but still connected by the contract: class X implements I.	Several phases are involved: design-time, implementation-time, and compile-time.	I, the team, and the client. The interface plays the role of being an abstract contract. The implementation is verified by the compiler.
A datatype for reference variables, and implicitly for objects.	The interface is a type which can be used by a client to refer to compatible object types. These are different implementations that fulfil the declaration of contents. The specified operations can always be invoked through this reference.	Several phases are involved: design-time, implementation-time, compile-time, and run-time. The run-time perspective reveals the inner life of the program.	Objects, references, variables, datatypes, classes, client and server.
An open connection towards new and unknown objects.	Polymorphism, low coupling and low dependency between code and developers, ability to introduce modifications, and reuse of code.	Design-time, run-time and also the future administration; updates, corrections, system management and maintenance.	New and unfamiliar objects, modules, users, the outer world, and the professional role.

The second description category introduces abstract properties to the interface concept in addition to its textual appearance. A relation between the interface and the class stands out on a higher level that resides above the pure textual level. The interface constitutes an agreement, or a contract, between involved parts. The class depends on the interface that imposes the class with a prescribed behaviour. This way of understanding shows an awareness of various phases in time: design, coding, and compiling. The descriptions implicitly put the interface in relation to the time when the contracts were formulated as result of a planning effort (design-time). At the same time, the descriptions associate to the phase where the contract is "signed" by the binding between the interface and the class, using the keyword "implements", and the actual implementation of the specified methods (implementation-time). Finally, the descriptions refer to the time when the compiler verifies the code stipulated for in the contract and the contract itself (compile-time). Focus is not only set on the individual programmer, and how he or she can use the interface. The descriptions implicitly point at other actors, artefacts, and roles, and how their corresponding perspectives can view the interface: the designer who has defined the interface (the contract), the programmer who implements the interface (signer of the contract), and the compiler that verifies that the contract is fulfilled.

The third category expresses insights into a software's "inner life" when the program is executing, and here the object, the interaction between objects, and references are central concepts. At this level, the descriptions introduce the "data type" as a concept and abstraction. Objects of various types exist in run-time, and the program must treat them with handles in the form of reference variables that have compatible types. The interface provides one such kind of compatible type that the program can utilize as reference variables to handle all the objects that implement the interface. The reference's type determines which methods are callable in an object, and in this way, the interface defines and delimits how the program can use the objects. This insight expresses a higher abstraction that is close to the idea of polymorphism. In this category, focus shifts between the part that uses an object (the client) and the object that provides the operations (the provider or server). In addition, focus is shifting between the discerned phases design-time, implementation-time, compile-time, and run-time.

The fourth category lifts the level of abstraction further as it expresses and emphasizes the polymorphic property of reference variables of interface type. It is obvious that the interface can be used to create polymorph reference variables that can refer to various objects, all having compatible types as their corresponding classes implement the interface. Compatible and exchangeable types are central components for the understanding of the concept polymorphism in object-oriented programming languages. Using interfaces as a systematic principle in the design reduces the degree of dependence and bindings between objects in a program, as well. The relations between the parts in the program become purely abstract, as the parts do not rely literally on other part's specific implementations. The interfaces describe the "naked" abstract semantics in a way that is free to realize, and in this manner, the designer separates the specification from the implementation. Systems, which already from the start have a modularized character and a need for exchangeable components, can utilize this technique. To realize the advantages of this requires a deep understanding of building systems and the consequences that can appear without a similar way to work. A perspective of the future maintenance is apparent and the descriptions foresee a need for improvements, additions and upgrades in the system, and a crucial aspect is to provide for the possibility to handle this in a smooth way. The descriptions reveal an awareness of the consequences for the involved parts and that one must consider the user perspective. This category of description takes a professional perspective and characterizes a mature approach to programming and software development.

We can recognize this pattern of widened perspectives within the description categories in other studies. Shirley Booth (1992) showed several results with a similar structure. The students' experiences of programming varied from a *computer-oriented activity*, to a *product-oriented activity* (p.94, p.101), and their conceptions of learning to program varied from *learning a programming language*, to *becoming part of the programming community* (p.119). In Christine Bruce et al. (2004) we can see the same tendency, where the lowest level focus on assignments and deadlines, but the most advanced level focuses on the programming community (p.148).

What conclusions can we make from this second analysis? There are several different foci involved, and clearly, we can see the increasing complexity in what the students associate with the interface concept. The advanced levels of understanding integrate several related factors at the same time, and motivate the way to structure the software by the benefits it gives considering the whole enterprise: maintenance, customers, users, and business. I would say that the advanced ways to think about software reveals a mature attitude and a professional perspective on software development.

Certainly, we can consider software from different levels of abstraction, all having a meaningful purpose. For example code as text, syntax, variables, functions, data structures, system operations, classes and objects, runtime machinery, modules and components, version control, maintenance, lifetime cycle and economic aspects. For a rich understanding, it is important to be able to move unhindered between the abstraction levels and see the effects of a change in code from different perspectives. In my view, the Java interface is an artefact that is a manifestation of a professional perspective. To be able to understand it in depth, the learner should be aware of the underlying motives for its existence.

10.3 Thoughts on the structure of the outcome space

One of the criteria for establishing qualitatively distinct categories in a phenomenographic analysis (see Chapter 3) is that the categories should have a logical relation to each other, often hierarchic and inclusive.

The person who makes the analysis should have good subject skills and should be well acquainted with the learning goals for the educational context. Otherwise, it would be difficult and almost impossible to assess the complexity of the individual categories and to range them in a meaningful hierarchy during the analysis.

In the following discussion, I will relate my results to other frameworks and perspectives, in order to support the way in which I have structured the outcome space. I will focus on the outcome space concerning the concept "interface" (see Table 1, Chapter 5), as it is a central concept in object-orientation, which should be of general interest for programming teachers.

First, I will discuss how that outcome space relates to two general and well-established educational taxonomies, comparing the complexity and depth of the categories' meanings with "corresponding" levels in the taxonomies. Secondly, I take a completely different approach, and speculate from an "object/process duality" point of view.

Educators often use taxonomies in educational situations to formulate and rank learning outcomes and to support construction of questions for examinations and tests. Two examples of such taxonomies, are for example the Bloom taxonomy (Bloom, 1984), and the SOLO-model, *Structure of the Observed Learning Outcome* (Biggs & Collis, 1982).

Lars Owe Dahlgren points out (in Marton, 1986a, pp. 45-49) that Bloom's taxonomy is a theoretical construction that has not evolved from studies of actual outcome of learning. However, the SOLO taxonomy bases its attempt to classify levels of outcome on empirical studies, and it has a versatile application area. Entwistle and Marton also claim that SOLO can enlighten the categories in the phenomenographic outcome space:

> "The different outcome levels, if they exist, can be in many cases described in terms of an SOLO taxonomy, or simpler as an attempt to either explain, or describe, or merely mention aspects of what has been learned" (Marton, 1986a, pp. 290)[18].

The present discussion aims its interest towards the theoretical subject field perspective as well as the empirical perspective, and hence, it is interesting to see what happens when we relate the categories of description to both of the classifications. The taxonomies define categories for different

[18] The reference points to a book written in Swedish. This is our own translation of the quote from Swedish to English.

levels of learning and learning outcomes. Bloom's taxonomy defines goals for learning in six levels:

1. *Knowledge* – is the ability to recall and recognize information.
2. *Comprehension* – is the ability to, express understanding of meaning in one's own words; to interpret, translate, and extrapolate.
3. *Application* – is the ability to apply information, rules, and principles, to achieve a result, in other words, problem solving.
4. *Analysis* – is the ability to identify parts, partial functions, and structural principles, to understand inner relations and to identify motives.
5. *Synthesis* – is the ability to develop new unique structures, systems, models, approaches, and to combine ideas.
6. *Judgement* – is the ability to evaluate the wholeness of the concept in relation to the world and external conditions, to make judgements and strategic comparisons.

Now, we attempt to connect this taxonomy to our study's outcome space for the interface concept, and we will do it by trying to relate each category of description to the levels of Bloom's taxonomy.

The first category, "*to-do list*," does not express analysis, synthesis or relevant judgements. The set of values that appear in the interviews focus on that it is good to have an interface due to practical reasons, from a personal point of view. The applications of interfaces and the achieved results are trivial. However, there is a clear conception and recollection of the concept and there is a subjective understanding, although it is not very deep. Hence, this corresponds to level one and two in Blooms taxonomy, with a weak connection to level tree.

The second category, "*content declaration*," includes knowledge, a deeper understanding, an application with connection to classes, and the understanding of the contract relationship between the class and the interface that implies a certain ability to see structural principles and good motives. This relates well to Bloom's level three, and has some of the properties of level four.

The third category, "*data type and reference*," further deepens the motives and principles, combining the interface concept with reference variables and the various objects they can refer to, and this puts the interface in an "outer" frame of reference. These descriptions reveal an analysis of the use of the interface concept (Bloom level four), and to some extent the described consequences reflect a synthesis (Bloom level five).

The fourth and final category, "*open connection*," is characterized by its clear motives, approaches, judgements and a comprehensive view on software development, which can be related to Bloom's level six, where judgment and the wholeness are desired.

The SOLO-taxonomy suggests five categories that Dahlgren (in Marton, 1986a, p.47) summarizes in the following list[19]:

1. In the *prestructural* category, the answers are denying, tautological, transductive, and bound to the concrete, in relation to the given conditions in the question.
2. In the *uni-structural* category, the answers contain "generalisations" only using one aspect.
3. In the *multi-structural* category, the answers contain generalisations only using few independent aspects.
4. Characteristic features of the *relational* category are induction (the ability to make conclusions from experiences) and generalizations within a given or lived context by the use of related aspects.
5. The category *extended abstract*, has elements of both induction and deduction (the ability to make conclusions from premises). One characteristic feature is the ability to generalize the current context to situations that are not part of the question's prerequisites.

This taxonomy describes a classification of characteristic features identified in answers from real persons in empirical studies, and this makes it suitable to compare with our categories of description in this study. As previously with Bloom's taxonomy, we attempt to connect the SOLO taxonomy with the outcome space of the interface concept.

The first category, "to-do list," is certainly bound to the concrete and tangible, however, it does not express tautological nor transductive reasoning. The descriptions in this category makes a generalization as they imply that interfaces in general can be used as to-do lists from which text can be copied. Hence, we can exclude the pre-structural classification, and rather, we classify the "to-do list" category as uni-structural.

The second category, "*content declaration*," describes the interface richer, and it is ascribed a meaning that comprises and influences other concepts, and in the third category, "*data type and reference*," even more relations to other concepts are involved. These categories clearly have a relational character.

Finally, the fourth category, "*open connection*," expresses induction, deduction, and generalizations to a considerably wider perspective, and hence, we can relate it to the SOLO category "*extended abstract*". This category ascribes features to the interface concept that not in any way are evident

[19] Translated from Swedish.

from the mere syntax of the Java programming language, on the contrary, an understanding of these properties build on experiences and understanding of the possibilities given by the language related to things outside Java.

Table 7 summarizes the attempts to relate the taxonomies to the interface concept's description categories. One interpretation of this reasoning is that the categories in the outcome space mainly follow the taxonomies' levels and stages, which is an argument for that the hierarchic structure of the outcome space is meaningful and relevant.

Clearly, we have seen how the SOLO taxonomy has an inclusive and hierarchic character in itself. This leads us to a discussion of the inclusivity of our outcome spaces, and again, we will focus on the results for the interface concept. The second category is an obvious augment of the first category, as the second category extends the meaning of the list as the literal program text, into a list of content in a wider, more abstract sense, which includes the insight of a commitment between parties. The third category presupposes an understanding of the connection between interface, class and object, since it views the interface as a data type for reference variables that can connect to precisely those objects that implements the interface. A fundament for that understanding resides in the second category. The fourth category represents an understanding of interface in relation to polymorphism, modifiability, and exchangeability, which in many object-oriented languages require the type of relation between references and objects that the third category describes.

Table 7. This shows the attempt to map the levels in the taxonomies to our categories of description.

Outcome space for *interface*	Connection to SOLO category	Connection to Bloom level
To-do list	S2	B2
Content declaration	S3	B3
Datatype and reference	S4	B4
Open connection	S5	B6

Thus, we can summarize our argument by concluding that there exists an inclusive relation between the first two categories, and that it has the type "*B is an augment of A*." The relations between the second and third category, and between the third and fourth, have the type "*B assumes A*." Hence, using my definition of inclusivity (see chapter 3.2.1), it can be concluded that the structure of the outcome space is inclusive[20].

A different way to view the hierarchy of the outcome space and its inclusiveness could be to use a perspective that considers learning as a particular form of gradual evolvement. My interpretation and conclusion of the follow-

[20] An outcome space is not necessarily linearly inclusive; it can have ramifications and other kinds of relations (Åkerlind, 2005, p.12).

ing unconventional theory of learning is speculative, nevertheless, I find it interesting and thought-provoking.

Orit Hazzan points at similarities between learning in mathematics and computer science, as for example programming, and she uses Anna Sfard's perspective on learning which means that there is a dual relation between the operational (process) and the structural (object) understanding (Hazzan, 2003; Sfard, 1991). Sfard, identified this relation by studying the history of mathematical development and she claims that we can find it in the individual's learning process, as the operationalization, the objectification, and the abstraction are interwoven in an interacting process, where the learning gradually reaches higher levels. Sfard argues that the operational understanding always comes first. This phase of learning is what Sfard calls *interorization*. The meaning of the learning that the learner can see is to master the processes.

After a while, a sensation of structure come into existence in the understanding of what they are processing, and that is a manifestation of generalizations, in other words abstractions. It is during this *condensation* phase that the learners begin to get a comprehensive view; the wholeness of the processes they conduct. The condensation phase lasts until the learner begins to see "the abstract" as a unit of its own, entirely disengaged from the processes in which it originally appeared, when the abstract has become an object, through *reification*[21].

The reification process is the last phase in the present iteration of learning, and it involves a fundamental ontological shift in the view of what the learner is doing, from a procedural perspective into a structural point of view. Now this newly constituted object can be involved in new procedures, at a higher level. However, Sfard indicates, it is necessary that the new object is part of new operations (in the next interorization) before it can be fully reified by the learner.

Finishing off the discussions on the structure of the outcome space, I will attempt to enlighten the categories for the interface concept taking a process/object perspective. We could consider the "*to-do list*" conception of the interface as procedural stage, where the focus is set on doing something practical with the textual contents of the interface, which would be an interorization. The "*content declaration*" represents something more than what the students can see literally, they discern an abstract meaning, which would indicate a condensation. In the description category "*data type and reference*," the students have reified the interface into an "object" in form of a tangible data type that is used in other processes. Finally, in "open connection," the reified object is used in a wider context where new abstractions possibly could be crystallized, as for example design patterns (Gamma et al., 1995), which would lead to a new interorization phase, on a higher level.

