A Student Perspective on Software Development and Maintenance

Jonas Boustedt
Division of Scientific Computing
Department of Information Technology
Uppsala University
SE-751 05 Uppsala, Sweden
jbt@it.uu.se

March 24, 2010

ABSTRACT

How do Computer Science students view Software Development and Software Maintenance? To answer this question, a Phenomenographic perspective was chosen, and 20 Swedish students at four universities were interviewed.

The interviews were analyzed to find in which different ways the informants, on collective level, see the phenomena of interest. The resulting outcome spaces show that software development is described in a number of qualitatively different ways reaching from problem solving, design and deliver, design for the future and then a more comprehensive view that includes users, customers, budget and other aspects. Software maintenance is described as correcting bugs, making additions, adapting to new requirements from the surroundings, and something that is a natural part of the job.

Finally, conclusions from the results and additional observations are discussed in terms of their implications for teaching, and some suggestions for practical use are given.

1. INTRODUCTION

One of the main goals for Computer Science and Computer Engineering education is to prepare students for careers in the software industry. It is possible to identify several indications on how the need for competence in the industry influences the academic education. For example we can see how the industry adopts new programming languages and new methods for software development and how this affects the content in computer science educations [1].

The greater part of university level Computer Science Education Research focuses on students who are learning to program at beginners' level. The reason for this is that many of the problems that educators are experiencing *within* the education programmes are related to novices and their first programming course. The throughput has been low and the drop-out rates have been high. A specific circumstance that relates to the introductory courses in programming is that many of the students who take these courses are not computer science majors. Still, they are often forced to take these courses as a mandatory part in their study programmes.

Compared to research on novices, research on advanced level undergraduate students with Computer Science or Software Engineering as major subjects, is most moderate, and research on graduated students who have recently started their professional careers in the industry is even more rare[1]. Consequently we do not know much about how well we manage to prepare our students for their professional lives *outside* the academic environment.

Surely, there are more to learn about what awaits a student who gets a job in the software industry and how prepared he or she is for that situation. In a rather unique study, the researchers contacted a number of former students who recently had been employed by a big software company, and they followed them for a couple of months. The beginners experienced that they were technically well prepared for the new situation; however, they did not think that they were prepared for the social situation in the large, hierarchic work teams and neither were they prepared for work with old and vast legacy code. Their tasks were mainly to debug the software, make minor additions and work on documentation [1].

A generic term for tasks aimed at keeping existing software system running is *Software Maintenance*[2]. Since it is expensive to develop new software from scratch, the natural endeavour is to get the longest possible life-time from existing software systems. To obtain this goal, it takes a carefully organized way to maintain the software that builds the system. The research literature in this area points out that software maintenance is a task that newly employed persons in IT organizations are most likely to get [1, 6]. This work requires qualified knowledge, for instance, the knowledge required to diagnose malfunctions in a system. Nevertheless, it is sometimes hard to recruit staff, partly because only a few have adequate education for this work, and partly because maintenance seems to have low status [6].

Even though software maintenance is a plausible work task for newly graduated students it is not at all evident that it is a topic in a computer science program. This is

[1]There are good examples of this kind of research however, e.g., [1].

[2]The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [14].

because it is often ascribed to the subject area Software Engineering, which in its turn often is only a small part of a typical computer science program [14]. Hence, it may be that the students could and should be better prepared for the tasks that many of them will face in the beginning of their careers. The question is how to alter the education in order to enhance the students' competence and awareness about aspects of their future profession. Can it be done without infringing too much in the existing subjects?

Having the goal of improving the computer science education, it is important to first take into account how the students themselves experience their studies and future prospects, and then start from this knowledge when it is time to consider how the education could be improved. The purpose of this study is to empirically investigate how students view the concepts *software development* and *software maintenance*. I also aim to indicate how the results from these investigations can give a basis for ideas that can help to make improvements to the education and make students better prepared for getting started with their future careers.

The study addresses students in the final stage of different computer science programmes and intends to answer the following research questions:

1. In which ways do students understand software development?

2. In which ways do students understand software maintenance?

2. METHOD

This study takes a phenomenographic perspective, which means that it is designed to investigate how students see things, and based on their descriptions, the aim of the analysis is to find categorizations of qualitatively different ways to experience the phenomena.