[21] The term reification means that something abstract becomes materialized, e.g., a model.

10.4 The intended and the lived object of learning

The *intended* object of learning is the teacher's perspective on what the students should learn, whereas the *lived* object of learning is the object of learning from the learner's perspective; what is actually learnt (Marton & Tsui, 2004, p.4-5). When I compared the outcome space for the concept "interface" with how interfaces appear in the experiment's software and in course materials, I noticed some differences in the intended and the lived object of learning. The aspects I missed (as being a teacher) were certainly nuances of what appeared in the outcome space; nevertheless, the students never articulated them explicitly.

The *filter* property and the *multiple interface* property, relate to interfaces and classes. Since classes can have more operations than the implemented interface specifies, we can use interfaces as *filters* to limit clients' access to operations. If a class has a number of operations that we can group logically in some way, such as read and write operations, and we want to allow full read and write access only to a few trusted clients, the programmer could create two different interfaces; one that specifies all methods and another that specifies the read operations only. Since classes in Java can implement *multiple interfaces*, the programmer would only have to change one single line of code in the class and "hand out" the appropriate interfaces. The client with the read interface can only call the read operations, while "trusted" clients can call all operations.

We can use a variant of the *Abstract Factory* pattern (Gamma et al., 1995) combined with interfaces to control the creation of objects and to restrict the dependence on specific implementations. In the experiment's software, an intermediate layer between the database and the rest of the application controlled the creation of all data objects, and their specific class names were restricted to (hidden in) this layer. The code outside this layer, handled all data objects only through interface type references, which made it impossible to create faulty objects or alter data in an inappropriate way.

Other aspects are *interface inheritance* and *multiple inheritances*. In Java, classes can only inherit from one ancestor; however, interfaces can in fact inherit from an arbitrary number of ancestor interfaces. In the example above, the "read and write interface" could be a subtype to the pure "read interface", and hence, it would be feasible if the read interface was defined first, and then the read and write interface could inherit the read interface and simply add the write operations.

Even though the programming courses and the experiment's software covered these aspects, they did not appear in the interviews. One explanation is that the students' foci were set on their particular task and they associated only to the aspects that were significant in that particular context. On the other hand, for that particular purpose, the students reflected very important features of the intended object of learning.

10.5 The voice of the individual

The phenomenographic results apply to a collective level. However, it is important to remember the individual and the individual's experiences of programming and studying. The interviews revealed much information that partly was outside the subject that the study intended to investigate, but still, this information is valuable.

All participants in this study contributed to the results in a positive manner, and there were no serious "misconceptions" of the studied concepts. From an educational point of view, each description and expression of relevant understanding can contribute, more or less, to a learning of the concept. The most successful person, who actually was the only one to complete the mission into an operational application, had a broad comprehension of the interface concept. He brought experiences from programming with him to his studies, and he had now and then worked as a developer, but he had never used Java in those circumstances.

It is significant to know that there are students who, in their own experience, are having a hard time to obtain skills in programming, and they are having difficulties to turn abstract descriptions of programming principles, in practise into real applications. In the following, we will provide space for two individuals and let them tell their own stories. In the first quote, Dan tells us how he experienced the new situation when he started the mission.

> **Dan:** […] "It became a bit, I don't know, it became too much, I felt I was stressed you know. I begun to not understand and then I became stressed and it didn't work, as I see it."
> **Int**: "Could we talk about that? You say you are under stress."
> **Dan:** "I feel stressed and at the same time I feel that now I'm going to fix this, and I reread and read again and it doesn't really work I think, and I feel a certain stress the you ought to accomplish something, and I get cramped in some way. I think, things that I normally cope with, maybe, if I sit by myself, I cannot do now. It seems to me that I reread and reread and then I feel that I must make it. Compared to me sitting at home, it would be different I think. Sit at home in peace and quiet, you know."

After a while in the interview, we discuss the same theme. Dan relates to other students, and he tells how they also experience the same feelings as he has when it comes to programming. He expresses a feeling of low self-esteem and he compares himself with those students in the class who already know how to do[22].

> **Dan:** […] "You almost believe that… yourself, you have the image that you are the worst in some way, see. I've also heard that from others that are not so good at programming, they have that image too, you know."

[22]All names in the quotes are fakes and they are in italics.

> **Int**: “How can we change that here at school?”
> **Dan:** “I’d say like this… we start at so many different levels. […] Unconsciously, we compare ourselves with the good ones. You should compare only with yourself. That’s what I say, and that’s what *Anna* says, but you still do it. You watch them, say, *Charlie* and another chap. When you run the project that they did so very well, then you get such a bad image of yourself, you know. And about tackling the problem, I don’t know if it is the pupils or the teacher. Perhaps you could have a group and solve it, but I don’t know. Many that haven’t programmed before have that image of themselves. Because it‘s not just *Anna* and I, there are many others. […]. They think they are so bad, but I don’t think they are that bad. If only they will spend more hours, they’ll make it too. That’s what it’s all about, isn’t it? Some pick it up after a couple of hours, others may need some extra hours, you know. But they have such a poor… They don’t think they’ll make it. And that’s a drawback, see. And I don’t know … It is the poor self-esteem that it all comes to in the end, I guess. And mentally manage to struggle on and try not to give a damn about those who maybe have programmed for five, six years. And you have taken one course in programming yourself. You can’t compare yourself with them, but unconsciously you do. Even though I’m a bit older than him, I do it anyway, see.”

Later, Dan makes an interesting remark about those he regards to be clever in programming, namely, that they are not among the top students in mathematics, and he seems to use this as a comfort to some extent. He describes how he managed to succeed with mathematics by hard methodical work and by participating in the practical tutorial lessons.

> **Dan:** “No, I did well with the maths, but I went there anyway. I was not so confident in the beginning but I managed with the maths. Absolutely no problems, but I went there. I absolutely didn’t feel that way when I went there, and neither did the others. Because I saw that we, the ones that went there taking it seriously, were the ones that did well in maths. At the same time, I notice that many of those who are clever in programming have not settled the maths. I’m thoughtful about that. I see those who are very clever and still they have still not finished the maths. I can’t see why, but perhaps you… Some programmers are very clever at programming – particularly programming – but then perhaps, as I see it, they are not so social with others. But they are incredibly clever with computers. On the other hand perhaps, but that’s varies from one individual to another, but as in my opinion perhaps are not so…”
> **Int**: “Perhaps there’s something in that.”
> **Dan:** “But I’m surprised that they don’t manage the maths of all things. I thought they were superb in every subject.”

Could it be, that the cause of the advantage that Dan experienced in his clever fellow students, was in fact their private interest in computers and programming and their background? Moreover, does their being hobbyists and autodidacts automatically imply they are better students? Personally, I ask my self to which extent the “clever” student group decides the culture in the classroom, and if it has an effect on the level, content and objective of the courses – in worst case at the expense of the possibility for the novice programmer to succeed.

Some individuals described ways to understand the phenomena that covered all of the qualitatively different categories in the collectively obtained outcome space. However, some persons only gave a limited view of the phenomenon. For a teacher who teaches programming, it could both be interesting and valuable to have the following example as something to think about. In spite of the interviewer's attempts to stimulate for a variation in the ways of experiencing interfaces, Alf persists with his conception that interfaces are only used as "to-do lists," and he finishes off by saying, you never care about them after they have been implemented:

> **Int**: "But if you may call it the client side, that is to say, the one that uses a queue for instance, then it uses an interface instead of using the concrete class?"
> **Alf**: "Mm, you mean the one that… has…"
> **Int**: "Well you know the code that tests or uses a queue in someway. Then it uses an interface type to get at the queue, or?"
> **Alf**: "Yes, or one of these priority, well, but that's you know, interface, that you never have to care about. That's kind of…, you implement it and then you don't care about it anymore."

Perhaps Alf established his understanding during a course where the teachers handed out interfaces to the students in order to define what operations they should implement. While the students really experienced that it was meaningful and useful, this way to understand made a deep, persistent, impact on Alf, who did not have much previous experiences of programming compared to many of his fellow students. A certain motivated way to use the concept established an understanding of what the concept was, and it all made sense:

> **Int**: "In 'algorithms and data structures', how far have you reached in that course?"
> **Alf**: "Eh, until, let's see, eh, what have we…? We have done this about linked lists, and now we are doing a linked list again, but not the one that we should do on our own… Gosh what is it? We are implementing a queue for, well, anything… for queues, for heaps."
> **Int**: "Do you use interfaces then, somewhere?"
> **Alf**: "Yes, everywhere, all the time! I really think it is terrific, I do, but it's just that there are so many of them."

Dan's story tells us that we should be aware that what seems to be the "dominating culture" in the class might not be representative for all of students. He gives evidence for the gap between the "beginners" and the students who already "know" programming. There is an obvious risk that teachers adapt their level only to the “clever” students, and forget about the others.

Alf’s story indicates that his understanding came from a situation where the teachers probably did not intend to teach about interfaces. Their intention was to make sure that all the students knew what to do, and what method names they should use. This shows the strength of learning in situated contexts; how our learning often comes as a side effect of doing. However, it also reflects the potential “danger” of using concepts in a one-sided way; once we see a “meaning”, it takes an effort to change our understanding. Obviously, Alf did not change his mind about interfaces during the experiment.

10.6 Discussion on the students’ approaches

The original intention was to study the students’ approaches though an analysis of the video recordings and the recorded dialogues with the “curious colleague”. I have not yet accomplished that goal and the reason for this is primarily a lack of time to establish a proper theoretical framework for this kind of analysis, witch is something I want to get deeper into in my future work. I was also uncomfortable with the type of data that the recordings contained and I was uncertain how to approach it. This data is a set of two-hour sequences of the students’ actions as they appeared on the computer screen and it does not reveal the students’ *thoughts, during* their work. To get to that aspect, I considered using a method called *stimulated recall* (Bloom, 1953; Haglund, 2003), where I could ask the participants to view the video recordings as a stimulus for their memory, and discuss their thoughts about how they approached the task and how they thought in particular situations. Another idea was to let students solve the same problem in pairs, and record their discussion as they did their job. The recordings from these discussions would be a very rich data source for a further analysis. However, I decided to abandon these ideas in this study, due to the limited time and the estimated effort to accomplish it. Instead, I decided to use the data I already had, and to see what results we could get from that.

What I did was a thorough analysis of the transcripts, and I tried to find all statements that concerned the approach and the process. Then I investigated the students’ stored data files and searched for evidence that confirmed what they actually had been doing, such as which files they had edited and compiled, and what code they had written, et cetera. I compiled the results into Table 4 and further summarized it as an overview in Table 5 (see Chapter 8.1). These results give a rich description of what the students did and what problems they encountered. However, they do not tell us much about the students’ intentions, why they acted in certain ways, and how they thought.

I wanted to describe typical approaches that the students used, using the results and my impressions from what I saw during the experiment and in the interviews, and I suggested the three categories: “Hands off,” “Waterfall,” and “Prototype” (see Chapter 8.2). My intention was *not* to reduce and simplify the complex reality into some general model. I wanted to point at different behaviours that I could observe in the material that are critical for the ability to handle the type of problem that the students struggled with during their work.

There were a number of reasons for why I could not use a phenomenographic analysis to find out how the students experienced their approaches, in this study. First, the interviews did not primarily address the students’ thoughts about how they approached the problem, which is an essential requirement for this kind of analysis. Secondly, the posed question does not address experiences of a limited phenomenon to which the students relate, rather, it asks for the process, a chain of actions. Thirdly, it is hard to identify and assess logical relations between different patterns of behaviour, especially as we did not explicitly ask the students to explain their way to approach the problem, in the interviews.

However, it *is* possible to use a phenomenographic approach to study how students solve programming problems. Taking this perspective, Shirley Booth investigated how students learned to program and she followed a group of students during a programming course in SML and interviewed the fourteen students six times (Booth, 1992). In two of these interviews, the students’ approach to writing programs was an explicit topic. She came up with the conclusion that, within the present setting, there were four qualitatively distinct approaches to (learn how to) program (see Table 8).

Table 8. The four approaches to programming identified in the study conducted by Shirley Booth (1992, p. 207).

Expedient approach, in which focus is on producing a complete program from the outset by making use of an existing program or by adapting some known program to the demands of the problem.
Constructual approach, in which focus in on recognizing details of the problem in terms of features of the programming language – constructs, functions and keywords – which can be used to build a program.
Operational approach, in which focus is on writing a program based on an interpretation of the problem within the domain of programming; the problem is considered from the point of view of what operations the program has to perform.
Structural approach, in which focus is on writing a program based on an interpretation of the problem within its own domain; the structure of the problem is considered and on that basis a program is devised.

Booth divides the approaches in pairs and groups them in two dimensions. The first dimension grades the character of meaning. The approaches *operational* and *structural* aims towards understanding and interpretation, whilst *expedient* and *constructual* are more devoted to an opportunistic attitude. The other dimension spans between focus on the program and focus on the problem, and hence, the approaches *operational* and *expedient* focus on the program code, whilst *structural* and *constructual* rather focus on the underlying problem.

It is tempting to use the term *expedient approach* for the strategy that most of the students in our study used; once they had understood that they were supposed to write a class of their own in a new file, they quickly copied a similar file to build on. However, this would be to misinterpret Booths intentions. We need to understand what she means with *complete program* and what environment she refers to. In her case, the students used a functional (declarative) programming language, and the programming assignments (problems) that her students worked with were, clearly defined and rather mathematical in their nature, consisting of recursive problems. Booth means that both the expedient and the constructual approaches were *opportunistic* approaches to solve the whole problem. My interpretation of what Booth means is that the students start from a construct in the programming language, or from a copied program text, instead of analysing the problem.

In our case, the students who used the approach "prototype" copied the files as a conscious and intentional strategy in order to get things right, and save time and work, and hence *constructual approach* is a better classification for that particular strategy. However, there are examples of students in our study who were searching for something to copy, but as they did not fully understand the purpose and the wholeness, as in Fia's case, they could not find out what to copy, which relates closer to an *expedient approach*, using Booth's terminology.

The classification of approaches as being advanced, appropriate or shallow, et cetera, must be seen in relation to the situation where the approach was taken. In contrast to Booth's study, our students used Java, which is an object-oriented and imperative programming language. Their main problems was to understand the structure and the principle of a particular software system and how to fit in pieces of their own code (programming in the large) in a limited time, rather than to understand the underlying "problem" that the program should solve.