In general, a phenomenographic study tries to answer questions that relate to persons, often pupils or students enrolled in some particular education. The questions concern in which ways various educationally related phenomena are understood or *experienced* within this specific group and the data are collected through interviews with the people in the group. However, it is normally not possible to conduct interviews with *every* person in the group, and consequently the participants in a phenomenographic study must be a selection. It is important that the informants are chosen in a way that allows for a broad variation of possible ways to see a phenomenon. This is because the phenomenographic research aims to find and show the differences and variations in the way phenomena in the world are "understood" (described) by people. It is anticipated that the most common and important understandings are captured if (1) the number of participants is big enough (about 20) and (2) the persons are selected with care and (3) the researcher uses a keen ear during the interviews and adapts to what the informant says by posing follow–up questions.

The population in this study are Swedish computer science or computer engineering students who will soon leave the university and look for work. The data were collected via semi structured interviews with 20 final year computer science or computer engineering students at four different universities in Sweden and there were 18 male and 2 female informants in ages between 20 and 39, some of whom have had work experience from the software industry. The reason for choosing different universities was to get a variation that reflects the entire population better compared with only selecting participants from one single educational institution. The informants follow different study programmes in computer science, computer engineering or informatics. These educations are three or five years long. Two of the four institutions are major universities with heavy research, whereas the other two are smaller institutions that have not received official university status.

In order to protect the informants, their real identities are kept secret. Instead, they are referred to as S01 – S20. Moreover, all informants are referred to as males using masculine pronouns (he, his, him), because there were only two female informants and for some persons it would be easy to guess their identity.

The interviews were held in Swedish, the mother tongue of the participants, and consequently all quotes from the interviews in this paper are translated into English.

Two main topics for discussion during the interviews was how the informants perceive the concepts of software development and software maintenance. Software development was discussed prior to software maintenance, and then the interviews went on with discussions of the informants' view of how they would be affected by these tasks in the future.

A phenomenographical approach was used to analyse the interviews, which means that the transcribed interviews were examined to find expressions of "meaning" of software development and maintenance. For each research question, the goal was to establish an "outcome space"; a set of categories that expressed distinct ways of experiencing the "phenomenon" in question, on a collective level. The next section gives an introduction to phenomenography and Section 3.1 describes how it was applied in this study.

3. PHENOMENOGRAPHY

Phenomenography originated in educational questions of how learning comes about and how the learning process can be improved. It gradually evolved and matured into a research tradition that concerns how different aspects of the world appear to people. Essentially, the studies within this approach are explorative and use empirical data, and they all take a second order perspective on a phenomenon. That is to say, the phenomenographer does not study the phenomenon as what it is (a first order perspective), but the variation in how it is experienced by a group of people. Marton gives the following definition of this research specialization:

> Phenomenography is a research method adapted for mapping the qualitatively different ways in which people experience, conceptualize, perceive, and understand various aspects of, and phenomena in, the world around them. [7]

Experiences from earlier studies had shown that dif-

ferent people described phenomena in only a few different ways, which led to a fundamental epistemological assumption, namely, that there are only a limited set of qualitatively distinct ways to experience a given phenomenon.

Bowden [3] outlines the phenomenographic research process as having four stages: plan, data collection, analysis and interpretation. The plan defines the purpose and the strategies for the research, which naturally is driven by an underlying question that the researcher tries to answer. Essentially, the data are collected from people's statements in interviews where they are asked open-ended questions about a phenomenon, and it is important to make a careful selection of interviewees in order to capture a wide variety of experiences. During the analysis, the transcribed interviews are sought for different meanings and contexts concerning the phenomena of interest. Finally, the results should be interpreted, and in applied phenomenography, this involves how researchers and teachers can use the results in pedagogy and instruction.

In phenomenographic analysis, the researcher converts the primary source of data by transcribing the recorded interviews. The next step is to search the texts for different expressions of meaning that relate to a certain phenomena. Walsh states:

> 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. [16]

Manifestations of meaning are found where the interviewee explicitly describes the phenomenon as such, however, implicit descriptions can also reveal meanings, e.g., in descriptions of the use, purpose, advantages or drawbacks of the phenomenon.

The meanings of the focused phenomena are expressed by quotes that form a pool of refined data, and the quotes are usually de-contextualized, which makes it possible to find distinct qualities. Nevertheless, references to their original contexts are kept for the possibility of re-interpretation. The fragments of meaning are condensed into clusters of meaning that are abstracted and outlined in "categories of description." A prominent feature of the categories is that they are on a "collective level" as they do not express any particular individual's understanding; rather they are the result of an analytical categorization of all relevant meanings found in the data. In the process of forming categories, the researcher tries to find different "dimensions" in the sense that each category opens up a new way to "see." This avoids categories that are instances or variations within the same dimension.