This difference makes it difficult to compare the outcomes of these studies. The "prototype" approach that we identified was an efficient way to solve the task and it certainly required a good understanding of the structure of the software, and we should therefore not regard it as an "opportunistic" approach.

The conclusion of this discussion is that we can study approaches to programming in many ways, and that different types of programming languages and problem types perhaps require different approaches. There is much more research to do here and for future work I have suggested two methods to stimulate and capture students' thoughts and reflections: stimulated recall and programming in pairs.

11 Implications for teaching and learning

This chapter deals with the “so what” aspect of this study, why we should care about outcome spaces, and how can we utilize the results and experiences of this work in teaching.

Who are the students, what do they already know and what do they want to achieve? What do we want to achieve with our teaching, what learning do we strive for, which educational aims do we and our “customers” have, and besides the students, who are our “customers”?

These questions concerning the relation between student, teachers, the subject, and the outside society, are not easy to answer and are not free from contradictions. In the academic tradition, we want to create conditions for free, associative, and critical thinking and learning. At the same time, we need to educate novices into persons who are well suited for a future professional career. Tight time schedules and economical resources delimit the educations, and at some point, there must be a compromise made. What we want our students to achieve is a basic competence with a potential for a variety of professions.

11.1 Creating connections to realistic situations

In the following discussion, I assume that we strive for an education where students, teachers, and the industry, all have an interest in the students becoming well prepared for a professional career in the IT business. This preparedness includes a good understanding of object-orientation, in a wider perspective than what it takes to pass in a beginner’s course in programming. An education that targets people who want to work with system development and programming, or perhaps with administration of such activities as a manager, needs to give the students profound insights in software development (software engineering).

I suggest that teachers should consider these aspects and think more about how we can establish and strengthen approaches to programming that help the students to widen their perspectives from the “here and now,” towards the outer world and the professional role and conditions, involving studies of communities, open source and APIs, companies, endurance, modifiability, efficiency and economy, et cetera.

For a student who wants to be a software developer in the future, every new experience of working with software will contribute to learning, skills and competence. Every new situation gives incentives for the individual to widen perspectives, reconsider previous experiences, and create learnings that she can add to her knowledge bank. In my experience from doing the interviews, most of the participants showed a constructive attitude to the situation during the experiment and most of them said that they had learned something from participating.

A hypothesis of mine is that some mechanisms of the object-orientated programming languages are easy to understand if we can explain them in a situated context, where the advantages they involve appear as natural and are well motivated. The experiment has shown that it is possible to let students work with large and advanced software, that they can get into it and achieve tasks in a limited timeframe, and learn things. Why not let them spend more time as *software engineering apprentices* (Dalbey, 1998) and do the same kind of things as the professionals do, and actually elaborate on authentic software from the industry, or other communities?

However, I believe an explorative learning using the “real thing” requires good basic skills and certain self-confidence. Not all students have the same prerequisites, and as the mission in our experiment required the students to solve the demanding problem independently, it turned out that not everyone succeeded. We, who are interested in similar settings, as a method for learning, must carefully ensure that the students have a constructive attitude, and that they have the required basic knowledge. Making this the wrong way could discourage the students and affect their self-esteem negatively. A way to neutralize this could be to let the students work in pairs.

My conclusion is that we should discuss these matters with the local companies, which sincerely want us to produce highly skilled software developers, and try to find good examples of authentic software that we can use in educational settings. The discussions should lead to a definition of important learning outcomes that both industry and academia share, and a set of tasks and exercises that would be encouraging and instructive for the students. In the best of worlds, representatives from the companies could give guest lectures and tell the real story about the software that the students will work with.

11.2 Opening possibilities to discern interfaces

How can it be that certain ways to understand sometimes tend to cement in the learner, and how can we change these ways to see?

Beginner’s courses often take a start focusing on explicit implementations of some codes, and then by doing many examples, the learner should gain a number of wisdoms on design principles. However, if the learner already considers himself as knowing how to program, new fancy principles might not be motivating to the “stubborn” learner, as it often is possible to just keep doing it the same way as before. For instance, I have met students at advanced level, who never realized that they actually could use the Java interface as a data type, and thereby use it to declare variables.

How can we provoke these students to reconsider their point of view? Continuous assessment and feedback naturally, but it is also important to consider how we introduce, motivate and discuss new concepts, and how we construct exercises. According to Ference Marton, the learner can discern something only if the learner can contrast it against its background, and we can help the learner by providing variations in the background, or variations of the viewing angle.

Marton och Tsui (2004) tell us that we should not only focus on what learners learn. We must also pay attention to in which ways they learn and we must be aware of the many different ways of seeing things that can be possible. We must consider how learners are able to discern parts from wholes, and how they can understand the whole. They claim that:

> […] variation enables the learners to experience the features that are critical for a particular learning as well as for the development of certain capabilities. (Marton & Tsui, 2004, p. 15).

They argue that it is necessary to consider variations in different learning situations and analyse what varies and what is held invariant. This would give information about what is possible to learn, which they call the space of learning. From empirical studies, they have identified four critical patterns of variation, which they describe in detail. In Marton and Pang (2006), these patterns are described in a more formal and condensed way:

Patterns of Variation

1. *Contrast*: A quality X cannot be discerned without the simultaneous experience of a mutually exclusive quality ~X.
2. *Separation*: A dimension of variation, which can take on different values, cannot be discerned without other dimensions of variation being invariant or varying at a different rate.
3. *Generalization*: A certain value, X_i in one of the dimensions of variation X cannot be discerned from other values in other dimensions of the variation unless X_i remains invariant while the other dimensions vary.
4. *Fusion*: The simultaneity of two dimensions of variation cannot be experienced without experiencing the two dimensions varying simultaneously (Marton & Pang, 2006, pp. 199-200).

How can we utilize these patterns when it comes to learning about the Java interface? Let us consider the patterns of variation combined with the empirical results from this study. Starting from the four qualitatively different categories of description, I will attempt to exemplify how we can design exercises and examples using the variation patterns applied to the dimensions that the categories open. Naturally, there are many possible approaches to this, and this example should be seen as an attempt to inspire other teachers' creativity and give an opportunity to reflect about how they can use similar results and ideas in their teaching.

The first category opens a dimension that concerns textual content, in which the interface is experienced as a text that constitutes a memo or a shopping list for the students to use in order to know what operations to implement. On this level, the *contrast* pattern is feasible to use. Let the students experience the effort of writing up a class from an informal specification on the black board, then give them the specification as an interface file, and show them how they can use the text in it as a skeleton, through copy and paste or by changing the keyword "interface" to "class". Be explicit and let the students verify that the compiler will not accept code in the interface – only in the implementation. Concentrate on the textual aspect of specification and *separate* this from other aspects during this exercise. At the end of this part of the lesson, the students could be asked to write a specification of their own in form of an interface, which can be used later.

The second category describes the interface as a contract between an interface and a class (or programmers) that forces the class to implement the interface. The new dimension of understanding considers various agreements between two parties and possibilities to manifest, verify, and maintain them. We can use the *contrast* pattern to bring out the contract quality of interfaces. First, the students can be instructed to implement a class, following the specification in the interface they wrote previously, without the keyword *implements* (not signing the contract), and compile it. If they are lucky, it will compile without errors. The agreement is in this case only informal. Then they should "sign the contract" and let the compiler be the judge who determines if the class fulfils the agreement. If the students introduce changes in the specification, they will see the effect. The physical appearance of the interface, the content in the text file should be put in contrast to its synthesis with the class through the contract that is signed by the keyword *implements*. In this way, the content and text dimension is held constant (*separation pattern*) and the focus would be set on the implementation of this precompiled interface and the keyword *implements*. The compiler will tell which methods that remain to be implemented. Another variation is to vary the number of methods in the class and see that it can have an unlimited number of methods as long as it implements the ones specified by the interface that is held constant (*generalization pattern*).

An alternative is to let two students make an agreement about the specification of a class. One of them implements the class and the other student makes a client class that uses the features of the specified class, and let them try to run their joint “program”. Before the client can even be compiled, the client designer has to wait until the specified class are implemented and compiled. If the client designer gets compiler errors that is caused by misinterpretations of the agreement, they have to decide who made the errors and correct them. In a second attempt, the students should put their agreement in an interface which should be seen as a formal specification. Now we can use the *contrast* pattern and let the students do the same thing using an interface that manifests their specification. The client uses the interface as a data type for the reference to the object and the server uses the interface as a contract. The effect is that both can work in parallel and compile their code.

The third category expresses a way of seeing the interface as a data type from which it is possible to create reference variables. The dimension of understanding regards the reference type’s relations to interfaces, classes, objects, and reference variables. Let the students create such variables and let them try to create an object. First using the interface, then using a class that has the proper methods but did not sign the contract, and last using the class that actually implemented the interface (*contrast pattern*). They will discover that they can create objects only from classes and never from interfaces. Moreover, that the interface reference only can refer to objects that explicitly implements the interface. The students will see that if the reference variable is of interface type, they only can call the methods described by the interface regardless of which object types it refers to (*generalisation pattern*). However if the reference variable is of class type, all of the operations can be called (*contrast pattern*).

In the most advanced category, the interface represents an open connection towards *any* object that implements the interface. The dimension is about exchangeability and polymorphism. Naturally, we could use the *contrast pattern* and show examples on what is polymorphic and what is not. In addition, we can use the *generalization pattern* by keeping the polymorphic quality of interfaces constant and vary the other dimensions. Let the students use objects from different classes that share the same interface; objects that their friends have written, objects from last year students, and the teacher’s version. Then pass them to a method using method parameters declared as interface type. Such a method can receive and handle any implementing object and it will lead towards a deeper understanding of polymorphic behaviour, especially if the code calls the objects’ methods and they explicitly give some kind of evidence for their distinctive character. Here the reference variable is held invariant while the implementation varies. This variation shows the possibility to delimit the services of objects, and it will reveal the polymorphic aspect. It is important to show aspects of dependency between parts of a program and what the effects are when the dependency is reduced.

One important feature of interfaces is that they constitute a barrier for the compiler. Changes in the implementing side (the server) can be done without having to recompile the client side. In order to show these aspects, the applications must have a critical mass. Otherwise, the discussion will have no meaning to the students.

One aspect of interfaces we could not find in the data was the possibility for a class to implement multiple interfaces. This means that the object fulfils several specifications that could be overlapping or exclusive. This could apply for a *product* object that is handled by several clients in a program. A client might not be allowed to call all of the operations of the product, and not the same operations as the other clients. Rather, they should only have a limited access. The *producer* might only call the operation *makeProduct*, the *salesman* might only call the operation *setPrice*, and the *user* should only call the operation *useProduct*. This can be achieved by using different interfaces that are handed out to the corresponding clients. The implementing class for the product implements all of the interfaces. At the same time, there can be many classes that implement different products but they use the same interfaces. This is a situation where there is a variation in two dimensions at the same time. One object can be referenced by different interface variables and at the same time, the object's concrete type can vary though the many possible implementations. This is an example of the variation pattern *fusion*, and I suggest that this kind of exercises should not be introduced before the students have a rich understanding of the different aspects as they appear while the other aspects are held invariant. After that, they are prepared to understand the simultaneous variations.

Marton and Tsui (2004) describe learning studies and lesson studies, in which teachers iteratively plan their lessons together, and in that way improve their way of conducting teaching according to experiences from accomplished lessons. The results from studies, such as the one you are reading right now, could widen the intellectual horizon and give important inputs to the discussion. The categories of description combined with the teachers experiences can reveal ways of understanding, and ways of how to discern, the different dimensions involved in the learning process.

11.3 Awareness of the industrial history and software engineering

The concept "interface" is a phenomenon full of nuances. Certainly, we can conceive it concretely, but we can also understand it in considerably more advanced and abstract ways. Discussions of how we could utilize interfaces would be an example of "object-oriented programming philosophy," and we should consider if, and when, we should introduce such a philosophy discussion in our courses. We can discuss and understand the principle to "program to an interface" by the industry's needs and long experience of standardization and control.

The industrial history witnesses of several crises and revolutions that in one way or another relate to control (Beniger, 1985). During the end of the twentieth century, the use of computers accelerated at a raging speed, and this accentuated the need for a control over the process of software development. The object-oriented paradigm took one leap towards a better way to structure, standardize, and communicate software. An interface in Java represents standardization in mini format, and it provides the ability to separate components from each other and make them as independent as possible.

In a historic perspective, this can be compared with how Henry Ford turned the automobile industry from its craftsmanlike production into an industrial production, where the knowledge that previously was a trade secret was systematically distilled, analysed and then built-in to artefacts and the entire production machinery. The students, who in the beginning of their education take a personal and craftsmanlike approach to programming, quite naturally, need an understanding of the consequences the industry (or programming community) would suffer if we do not produce software in a professional (standardized) manner. We cannot allow software to be personal secrets that only the individual programmer can understand and explain. We must make sure that we do not teach programming in a “pre-Ford” manner. Java’s interface is one of the artefacts we can use to shed light upon and emphasize the connection between system analysis and design on one hand, and programming on the other.

“Design patterns” (Gamma et al., 1995) are named, “smart” standard solutions of design problems that often occur in software development processes. I suggest that we involve them early in the courses, as they can be a good basis for discussions and an intellectual challenge for both students and teachers, and we need more of discussions in programming education. An example of a basic pattern is the “Bridge Design Pattern”, which describes the technique to use an interface between a client and a server, so often referred to in this text.

11.4 The voice of the researcher and the teacher

The work with this study has affected me in many ways. The interviews with the students were worthwhile, and I wish that all teachers would get a chance to discuss with their students in a similar way. Now, I am more aware of the variation within and between student groups, and the different ways students consider and approach programming. My view of the studied concepts has changed, and my view of how others experience them has changed, and naturally, it will affect how I will teach these topics in the future. For one thing, it is crucial to make a serious effort to help students understand the real benefits, the smartness, and the reasons for many of the concepts we teach, which we often take for granted that the students understand.

I have learnt much from what the students told me. Especially Cia, Dan and Joe gave me many moments of thought. Cia with her many clever ideas, who was hindered by her stubbornness when she never even wanted to run the software. Joe with his smart, elegant and complete solution of the problem, who, as it turned out, had experience from working professionally in the field. Finally, Dan gave me much to consider when he told me of his feeling of being outside the "programming culture".

Many Swedish universities are currently adapting their courses according to the "Bologna model". This revolutionary process transforms the course plans from being content-oriented towards a focus on the desired learning outcomes. John Biggs, the founder of the SOLO taxonomy, claims that the courses and all teaching activities should be designed using "constructive alignment" (Biggs, 2003). This means that we should align the assessments, examinations and all other activities, with the goals for the learning outcomes. It would then automatically follow that the students' activities would focus on what really matters. The key is hence to make the learning outcomes very clear and to adapt all the activities accordingly. For instance, in relation to the Java interface in this study, this means that the learning outcome for that particular concept must be made explicit, and that the students' activities and understanding are controlled by assessment and examination. In this case, it is important to formulate the expected learning outcomes on empirical results, and therefore studies that concern students' experiences are valuable.

12 Conclusions

In Chapter 1, I formulated a number of research questions, which represented the starting-point of this study. Throughout the work, I have tried to answer these questions, and in this chapter, I will summarize my conclusions.

12.1 Experiences and understandings of concepts

How do students experience and describe concepts that relate to programming in the large?

On a collective level, we have seen that students are aware of several different dimensions of understanding interfaces, plugins and the software system. The surface level of understanding relates interfaces with texts, plugins with small programs, and the system with a black box. These understandings do not have many connections with each other. However, the deep levels of understanding *are* integrated with each other, as they all are aspects of a holistic view of the software and related concepts that reflects a professional view on software development. This view involves the motives for using interfaces and plugins in the system.

12.1.1 Interface

The students, on collective level, describe the concept interface as:

- *To-do list*
- *Content declaration*
- *Data type and reference*
- *Open connection*

The results show that there is a depth in the interface concept and that there are several qualitatively different ways to understand it. At least three of the four categories of description contribute with something fruitful to the understanding of object-orientation, and we should take advantage of them in the computer science education.

There are aspects of interfaces that did not appear in the interviews. One example of this is Java's way to gain the advantages and avoid the disadvantages of multiple inheritance, and an explanation for the absence could be that the courses only covered it theoretically, and neither was it a salient feature of the experiment's software.

12.1.2 Plugin

The plugin concept is not treated explicitly in our courses (at the present), and thereby, the concept's meaning to the students probably originated from informal learning environments, or the students' own experiences of their mission assigned a meaning to the concept. The students described the plugin concept in two different ways:

- As *a small program*
- As *a part of a conceptual model*

There are points in common with the descriptions of the interface concept, and the way to describe plugin as "a part of a conceptual model" has strong relations to the description of interface as an "open connection." This is no surprise, as the plugins in the system utilize interfaces in the "open connection" sense and thereby interweaves with each other.

12.1.3 The System

The analysis of the students' descriptions of the software system has resulted in three qualitatively distinct categories:

- The system is described in terms of what an end-user can do, and what the system can do for her (the purpose expressed from the end-user's point of view)
- The system is described as constituted by collaborating parts, client, server, and database, which together can do the above.
- The system is described as dynamic, adaptable, extendable, and maintainable, due to the way the parts are structured.

The third description category combines the most advanced ways to experience the interface and the plugin concept to a holistic view of the system. The descriptions consider the design of the system and its consequences for the various involved roles associated with the system.

12.2 Successful strategies

Are there typical behaviours when students face problems of this type?

We have tried to find typical behaviours, and we saw that the students behaved in three typical ways. Some used what we call a "hands off" approach, in which the students tried to understand only by reading, and they did not start to program (some did not even run the software). Others used a "waterfall" approach. They read and started to sketch design and wrote some code. But they could never run their code. Those who used a "prototype" approach copied a plugin, inserted the copy in the system, they could see what happened, and could gradually modify their code.

Working with prototypes, and gradually developing them seems to be a successful and effective strategy in a task like the one in this study. The alternation between making some changes in the code on one hand, and executing it to see the effects, on the other hand, motivates, and stimulates for an understanding of the system. In addition, the system confirms that the students have understood it the right way. For most students, it was very important to try to run the system in order to "see" how it worked and get a better understanding of what they were supposed to do. Besides, a customer would probably appreciate to see an executable and visible prototype, at any time.

Trying to complete the mission by writing program code only, not even trying to compile the code, turned out to be an unsuccessful method. Although these students had come a long way in their coding, the presumed time-estimate for error corrections would be unreasonably high, due to the many compile errors they probably would get if they should compile everything in one big chunk. In this way, they neglected to use the compiler as a valuable resource in an early stage, as they only used it as a code translator in the final stage.

Naturally, yet another strategy was to ask for help, and in a real situation it is important to dare to ask for it, when it is legitimate. However, admitting that one cannot do it by oneself would imply a loss of prestige. To accentuate this feeling, it was only possible to get help at two times during the experiment, and only two persons used this possibility.

Are there connections between conceptual understanding and the practical abilities to program in the large?

One conclusion of this study is that concept comprehension connects to the practical skills that are required to be successful in programming. On an individual level, we can conclude that those who almost completed the mission also expressed a good understanding of interfaces, plugins, and could explain how the system really worked. The only student who actually completed the mission described the involved concepts in a way that reflects a comprehensive view. Those individuals, who only expressed their experience of interface as in the first category, did not manage to solve their task.

On the other hand, without having expressed the most advanced understanding, some students managed to solve much of the task anyway, thanks to a successful strategy. Without an effective strategy, they failed.

12.3 The outcome of the task

Are the students well prepared for working with extensive software, in other words, is the education relevant for the profession?

It is reasonable to conclude that some students were well prepared. Only one student managed to complete the mission to obtain a working solution, which was something of an achievement. However, this student had worked with programming before he started to study. Some of the students used an ineffective strategy and some did not give comprehensive descriptions of the concepts. Two students were stuck and seemed to have poor skills in Java programming and witnessed of low self-esteem, and poor skills.

Five out of eleven came up with a visual (executable) result, and they had started to implement the logical functionality. In a couple of cases, the students had started to write code, but they could not execute it.

I conclude that the students need more training in reading software, documentation, and in seeing practical use for theoretical concepts.

12.4 Implications for teaching and learning

If we can find any answers to the questions above, how can we use them in our teaching?

I agree with Schmolitzky (2004), who advocates that we should introduce interfaces early in programming courses because it is a powerful concept that enables teachers and learners to reflect on dependency, responsibility and flexibility without the machinery of inheritance and subtypes. These topics are central features of a professional perspective on software development and it is important to address them explicitly. Moreover, as Sicilia (2006) points out, we should describe design situations where the use of interfaces stands out as a motivated concept to use.

One conclusion from the experiment is that the students probably would benefit from more training in dealing with this kind of situations. The students liked to participate in the role-play, and were really engaged in the work. They thought that the experiment gave them a deeper understanding of software and involved concepts, such as using interfaces as bridges towards plugins. Therefore, using various tasks in large-scale software could be a stimulating and fruitful element in programming courses (see Dalbey, 1998).

I suggest that we explicitly can use the description categories and the dimensions they open for the theoretical part of teaching. For instance, we can ask the students to discuss and reflect on these ways to understand interfaces. In the practical parts we can design exercises that makes students discern the dimensions of understanding interfaces, by variations (Marton and Tsui, 2004) of how interfaces are used and not. Plugins could be a topic in a course that could be an interesting experience for students, especially if they design their own.

If we follow the hierarchic structure in our outcome spaces, we can see how focus shifts from being aimed at writing a certain code, via various semantic meanings in design-time and run-time, and finally aiming towards how the program's structure affects the surrounding world. It is this understanding of the wholeness we want to promote, and the experiment has shown us that if we provide students with a realistic and complex task, they will try to understand it, and in most cases, abstract concepts will get meaning in a situated context.

To make it a lasting and meaningful experience, I suggest that we cooperate with the industry to get ideas and perhaps even sample software from them. Concepts are easier to understand and remember if the students can relate them to situations where they appear as meaningful and efficient tools for writing great code.

12.5 Plans for future work

This study gave some answers concerning students' behaviours and approaches, but I think there is much more to learn about how students act when they solve *programming in the large* problems. It is also interesting how students use and reason about strategies, concepts and terminology in order to improve teaching in these regards.

In a continuing study, I would like to video tape students working in pairs, and follow up their work with stimulated recall interviews. It would be very interesting to integrate this technique for data collection in a course and follow how the students evolve during the course.

What is software and what does it mean to work with software? It would be interesting to study how students answer these questions in different study programmes and in different stages of their education. How do the descriptions change as the students are affected by their studies and the cultures they meet?

The third question I would like to investigate is what alumni students, at work, think were the most valuable lessons the learnt from their studies. In addition, what lessons they did not learn, what knowledge or skills they lacked when they started to work. Is that something we can improve in our education of professional software developers?

13 References

Abrandt Dahlgren, M. (2006). From senior student to novice worker: learning trajectories in political science, psychology and mechanical engineering. *Studies in Higher Education*. Vol. 31, No. 5, pp. 569-586.

Ben-Ari, M. (2004). Situated Learning in Computer Science Education. *Computer Science Education*. Vol. 14, No. 2, pp. 85-100.

Ben-David Kolikant, Y. (2004). Learning Concurrency as an Entry Point to the Community of Computer Science Practitioners. *Journal of Computers in Mathematics and Science Teaching*. 23(1), pp. 21-46.

Beniger, J. R. (1985). *The Control Revolution: Technical and Economic Origins of the Information Society*. Cambridge, MA: Harvard University Press.

Berglund, A. (2005). *Learning computer systems in a distributed project course*. Department of Information Technology. Uppsala, Sweden.

Berglund, A., Daniels, M., & Pears, A. (2006). Qualitative Research Projects in Computing Education Research: An Overview. In D., Tolhurst and Samuel Mann (Eds.) *Conferences in Research in Practise in Information Technology,* Vol. 52. Australian Computer Society, Inc.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning. The SOLO Taxonomy (Structure of the Observed Learning Outcome).* New York: Academic Press.

Biggs, J. B. (2003). *Teaching For Quality Learning At University. What the Student Does.* Milton Keynes, UK: Open University Press.

Bloom, B.S. (1953). Thought Processes in Lectures and Discussions. *Journal of General Education*. Vol. 7, No. 3, pp. 160-169.

Bloom, B. S. (Ed.) (1984). *Taxonomy of Educational Objectives. Book 1: Cognitive Domain*. The Classification of Educational Goals. New York: Longman.

Booth, S. (1992). *Learning to program: A phenomenogaphic perspective. Acta Universitatis Gothoburgensis 89:1992.* Göteborg, Sweden: Acta Universitatis Gothoburgensis.

Booth, S. (2001a). Learning Computer Science and Engineering in Context. *Computer Science Education*. Vol. 11, No. 3, pp. 169-188.

Booth, S. (2001b). *Learning to program as entering the datalogical culture: a phenomenographic exploration.* Unpublished paper presented at the 9th EARLI conference, Fribourg, Switzerland, August 2001.

Bowden, J. (2000). The nature of phenomenographic research. In J. Bowden and E. Walsh (Eds.) *Phenomenography.* Melbourne: RMIT University Press.

Bruce, C., Buckingham, L., Hynd, J., McMahon. C., Roggenkamp, M., & Stoodley, I. (2004). Ways of Experiencing the Act of Learning to Program: A Phenomenographic Study of Introductionary Programming Students at University. *Journal of Information Technology Education*. Vol. 3, pp. 143-160.

Budd, T. A. (2002). *An Introduction to Object-Oriented programming* (3rd ed.). Boston: Addison Wesley.

Clancy, M., Stasko, J., Guzdial, M., Fincher, S, & Dale, N. (2001). Models and areas for CS Education Research. *Computer Science Education*. Vol. 11, No. 4, pp. 323-341.

Cope, C. J. (2006). *Beneath the surface: The experience of learning about information systems*. Santa Rosa: Informing Science Press.

Dalbey, J. (1998). The Software Engineering Apprentice. *Computer Science Education*. Vol. 8, No. 1, pp. 16-26.

Denzin, N., & Lincoln, Y. S. (Eds.). (1994). *Handbook of qualitative research.* London: Sage Publications.

DeRemer, F., & Kron, H. (1975). Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software, Volume 10 Issue 6, pp. 114-121.* New York: ACM Press.

Eckerdal, A. (2006). *Novice Students´ Learning of Object-Oriented Programming*. Department of Information Technology. Uppsala University, Sweden.

Ekstedt, E. (1988). *Humankapital i brytningstid: kunskapsuppbyggnad och förnyelse för företag*. Stockholm: Allmänna förlaget, Publica.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns – Elements of Reusable Software*. Addison-Wesley.

Gunderman, R. E. (1988). A glimpse into a program maintenance. In G. Parikh (Ed.), *Techniques of program and system maintenance* (pp. 55-59). Wellesley, MA: QED Information Sciences Inc.

Hadjerrouit, S. (1999). A constructivist approach to object-oriented design and programming. *ACM SIGCSE Bulletin*, Vol. 30, No. 3, pp. 171 – 174.

Hadjerrouit, S. (2005). Constructivism as Guiding Philosophy for Software Engineering Education. *ACM SIGCSE Bulletin.* Vol. 37, No. 4, pp. 45-49.

Haglund, B. (2003). Stimulated Recall. Några anteckningar om en metod att generera data. *Pedagogisk Forskning i Sverige*. 8th Year, No. 3, pp. 145-157.

Hazzan, O. (2003). How Students Attempt to Reduce Abstraction in Learning of Mathematics and in the learning of Computer Science. *Computer Science Education*. Vol. 13, No. 2, pp. 95-122.

Holmboe, C., McIver, L., & Carlisle, G. (2001). Research agenda for Computer Science Education. In G. Kadoda (Ed.), *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group,* pp. 207-223. Bournemouth. UK.

Jaccheri, L., & Morasca, S. (2006). On the Importance of Dialogue with Industry about Software Engineering Education. In *Proceedings of the 2006 international workshop on Summit on software engineering education*, Shanghai, China, pp. 5-8. ACM Press.

Jaccheri, L. (2001). Software quality and software process improvement course based on interaction with the local software industry. *Computer Applications in Engineering Education*, Vol. 9, No. 4, pp. 265-272. John Wiley and Sons.

Kajko-Mattsson, M., et al. (2001). Developing CM^3: Maintainers' Education and Training at ABB. *Computer Science Education*. Vol. 12, No. 1-2, pp. 57-89.

Klein, H. K., & Meyers, M. D. (1999). A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems. *MIS Quarterly*, Vol. 23, No. 1, pp. 67-93.

Kölling, M. & Barnes, D. J. (2004). Enhancing Apprentice-Based Learning of Java. *ACM SIGCSE Bulletin.* Vol. 36, No. 1, pp. 286-290.

Lave, J., & Wenger, E. (1991). *Situated Learning: Legitimate peripheral participation.* Cambridge: Cambridge University Press.

Lethbridge, T. C. (1998). A Survey of the Relevance of Computer Science and Software Engineering Education. In *Proceedings of the 11th Conference of Soft-*

ware Education & Training. IEEE Computer Society Press, February 1998, pp. 44-55.

Lincoln Y. S., & Guba E. G. (1985). *Naturalistic inquiry*. Newbury Park, CA: Sage.

Marton, F., and Säljö, R. (1976a). On Qualitative Differences in Learning – 1: Outcome and Process. *British Journal of Educational Psychology.* Vol. 46, pp. 4-11. British Psychological Society.

Marton, F., and Säljö, R. (1976b). On Qualitative Differences in Learning – 2: Outcome as a function of the learner's conception of the task. *British Journal of Educational Psychology.* Vol. 46, pp. 115-127. British Psychological Society.

Marton, F., Hounsell, D., & Entwistle, N. (Eds.) (1986a). *Hur vi lär.* Stockholm: Rabén & Sjögren.

Marton, F. (1986b). Phenomenography – a research approach to investigating different understandings of reality. *Journal of Thought.* Vol. 21, No. 3, pp. 28-49.

Marton, F. (2000). The structure of awareness. In J. Bowden and E. Walsh (Eds.) *Phenomenography.* Melbourne: RMIT University Press.

Marton, F., & Pang, M. F. (2006). On Some Necessary Conditions of Learning. *The Journal of Learning Sciences.* Vol. 15, No. 2, pp. 193-220. Mahwah, New Jersey: Lawrence Erlbaum Associates, Inc.

Marton, F., & Booth, S. (1997). *Learning and Awareness*. Mahwah, New Jersey: Lawrence Erlbaum Associates, Inc.

Marton, F., & Tsui, A. B. M. (2004). *Classroom discourse and the Space of Learning*. Mahwah, New Jersey: Lawrence Erlbaum Associates, Inc.

Muhr, T. (2004). User's Manual for ATLAS.ti 5.0, 2nd Edition. Berlin: Scientific Software Development.

Mulholland, J., & Wallace, J. (2003). Strength, Sharing and Service: restorying and the legitimation of research texts. *British Educational Research Journal.* Vol. 29, No. 1, pp. 5-23.

Parnas, D. L. (1998). Software Engineering programmes are not computer science programmes. *Annals of Software Engineering 6*. pp.19-37.

Pears, A., & Daniels, M. (2003). Structuring CSEd Research Studies: Connecting the Pieces. ACM Conference on Innovation and Technology into Computer Science Education, Thessaloniki, Greece, June 2003.

Richardson, J. T. E. (1999). The Concepts and Methods of Phenomenographic Research. Review of Educational Research. Vol. 69, No. 1, pp. 53-82.

Robins, A., Rountree, J., & Rountree, N. (2003) Learning and Teaching Programming: A Review and Discussion. *Computer Science Education.* Vol. 13, No. 2, pp. 137-172.

Schmolitzky, A. (2004). Educators' symposium: "Objects first, interfaces next" or interfaces before inheritance. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications,* pp. 64-67. OOPSLA 2004. ACM Press.

Sfard, A. (1991). On the dual nature of mathematical conceptions: Reflections on processes and objects as different sides of the same coin. *Educational Studies in Mathematics*. Vol. 22, No. 1, pp. 1-36.

Sicilia, M-Á. (2006). Strategies for Teaching Object-Oriented Concepts with Java. *Computer Science Education*. Vol. 16, No. 1, pp. 1-18.

Skrien, D. (2003). Learning Appreciation for Design Patterns by Doing it the Hard Way First. *Computer Science Education*. Vol. 13, No. 4, pp. 305-313.

Steimann, F., Siberski, W., & Kühne, T. (2003). Programming techniques: Towards the systematic use of interfaces in JAVA programming, In *Proceedings of the 2nd international conference on Principles and practice of programming in Java PPPJ '03*. Computer Science Press, Inc.

Stroustrup, B. (1997). *The C++ Programming Language* (3rd ed.). Reading, Massachussets: Addison Wesley.

Tvedt, J. D., Tesoriero, R., & Gary, K. A. (2002). The Software Factory: An Undergraduate Computer Science Curriculum. *Computer Science Education*. Vol. 12, No. 1-2, pp. 91-117.

Vaughn, Jr, R. B. (2001). Teaching Industrial Practices in an Undergraduate Software Engineering Course. *Computer Science Education*. Vol. 11, No. 1, pp. 21-32.

Walsh, E. (2000). Phenomenographic analysis of interview transcripts. In J. Bowden and E. Walsh (Eds.) *Phenomenography*. Melbourne: RMIT University Press.

Åkerlind, G. S. (2005). Variation and commonality in phenomenographic research methods. Higher Education Research & Development. Vol. 24, No 4, pp. 321-334.

14 Appendix A

14.1 Word list

Abstraction	Abstraction is the fundamental concept within object-orientation. A software object can be an abstraction of some real phenomenon, e.g., an invoice, and its corresponding class is a meta abstraction that describes the object. The class is a model of what characterizes the invoices in the system that is developed. A class can also model less tangible phenomena, such as equations, or internal parts of the machinery like controls or object factories. Abstraction can also involve levels of abstraction in the code, starting from general and easy to use classes, and ending in specialized, concrete classes, and similarly for calls to operations. One strives to write code in abstract manner, as it gets less dependent of the circumstances in the specific case. For instance, in a fruit store software, the class Fruit can be designed at abstract level, and the parts of the program that only use the general fruit aspects should only be aware of this meta abstraction of fruits, and its code will be short and robust. Other parts of the program, which need to use the specific fruits, use subtypes of the Fruit class, e.g., Banana and Orange. One of the advantages is the possibility to add new fruits, such as Apple and Kiwi, with minimal or no changes in the program. See also polymorphism and dynamic binding.
Attribute	An object has various properties with values, e.g., speed, height, width, temper. Properties that are visible from outside the object is called attributes (or properties), and are internally implemented in the class as variables and usually have designated access methods.
Class	A class is a definition of a family of objects. Its written program text defines a list of the attributes (data) and methods (program code that does something) needed for the objects of this type. Normally, a programmer defines a class statically in design-time.
Client	We use the term *client* to denote the user perspective or the outside view of something. The client is someone or something that uses something that a server provides. The client can be another class, another part of the system, another programmer, and so forth.
Compile	To compile something refers, in our context, to the process that takes place when a compiler program analyses the source code that a programmer has created in text files. If the source code is correct and all the resources it refers to are valid, the compiler translates it in a form that the computer can execute, and the result is stored in a binary file. In Java, the Java Virtual Machine (JVM) executes the code virtually.
Compile-time	This is the space of time, the specific conditions, and circumstances, which are associated with the compilation process. See dynamic and static binding.
Design-time	This is the space of time, the conditions, and circumstances, which are

	associated with the sketching and planning of a program or system, as the solution for some need. Usually the result of this work is part of the software or documentation. Sometime design-time and implementation-time are concurrent, as in the case of extreme programming.
Dynamic binding	Dynamic binding, or late binding, means that the logical bindings between elements of the program (client and provider) are undefined until their actual use in run-time, i.e., when a specific method is called. This implies that an object handle (a reference variable in Java) can be bound to an object without being bound to its specific type. Nevertheless, there is a static binding between the abstract types of the handle and the object, which implies that the compiler verifies their compatibility. For example, a Vehicle handle can refer to all objects that descend from that Vehicle type by inheritance. Via the handle, we can call only the specified methods in the super type, but the different types of descending objects can implement of these methods differently, provided they have the same signature. Hence, the run-time situation decides which specific method is bound to the method call. Normally in Java, all method calls use dynamical-binding.
Entity object	An entity object represents something “real” and includes some form of data. It is commonly associated with a row in a database table. Entity objects have a passive role, in contrast to the control objects that defines the program flow in a program.
Implement	To implement something is to make it happen, to pursue the programming job, often according to a plan or algorithm.
Implementation	In this context, implementation means to write code that defines behaviour and representation, i.e., program code for classes, coder for their operations and declaration of variables, et cetera.
Implementation-time	In my definition, it is the time space, or stage, when the programmer explicitly implements the design in form of program code. However, the activities could be scattered, depending on the way people work.
Inheritance	Inheritance in Java means that a class can include all of the declarations and contents in another class, simply by saying that it wants to “extend” the other class. Naturally, the extending class can define an unlimited number of own methods and attributes. It can also choose to override on or more methods that descend from the inherited class. This would conceal the old version and replace it with the new version. In other words, that is how to utilize polymorphism. The extended class (the original) is called a super class, and the extending class is called a sub class. Noteworthy is, that a reference variable of super type is compatible with both types, while a reference variable of sub type only would be compatible to the subclass or its sub classes.
Interface	Interface can have many meanings, but in this context, it generally refers to the accessible methods in an object that an extern client can call. Java puts the concept in concrete form, as it is a construct of its own in the language. Java suggests the keyword interface to define an incomplete type that specifies a set of operations. All classes that implement this type by explicit declaration and implementation surely have the specified methods, and are therefore to regard as implementations of the type. See polymorphism.
Method	In the object-oriented paradigm, the word “method” is synonymous with the words operation, function, and procedure, stemming from other paradigms. A public method is a sub-program that a client can issue, or a private method that only methods in the same class can call. Methods are specified by interfaces or classes, and are implemented only in classes.
Object	An object is an encapsulation of data and functionality (methods) that exists in run-time only. The object is a run-time representation for a

	"thing" that might have a correspondence to the real world, or be an internal abstraction. The object is an instance of its class, which defines all of its structure, but not the data contents. The objects do all of the action in an executing program, as it is their calling methods between themselves, that actually is what "is alive" during run-time.
Polymorphism	The word polymorphism comes from the Greek words poly (several) and morph (shape), and it represents a very important form of abstraction in object-oriented languages. It means that a piece of program code that use general or abstract variables can handle various types of concrete object, without having to consider their specific implementations. This would imply that an unchanged line of code, can cause an unlimited variation of actions in run-time, due to what is on "the other end." In Java, this is utilized by inheritance of classes, or by implementation of interfaces. If the programmers use a super-type to declare their handle, they can use it to handle various versions of objects.
Reference	A reference in Java is the association to an object, and we often interprete this as the object's address in memory.
Reference variable	A reference variable is a variable that can store references to objects of a specific type. I Java, this is the only way to access and handle objects. The term handle is often used both for references and reference variables.
Reification	The term reification denotes the process of when something theoretical or abstract materializes. A novice probably understands the concept equation as something abstract and undefined, but the experienced mathematician has reified the concept into a mathematical object, something with a clear structure. Nevertheless, it is still an abstract concept of course.
Run-time	This is the space of time, the conditions, and circumstances, which are associated to the execution of a program. It comprises the notion of the objects' existence in the memory, how they are structured, what methods that are used, and in which order objects call them, et cetera.
Server	The term server denotes something that provides for a client. What it provides can be any kind of services or operations. As it is a flexible notion, the server can be a computer, but it could also be a running program, a piece of code, or an object.
Signature	A method's signature consists of the method's return type, its name, and its ordered set of declared parameters. It is the signature, which makes the particular method uniquely identifiable, together with its class scope. In Java, an interface is actually a set of signatures. All classes that implement the interface have at least a set of operations that matches the signatures specified by the interface.
Static binding	Static binding, or early binding, denotes the bindings between elements in a system that the compiler does, and unless the program is recompiled, the bindings remain unchanged. The more the occurrences of static bindings that exist between the parts of a system, the harder it gets to change, adapt, and modify the system at a alter stage. Certainly, a program that extensively utilizes static binding may execute faster, however it will not be robust to changes and new conditions.

15 Appendix B

15.1 About object-oriented programming

A central problem in programming and software development is that the dependencies between different logical parts in a system become too extensive and intertwined, and hence, changes in one part can lead to a need for compensational actions in many other parts of the system, having large costs as an unwanted effect. Moreover, when a part allows a misuse, not aligned with the designer's intentions, changes in it can lead to malfunctions in other parts. Such faulty often remain undiscovered until the system is tested. Naturally, it is possible to design systems in a way that avoids these problems, but the old imperative programming languages have few support mechanisms and structures to compensate for the problems, which demand a very thorough planning and individual discipline. The lack of clear structures and delimitations lead to programs that are too hard to grasp for uninitiated persons. These problems are always present, but in a historic point of view, something happened during the seventies. As the software industry expanded and the complexity of the software systems grew, and the demands on the productivity raised, the problems became so serious that it led to a crisis in the industry; the *software crisis*. The costs for development had become too high, and the time for delivery too long, which the customers experienced as having to pay all too much for a product that was already out of fashion.

The object-oriented paradigm arose as an evolutionary step and it was a natural consequence of the complexity problems, and experiences from earlier programming abstractions. The object-oriented languages, or anyway the popularity of them, came from a need to structure the software better than before. The language provides features that connect certain parts hard to each other. The programmers use these features when they *encapsulate* data and its associated operations into unities called objects. All data resides in objects, and the only way to manipulate an object's data is by using the object's own intrinsic operations, which means that it is only possible to manipulate data in a controlled manner. In addition, this reduces the semantic gap between the reality and its representation in the software, as the encapsulated objects can represent objects in the real world. This allows many more persons to understand the software, also non-technicians. It is possible to group objects that are naturally involved with each into sub systems, which follow the principle of *high cohesion*. Between the sub systems, we can obtain *low coupling* to minimize the degree of dependency, and in this way, it is easier to modify or exchange parts of the system. The reinforced, imposed structure makes it easier to reuse the code in other projects, especially using inheritance. The object-oriented view makes it possible to increase the abstraction level from program code and functions, to something that in a very powerful way can capture and describe concepts and processes.

15.1.1 The concept of an object

Objects are the things that an object-oriented program handles, and the program "is" all about objects asking each other to do things for them, or just holding other objects "by the hand". An object represents a model of something that can be inspired from the real world, such as a person, or something abstract, such as the Swedish "birth number" (comparable to NIN[23] and SSN[24]). As there are many persons out there, and many birth numbers, there would be several objects of the same type in the program. The objects' tasks are partly to store information, such as the sequence of digits in the birth number, but also to provide the operations that can manipulate the information. As for an example, there can be an operation to verify if the number is correct using a checksum algorithm, and an operation to decide whether the number belongs to a male or a female, and when he or she was born.

The object exists only when the program executes, and when it is born, using its class as a mould, it is assigned with a reserved area in the memory at its own disposal, to store its information. To get access to an object, and actually to keep it alive, you need a handle of some form that keeps track of the object, similar to a dog's leash. Through this handle it is possible to communicate to the object by calling some of its methods according to the following syntax: "*handle.operation()*." In strictly typed programming languages like Java, it is required that the handle, the reference, is compatible with the object, otherwise it cannot "hold" the object. Namely, the handle informs the compiler of which are the callable operations, not the object. Hence, the type of the handle must be the same as for the object, or be a subtype. This makes it impossible to issue calls to operations that does not exist in the object, and the compiler verifies it is true.

15.1.2 The concept of a class

A class is what defines a common type for a family of objects. In Java, the programmer puts the definition of a class in a text file according to a specific

[23] The British National Insurance Number
[24] The USA's Social Security Number

syntax, which involves its name, its need for information storage, and its implemented operations. The definition of the class actually creates a new data-type that the software can utilize to make handles and objects, and when a class is complete and tested, we can reuse it repeatedly. In this way, an abstract model of a real life phenomenon turns into a component we can use to construct new software.

In the object-oriented languages, there are mechanisms that enable us to create derived classes, or sub-classes. Derived types inherit their base configuration from an existing class, which means that a sub-class *declares* that all content in a super-class also is part of the sub-class. However, this does not literally copy the source code content into the derived class, as this is an entirely abstract mechanism. When the programmer continues to implement the derived type, he or she can choose to refine some of the inherited operations or take them all as they come. The derived class, modified or not, is still compatible with the inherited class, and the advantage is that it is possible to exchange components with new derived versions and use it exactly in the same way as before, which means that you do not have to make any adjustment in other parts of the system. In this manner, it is possible to isolate changes so that they give a minimal effect on existing code.

Well-designed software can handle heterogeneous object types. Programs designed for that purpose usually introduce base classes that represent the least common denominator for a whole hierarchy of classes belonging to the same group. Imagine that we are to develop a piece of software that handles a motor-vehicle register, which can handle many different types of vehicles. It is appropriate to create a base class *Vehicle* that has all the common features of vehicles. Now we can use this type when we implement the greater part of the system. Then, whenever the need occurs, it is possible to create new classes that inherit the base class, such as Automobile, Lorry, or Motorbike, and the system would accept them immediately. When a program is expected to handle different object types in a common manner, and therefore does not have to know about the explicit types of the objects, it is a good reason to introduce this polymorphic technique.

15.1.3 The concept of interface

The everyday meaning of interface is the features of something's connections to the rest of the world. Related to the world of computers, people probably associate the word with graphical user interfaces (GUI), or the parts of a program that communicate with users or other machines. We can also relate it to communication protocols and physical interface, such as the USB interface. However, when we enter the world of software and programming, we rather allude to the abstract links between different parts on the "inside" of computer programs, and hence, it is something that concerns programmers, not end-users. A software component's interface is an abstraction that alludes to the collection of operations that are available to others on "the outside." When programmers speak about an object's interface, they mean "*the public interface,*" which involves precisely the parts of the objects code that others are allowed to call. However, programmers have used the concept long time before the object-oriented languages entered the scene. Also in older languages, there are mechanisms that can separate the public and the internal parts in a module. A convenient way to keep the information about external module's accessible functions and variables is to put these declarations in special file that can be loaded by any module that needs to use the specified operations.

Authorities in literature, emphasize that we should separate our plans for the system from its implementation (which would imply that we actually have plans for our work…). We can separate the two terms *specification* and *implementation* by using the concept interface, in the sense that it is something that specifies which operations the code can offer, while the class constitutes the implementation that explicitly defines the operations. The part that wants to use the specified operations should never bind immediately to the supplier. Instead, it should always go through the interface. We should use this as a general guidance, and always "*program to an interface*" (Budd, 2002; Gamma et al., 1995, p.18).

The programming language Java has two principal syntactic units, the commonly known *class*, and the *interface*, which is a modern reification of the abstract interface concept, introduced as a new keyword in the language. The interface structures its code similarly to the class, but it does not have any variables or method bodies. A class implements and defines explicitly the objects' constitution, and it defines a data type. With the interface construct, we can separate the "what and how aspects" and only specify the signatures of a number of operations, that is to say *what* is included but not *how* to implement the operations; the public interface of an object. In addition, we can say that the interface enables a refined version of the polymorphism concept by its feature of inheritance of specification instead of inheritance of implementation.

This becomes even more interesting when we realize that the interface also defines a data type, and that is as with the class is possible to declare reference variables of this type. These variables can handle any object that has a compatible public interface, but having the proper set of operations is not enough. In addition, the class of the object in question must declare that it implements the interface. The class makes this declaration by the keyword *implements* in the class definition, and hence, the class has pledged itself to contain the operations that the interface specifies. In this way, we split up tight connections between objects and we insert the interface in between, as a "proxy" or "middleman", which conceals the "real" object behind the curtain, and this makes it possible to exchange the objects (see Figure 7). By using interfaces and implementations of them, we can now achieve polymorphism without code inheritance. It also leads to looser connections between the objects that we want to use polymorphic, as they do not need to be part of a hierarchy where all classes inherit from a common base class. In inheritance hierarchies, all classes normally are of “the same kind” as the code accumulates in the specialized classes. When we use interfaces, the implementing classes can be very different, because they all provide their own unique implementations, and what accumulate are the method prototypes, not code. We should mention that we could utilize this technique in other languages, as for instance with purely abstract classes in C++, but there is no support in the language in form of an artefact, we only have the mental metaphor.

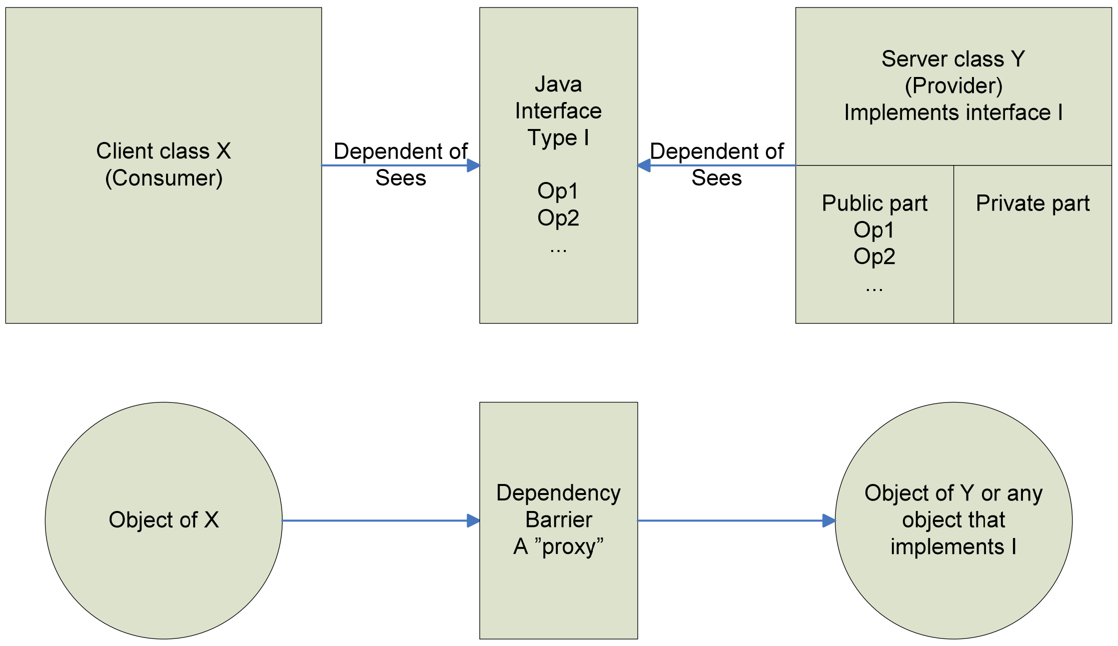


Figure 7. The Java interface encourages programmers to program towards specifications, instead of being dependent of specific implementations.

16 Appendix C

16.1 Interview questions and themes

The following is a compilation of the themes and questions that we planned for the interviews. We prepared ourselves for a variation in the individual interview's context and theme, and therefore we intentionally did not formulate the questions literally.

Theme – the mission

- Describe what you were doing when we stopped the experiment
- Describe what you focused in the beginning… and then…
- Describe how you experienced the entire situation
- Describe your mission
- Describe how the system works and how it is built
- Describe the software's structure, the packages
- How did you experience my absence, and presence?
- About the documentation of the system
- About the documentation of the code

Theme – interface

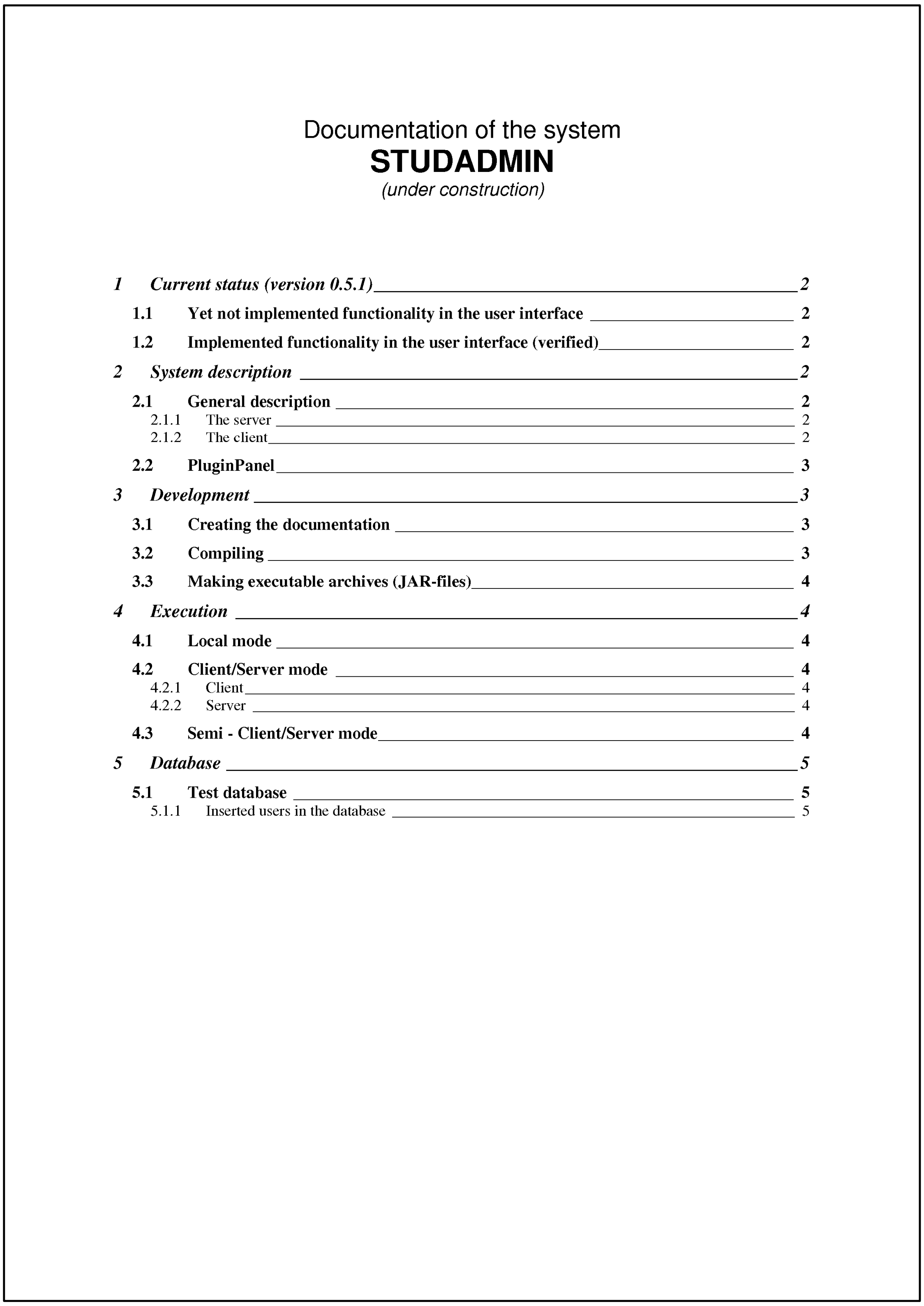
- Describe interface
- Describe the ides of using interfaces
- Describe where you use interfaces
- Describe plugin

Theme – learning and problem solving

- What have you learnt from this experience?
- Where did you get your problem solving skills?
- Describe what you would like to learn more about
- Do you like to solve problems – do you approve to take a leap in the dark?
- Describe how you do when you solve problems. Are there principles or methods?
- Describe how you go about to write program.
- Describe your experiences of programming and problem solving

- Which obstacles and problems did you meet?
- Tell me of previous experiences of getting stuck
- What got you on the track? Describe your plan to solve the problem
- Describe aha-experiences
- Describe how you usually work, the facilities, the compiler …
- Do you think it is plausible that you will get in a similar situation in the future?

17 Appendix D



Documentation of the system

STUDADMIN

(under construction)

1 Current status (version 0.5.1)

The application has been developed incrementally and is therefore executable and it is to some extent also tested. However, an important module is missing in order for the system to be ready for a complete system test.

1.1 Yet not implemented functionality in the user interface

- *Create and remove registrations (of students on course instances).*

All necessary database operations for this case are already implemented in the server class. It only remains the user interface handling of this in a new *plugin* (see *2.2 PluginPanel*).

1.2 Implemented functionality in the user interface (verified)

Tab sheet	Functionality (towards the database)	Plugin-class
Courses	Add, update and remove courses	CoursePluginPanel
Course Instances	Add, update and remove course instances	CourseInstancePluginPanel
Students	Add, update and remove students	StudentPluginPanel
Users	Add, update and remove system users	UserAdminPluginPanel
Plugins	Add, update and remove plugin-modules	PluginAdminPluginPanel

2 System description

2.1 General description

The system is a client/server application that offers administrative routines for handling students and courses in a university environment. A relational database stores all information concerning students, courses, course instances, registrations, users, etc. The client application (a leight weight user interface) calls methods that a server application provides. The system is developed in Java and can be executed as: (see *4 Execution* below for more information)

- a coherent single user application, where client- and server objects are executed on the same JVM on the same machine (suitable for test).
- a distributed multi user application, where the clieny- and server applications can be executed on different JVMs on arbitrary machines.

2.1.1 The server

The code on the server provides and executes all operations aimed for the database. In principle, the development of the required server operations for the given use cases is finished. The interface `studadmin.common.interf.Service` declares all operations that the server can perform. The implementation is in the class `studadmin.server.Server`.

2.1.2 The client

The code on the client handles the graphical user interface using Java Swing and takes care of user events and delegates operations to the server. The client consists of a window that contains menus and “tab sheets” for the various use cases. The client class itself handles only the most fundamental use cases:

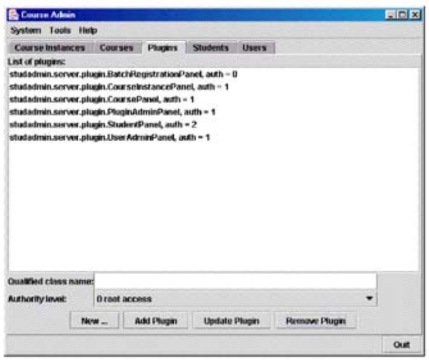
- log in,
- log out,
- quit.

All other functionality is implemented in so-called “plugin classes” (see *2.2 PluginPanel* below).

2

2.2 PluginPanel

This system has the possibility to create freestanding functionality in plugin classes. A condition for this to work is that all classes implement an interface named `studadmin.common.interf.PluginPanel`. This implies that it is possible to develop code for new use cases and add them to the system after it is installed and launched since there is no need to compile the client side of the system. All implementations of PluginPanel should be put in the package `studadmin.server.plugin`. In order to register them in the system (the database), the administrator must start the application, log in as root and enter the qualified name of the new class in the tab sheet "Plugins". This is the place to assign which authority level that is required for the end user to get access to the new plugin. The plugin objects are fetched dynamically from the server in run-time. Which the fetched PluginPanels are depends on the user's authority level. The plugin objects appear as tab sheets in the graphical user interface (see Figure 1).



```
List of plugins:
studadmin.server.plugin.BatchRegistrationPanel, auth = 0
studadmin.server.plugin.CourseInstancePanel, auth = 1
studadmin.server.plugin.CoursePanel, auth = 1
studadmin.server.plugin.PluginAdminPanel, auth = 1
studadmin.server.plugin.StudentPanel, auth = 2
studadmin.server.plugin.UserAdminPanel, auth = 1
```

Qualified class name:
Authority level: 0 root access

Figure 1. The tab sheet *Plugins* (PluginPanel)

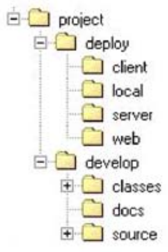


Figure 2. File structure

3 Development

The programming language Java is used overall (J2SDK, version 1.4.2). The working directory is normally `project/develop/` (see Figure 2). There are a number of shell scripts facilities for the work. The script files are made for MS-DOS environment, but can be modified for other environments such as UNIX.

3.1 Creating the documentation

Using the tool *JavaDoc*, a complete documentation of the source code and the packages is created in html format. Each class or interface has a hyper link to its source code. Create the documentation by running the script `makedoc.bat`. The first time the documentation is created, the script must be executed twice as it contains references to itself. The documentation is created as a directory tree and the files are automatically put in the directory `project/develop/docs/`. The root of the tree is the file `index.html`.

3.2 Compiling

In order to compile the source code the script file `compile.bat` is used. The class files is automatically put in the respective package in the directory `project/develop/classes/`.

3.3 Making executable archives (JAR-files)

In order to create executable archive files the script file `deploy.bat` is used. Archive files are created for every execution mode and is automatically put in respective directory in the directory `project/deploy/`. After code modifications, this must be done before execution.

4 Execution

The application can be executed in three different ways: local single user application, distributed multi-user application (client/server), and a combined mode where both client and server are executed on the local machine.

4.1 Local mode

Local application – recommended for tests during development (only one user).

- All files reside in the directory `project/deploy/local/`
- The class files reside in the archive file `local.jar`.
- Execute using `local.bat`

4.2 Client/Server mode

The client and the server can be executed on separate machines. The server handles several concurrent clients (multi user system).

4.2.1 Client

- All client files reside in the directory `project/deploy/client/`
- The client's class files reside in the archive file `client.jar`.
- `client.bat` must be modified with the correct IP address for the server machine.
- Start the client using `client.bat` in a shell (windows).

4.2.2 Server

- All of the server's files reside in the directory `project/deploy/server/`
- The server's class files reside in the archive file `server.jar`.
- Start the server using `server.bat` in a shell (windows).
- `project/deploy/web/web.jar` must be but on a web-server and its address must be put as an URL (`codebase` in `server.bat`)
- It is possible to give the following four commands to the server:

>DROP	- erase all tables in the database
>CREATE	- create tables in the database
>SAMPLES	- inserts test data in the database
>Q	- shut down the server

4.3 Semi - Client/Server mode

Execution on the same machine – the localhost.

- Start the client in the directory `project/deploy/local/`, using `client.bat` in a separate shell (windows).
- Start the server in the directory `project/deploy/local/`, using `server.bat` in a separate shell (windows).
- The server handles commands as in Client/Server mode.

5 Database

The system is designed to use a relational database that supports SQL and so far it is tested using Mimer and Access (ODBC). The tables and relations in the database are accounted for in Figure 3. Each table corresponds to a class in the package `studadmin.server.entity`. All responses (objects) to database queries, give references of interface type. These interfaces reside in the package `studadmin.common.interf`. The SQL statements that are used to create the database are put in the class BootStrap in the package `studadmin.server.db`. SQL statements for storing, updating and searching objects are in the package `studadmin.server.db.dao` in the corresponding DAO class. All of the specified database operations are implemented and tested. The operations are called through the server's interface (`studadmin.common.interf.Service`).

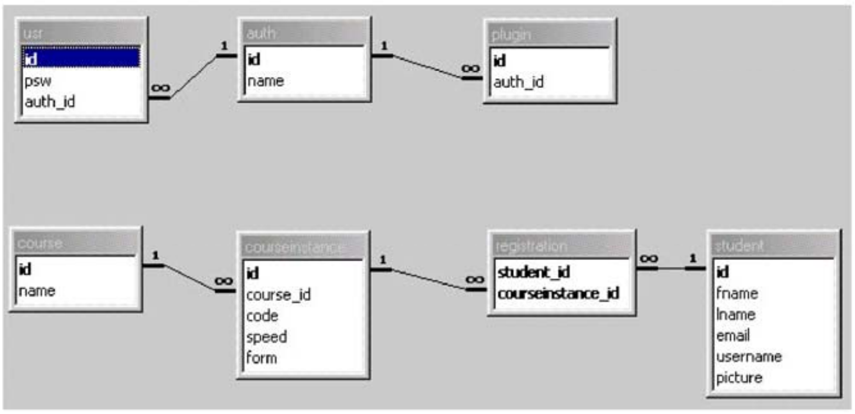


Figure 3 The tables in the database and the relations between the tables.

5.1 Test database

The present settings in the property files (`studadmin.server.db`) apply for Microsoft Access and a database named "`stud`". There is a test database in the access file `project/database/stud.mdb` (if the database does not exist, an empty database can be created using Microsoft Access – the name is not important). If the application fails to connect to the database, the problem could be that there is no defined data source with the name "`stud`". Set up a data source using the control panel and "32-bit ODBC data sources". Select "add" and "microsoft access driver". Name the data source "`stud`" and select the file that was created in access. You can enter test data to the database using the server application (see execution - client/server mode).

5.1.1 Inserted users in the database

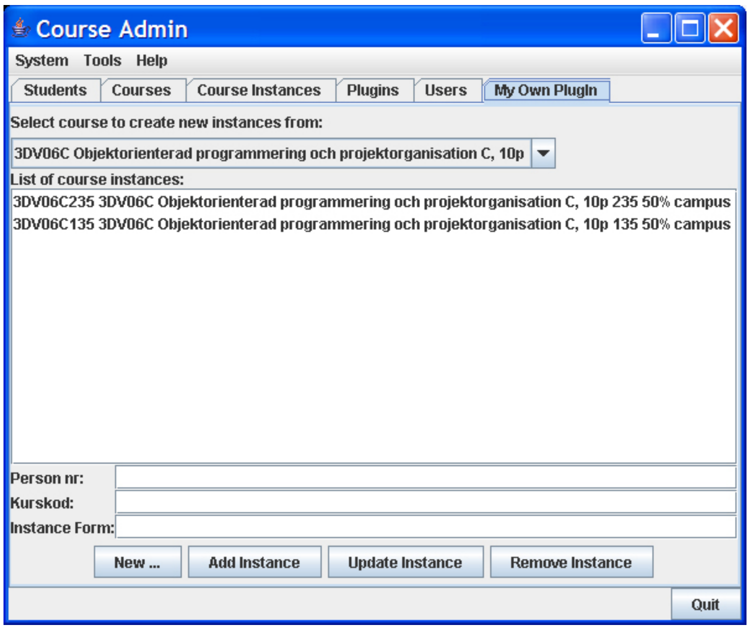
User name	Password	Authority
root	root	0 (highest)
admin	admin	1 (high)
user	user	2 (normal)

18 Appendix E

18.1 Visible traces of the participants designs

This appendix accounts for the visible evidence that the students left behind. Some of the students had succeeded to install their plugin to the system, and where appropriate, we could execute their programs and analyse them, as their plugins was part of the system's graphical user interface.

However, among the students who did not make it all the way to an executable plugin, some still left evidence that reveal how they had planned the graphical layout. Most of those who started to program had copied a similar plugin's source code, which contained a sketch of its layout in a comment block, and hereby, some students were inspired to plan their own layout in the comment block. Hence, we can study their intended design, also.



Select course to create new instances from:

3DV06C Objektorienterad programmering och projektorganisation C, 10p

List of course instances:

3DV06C235 3DV06C Objektorienterad programmering och projektorganisation C, 10p 235 50% campus
3DV06C135 3DV06C Objektorienterad programmering och projektorganisation C, 10p 135 50% campus

Person nr:
Kurskod:
Instance Form:

Figure 8. Alf copied a class in order to create a plugin of his own, but did not have time to change much of its design.

Alf was stuck and asked for help to get going, and he was suggested to look at code in the directory where all the other plugins resided. With this tip he managed to create the class "MyOwnPluginPanel" by taking a copy of the existing class “CourseInstancePluginPanel” and then changing some identifier names the code. Then he could compile and execute the program and install the new plugin module (see Figure 8). After that step was taken, we can see that he had started to adapt the code to the desired functionality by changing two of the text labels, in the lower left, to “Person nr:” and “Kurskod:”, where it used to say “Instance Code:” and “Instance Speed.” This was the only changes Alf had time to do, and without the help, he probably would not have reached this far.

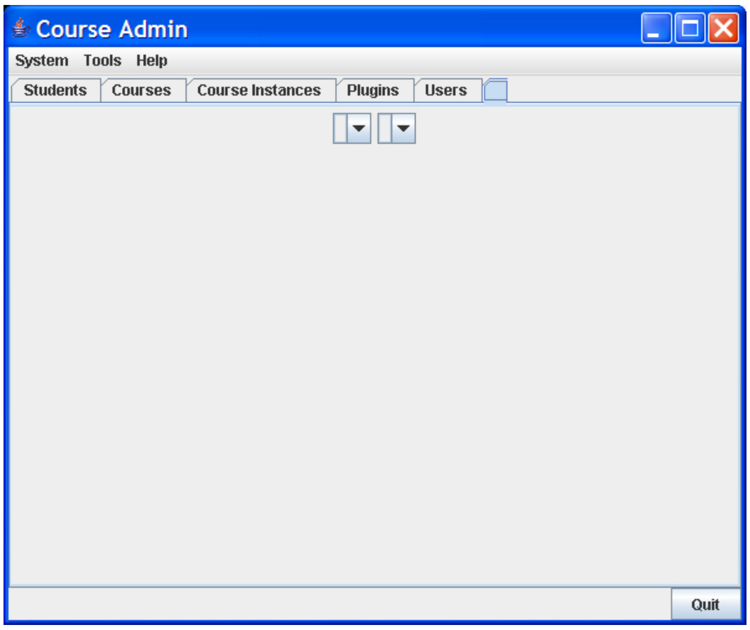
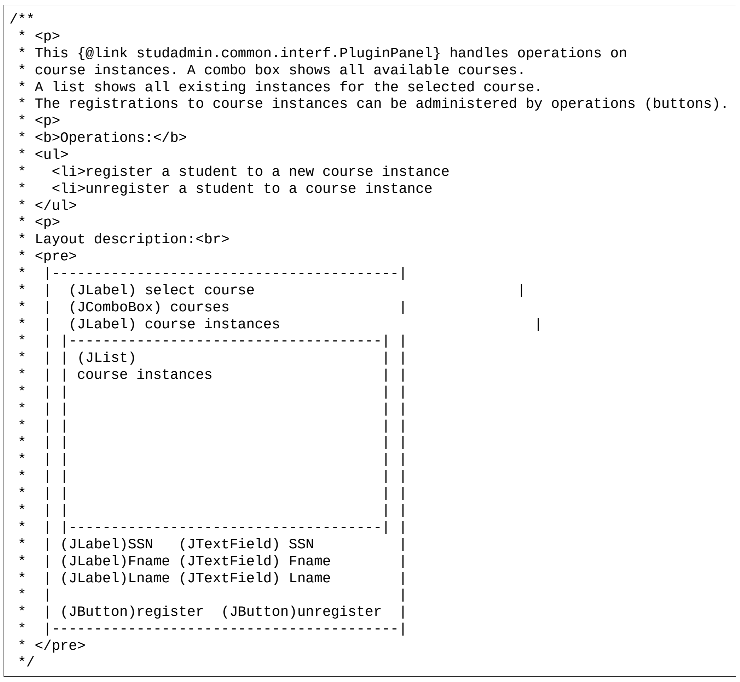


Figure 9. Bea started from the *interface* PluginPanel, and that is why her plugin looks so empty.

Bea worked was completely independent in her working and managed to create a plugin class that was she installed in the system. Apart from the others who reached as far as to write code, she started from the *interface* “PluginPanel” instead of using a class. She copied the list of method specifications in the interface and completed them into “dummies” in her new class. This explains why her implementation looks so empty; she has not even put the title on the tab (see Figure 9). However, she has started to work on the implementation, as we evidently can see how she has planned for two *combo boxes*. Presumably, they are supposed to show data that are connected, one for students and one for course instances. That is a good start, but it still needs some extra components to make it a well-working solution; buttons to execute registration and deregistration, for one thing. In addition to this, there is a logical problem with the two combo boxes. It will work fine for a registration, as it is easy to select a course instance and one of all available students. However, for deregistration, it would be much easier if the combo box listed only the registered students. These wants contradicts each other and are hard to meet using this design.

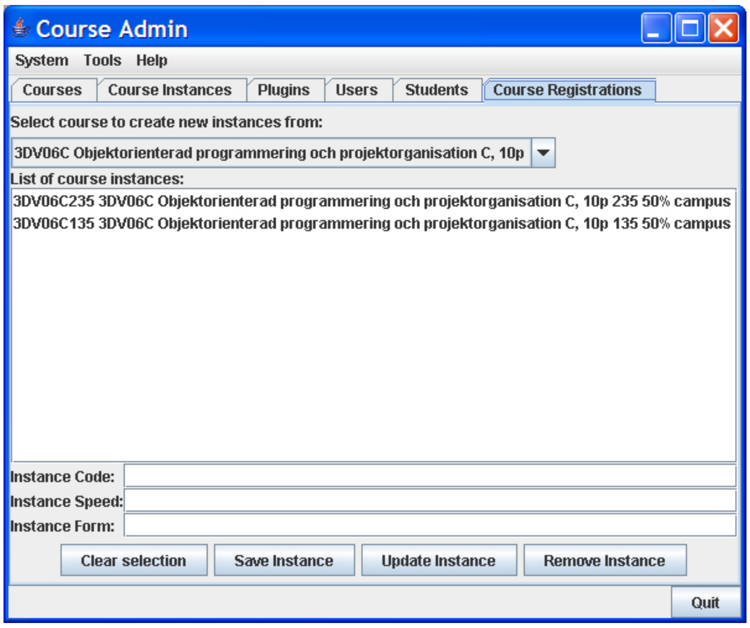
Cia’s way to work was unique as she never tried to run the program or even tried to compile her code. In spite of this, she managed to get fairly well ahead through her using the interface “PluginPanel” and copying selected parts from other classes. We can see how she designed the layout of her plugin by looking at her code. In a comment block in the class, she has sketched how the components should be located in the GUI (see Figure 10). Here is all we need to select course, select course instance, register, and deregister. However, it is noteworthy that Cia defined text fields for social security number, name, and surname for a student, as those data are handled in a separate part of the program (“StudentsPluginPanel”). The only required input is to define the student’s social security number, from selecting it in a list or a combo box, or by entering it manually, which is what uniquely identifies a student in the database system, and hence the name and surname is just redundant information for the registration.



```
/**
 * <p>
 * This {@link studadmin.common.interf.PluginPanel} handles operations on
 * course instances. A combo box shows all available courses.
 * A list shows all existing instances for the selected course.
 * The registrations to course instances can be administered by operations (buttons).
 * <p>
 * <b>Operations:</b>
 * <ul>
 *   <li>register a student to a new course instance
 *   <li>unregister a student to a course instance
 * </ul>
 * <p>
 * Layout description:<br>
 * <pre>
 *  |-----------------------------------------|
 *  |  (JLabel) select course                               |
 *  |  (JComboBox) courses                    |
 *  |  (JLabel) course instances                              |
 *  | |-------------------------------------| |
 *  | | (JList)                             | |
 *  | | course instances                    | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |-------------------------------------| |
 *  | (JLabel)SSN   (JTextField) SSN          |
 *  | (JLabel)Fname (JTextField) Fname        |
 *  | (JLabel)Lname (JTextField) Lname        |
 *  |                                         |
 *  | (JButton)register  (JButton)unregister  |
 *  |-----------------------------------------|
 * </pre>
 */
```

Figure 10. Cia made a sketch of the layout in the comments.

Git copied “CourseInstancePluginPanel” precisely as Alf, but did not make any visible changes as all (see Figure 11).



Course Admin

System Tools Help

Courses | Course Instances | Plugins | Users | Students | Course Registrations

Select course to create new instances from:

3DV06C Objektorienterad programmering och projektorganisation C, 10p

List of course instances:

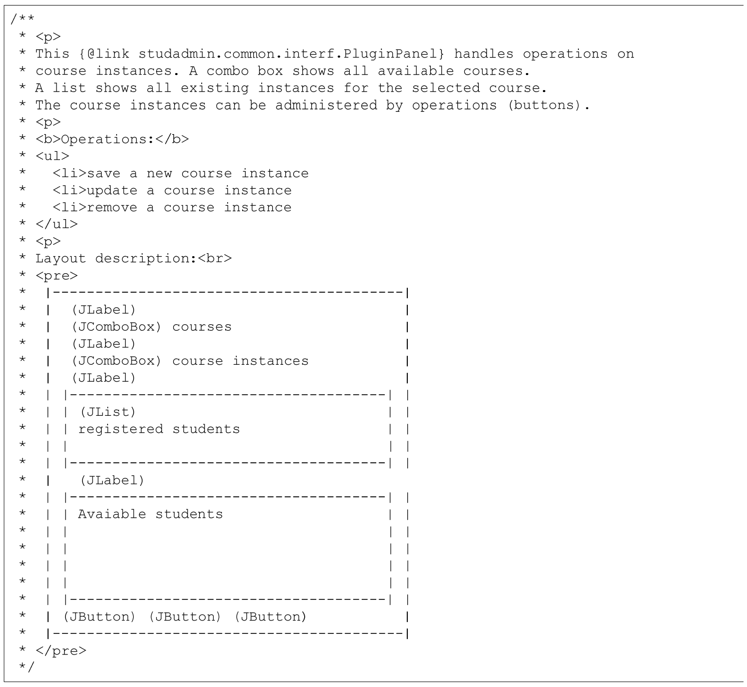
3DV06C235 3DV06C Objektorienterad programmering och projektorganisation C, 10p 235 50% campus
3DV06C135 3DV06C Objektorienterad programmering och projektorganisation C, 10p 135 50% campus

Instance Code:
Instance Speed:
Instance Form:

Clear selection | Save Instance | Update Instance | Remove Instance

Quit

Figure 11. Git copied an existing plugin but did not change the copy.

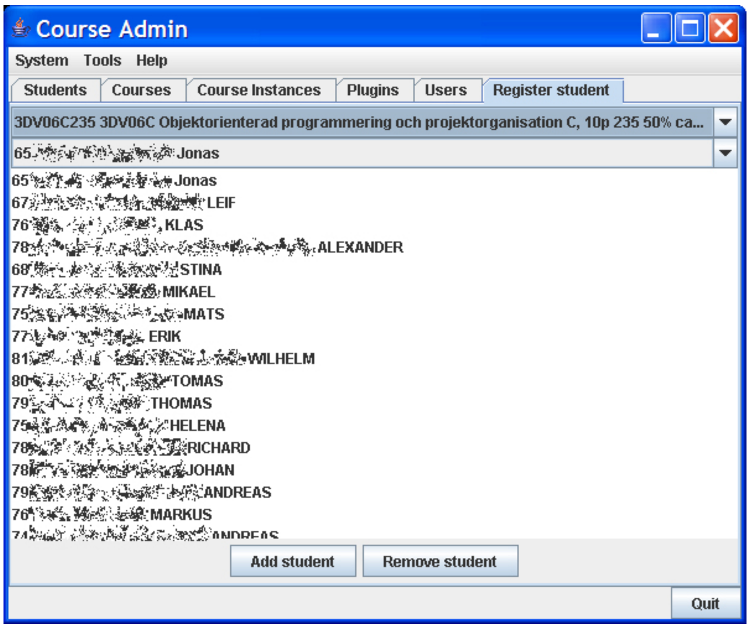


```
/**
 * <p>
 * This {@link studadmin.common.interf.PluginPanel} handles operations on
 * course instances. A combo box shows all available courses.
 * A list shows all existing instances for the selected course.
 * The course instances can be administered by operations (buttons).
 * <p>
 * <b>Operations:</b>
 * <ul>
 *   <li>save a new course instance
 *   <li>update a course instance
 *   <li>remove a course instance
 * </ul>
 * <p>
 * Layout description:<br>
 * <pre>
 *  |------------------------------------------|
 *  |  (JLabel)                                |
 *  |  (JComboBox) courses                     |
 *  |  (JLabel)                                |
 *  |  (JComboBox) course instances            |
 *  |  (JLabel)                                |
 *  | |------------------------------------| |
 *  | | (JList)                            | |
 *  | | registered students                | |
 *  | |                                    | |
 *  | |------------------------------------| |
 *  |   (JLabel)                               |
 *  | |------------------------------------| |
 *  | | Avaiable students                  | |
 *  | |                                    | |
 *  | |                                    | |
 *  | |                                    | |
 *  | |                                    | |
 *  | |------------------------------------| |
 *  | (JButton) (JButton) (JButton)          |
 *  |------------------------------------------|
 * </pre>
 */
```

Figure 12. Hal planned for a smooth design of the user interface.

Hal never managed to get his class to run even though he copied from the existing class “CourseInstancePluginPanel”. However, he made a thorough description of the planned layout in the source file’s comments, and it seems very well considered (see Figure 12). He saw two combo boxes where you choose course and course instance. Below the combo boxes, the uppermost list shows the registered students for the selected course instance. The other list shows all of the students in the system. To make a registration, you select a student in the lower list and select a course and its instance. Then you press the register button, and the student is “moved” to the upper list, which confirms the registration. When you want to deregister a student, you select course and instance as before and then you selects an already registered student from the upper list and press the deselect button, and as a confirmation, the student is moved back to the lower list. This was a very convenient solution, but sadly, Hal never saw it working.

Joe was the only student who made a complete and working implementation on time (see Figure 13). He started his work with the code similar to the most of the students by taking a copy of an existing class, and then started from the copy by removing parts of the code and by making changes and additions to it. His solution for the layout was to put a combo box at top, in which he would list every available course instances in the database. Below was yet another combo box that in a similar way listed all the students. A list below the combo boxes showed all registered students. To register a student you should simply select the course instance and student in the combo boxes and then press the “add student” button. Rather, if you want to deregister, you select the course instance in the combo box and the registered student in the list, and then press the “remove student” button. We have tested Joe’s solution, and it works precisely as is it supposed to do, all the way to the database.



3DV06C235 3DV06C Objektorienterad programmering och projektorganisation C, 10p 235 50% ca...

65[illegible]Jonas

65[illegible]Jonas
67[illegible]LEIF
76[illegible]KLAS
78[illegible]ALEXANDER
68[illegible]STINA
77[illegible]MIKAEL
75[illegible]MATS
77[illegible]ERIK
81[illegible]WILHELM
80[illegible]TOMAS
79[illegible]THOMAS
75[illegible]HELENA
78[illegible]RICHARD
78[illegible]JOHAN
79[illegible]ANDREAS
76[illegible]MARKUS
74[illegible]ANDREAS

Figure 13. Joe’s plugin is operational (we have “blurred” all authentic names).

Ken never could trial run his plugin, but he sketched the layout in the comments (see Figure 14). His solution has all required components. Two combo boxes select course and instance, and under them, he locates a list that shows registered students to the course instance and in the bottom, yet a combo box containing every available student and buttons for registration and deregistration.

```
/**
 * <p>
 * This {@link studadmin.common.interf.PluginPanel} handles operations on
 * course instances. A combo box shows all available courses.
 * A list shows all registrated students for the selected course.
 * The registrations can be administered by operations (buttons).
 * <p>
 * <b>Operations:</b>
 * <ul>
 *   <li>create a new student registration instance
 *   <li>remove a registrated student instance
 * </ul>
 * <p>
 * Layout description:<br>
 * <pre>
 *  |-----------------------------------------|
 *  |  (JLabel)                               |
 *  |  (JComboBox) courses                    |
 *  |  (JLabel)                               |
 *  |  (JComboBox) course instance            |
 *  |  (JLabel)                               |
 *  | |-------------------------------------| |
 *  | | (JList)                             | |
 *  | | Student instances                   | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |-------------------------------------| |
 *  | (JLabel) (JComboBox)                    |
 *  |                                         |
 *  | (JButton) (JButton) (JButton)           |
 *  |-----------------------------------------|
 * </pre>
 */
```

Figure 14. Ken’s design as shown in the comments.

Leo made comments in the comments, and his plugin “CourseMember-sPluginPanel” is running. Moreover, when the program executes, the real thing matches his layout in the comments (see Figure 15 and Figure 16).

```
/**
 * <p>
 * This {@link studadmin.common.interf.PluginPanel} handles operations on
 * course members. One combo box shows all existing course instances, and
 * another shows all existing students.
 * The course instances can be administered by operations (buttons).
 * <p>
 * <b>Operations:</b>
 * <ul>
 *   <li>Register a new student on the course instance
 *   <li>Remove a student from a course instance
 * </ul>
 * <p>
 * Layout description:<br>
 * <pre>
 *  |-----------------------------------------|
 *  |  (JLabel)                               |
 *  |  (JComboBox) Course Instances           |
 *  |                                         |
 *  | |-------------------------------------| |
 *  | | (JList) List of course members      | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |                                     | |
 *  | |-------------------------------------| |
 *  |                                         |
 *  |  (JLabel)                               |
 *  |  (JComboBox) Students                   |
 *  |                                          |
 *  | (JButton) (JButton) (JButton)           |
 *  |-----------------------------------------|
 * </pre>
 */
```

Figure 15. Leo’s design as he sketched it in the comments.

His solution of the graphical user interface consists of a combo box that holds all course instances. Below it, a list shows all the “course members”. Below the list, yet another combo box contains all of the students in the system. If you want to register a student to a course instance, the intention is that you choose course instance and student in the combo boxes, and then presses a button. To deregister a student, you choose course instance as previously, select the student in the list, and press the other button. This solution would work fine. Unfortunately, he did not implement any of the underlying functionality in his plugin.

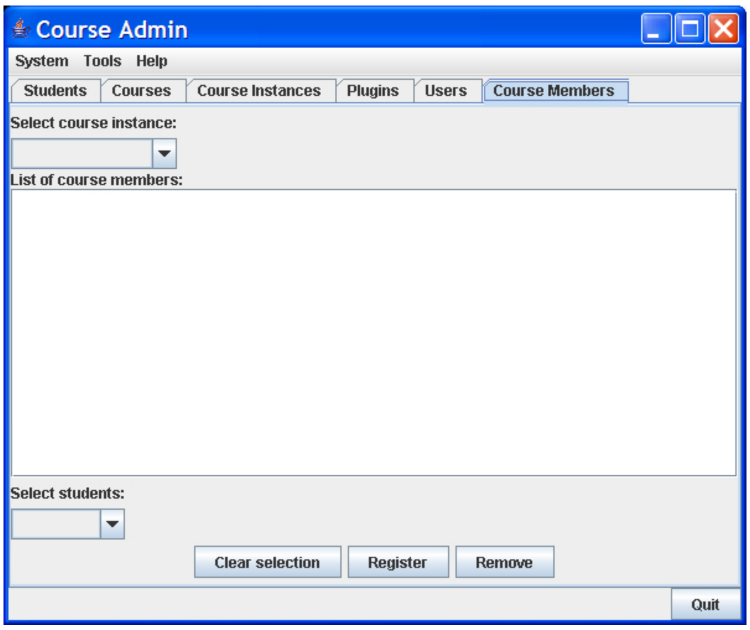


Figure 16. Leo’s plugin as it is seen in the running application.