The main result of a phenomenographic study is the "outcome space" which is constituted by the categories of description and their logical interrelations, and since a non-dualistic view is assumed, the outcome space can be regarded as a synonym for the phenomenon [8]. A common logical relation in an outcome space is "hierarchic inclusiveness," which implies that the categories include each other in the sense that a certain understanding also includes or implies a more elementary understanding. As phenomenography originated in studies that aimed to understand or improve 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. Marton and Booth explain:

> 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. [9, p.126]

3.1 How the phenomenographic analysis was conducted in this study

In this study, the analysis started with reading all the transcribed interviews to get an appreciation and overall perspective of the whole context. During the reading, all text sections relating to the specific concept, e.g., software development, were marked.

The next step was to collect all of the marked text sections and copy them into a separate document, and then import the document into the computer based analysis tool Atlas.ti [12]. This tool did not analyse the data automatically in any sense; however, it made the text easy to tag. The tool made it possible 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 were added, identifying interpretations of each marked quote. The software supported examining the data from several perspectives. For instance, it was easy to find and collect 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 were represented as graphical symbols, structured as nodes in a graph.

The various codes of meaning were then analysed to find qualitative similarities and differences between them, and hence, different clusters of meanings were condensed. Before these groups were considered as preliminary categories of description, they were further scrutinized in view of the requirement that categories should open up new dimensions in the phenomenographic outcome space, or new relations between dimensions.

The final step of the analysis process regarded the relational perspective where the categories were arranged in a logical structure based on two criteria: (1) an evaluation of their compliance with the educational goals, and (2) an hierarchical ordering of the categories, such as inclusiveness and dependency.

4. THE INFORMANTS AND THEIR PLANS

Most of the participating informants in this study were relatively young. The median age was 23 and the majority seemed to have started their university studies shortly after graduating from upper secondary school. However, some informants were more experienced – the ages vary between 20 to 39. Most students studied at three year bachelor or engineering programmes in computer science; however, in three cases the informants were studying at master level, which means four or five years of study. Three of the students were not following an integrated study programme consisting of sev-

eral subjects, but were taking a number of full semester computer science courses instead, and they were doing it on top of some other university degree. Unfortunately, there were only two female informants in this study and this reflects the uneven and undesired gender distribution in many computer science related study programmes in Sweden. All informants were in the final stage of their education. Half of them had only about a month to go before graduating and the other half had one year to go.

Table 1: Summary of the informants' future prospects

Future profession	n
Software developer	11
Programmer	4
Has other job, wants to be a programmer	1
Project leader, programmer, entrepreneur	1
Human Computer Interaction, programming	1
Requirements analyst	1
IT and design	1

Since the purpose of the present study is to investigate how education prepares students for the demands from the software industry, it was important that the informants wanted such careers. It turned out that all informants wanted to go for a job in the software industry in the future, with some different directions. A summary of the informants' choice of future professions is shown in Table 1. Three main groupings appear in the list of professions. The first and most dominating desire (15 out of 20 informants) is the wish to work with software development or programming. Within that group, 11 informants want to work with software development and 4 use more specific terms when they say they are interested in "programming" or "coding". One of them had already got a surveillance job and was pleased with that for now, but would actually like to work as a programmer later.

The second group expressed a desire to do something specific or have a specific role where programming is part of the means to achieve the goals. One informant would like to be a project leader and start his own business in the future, but would like to start as a programmer. Another informant was especially interested in the subject area Human Computer Interaction and would like to improve the computer based tools that people use at work, which probably would require some programming.

The third group desired to work with "softer" areas connected to software and system development. One informant wanted to work with requirements analysis, "the person who investigates what the program should contain", and one informant wanted to work with "IT and design". It is noteworthy, however, that these were the two only female informants in the study.

In summary, an observation is that most of the informants want to work with software development on a rather tangible level and that some want to deal with softer aspects, such as design of graphical user interfaces or end user interviews; however, all of the informants see a future profession related to software development.

5. HOW THE INFORMANTS VIEW SOFTWARE DEVELOPMENT

A number of qualitatively distinct ways to describe software development appear in the informant collective, forming categories of description and the outcome space, which can be interpreted as qualitatively different ways to experience, understand or "see" the phenomenon "software development".

In the analysis, a professional perspective on software development was used as a guidance when interpreting the relations between the categories of description. Taking this position means that descriptions of "soft" qualities is valued highly. The students are going to work in a reality where aspects such as team work, communication with customers and users, helping them to define what they want, the time frame, and the budget limits, are very important to understand.

The first and most fundamental category describes the phenomenon "software development" as creating or building a program that solves a problem or satisfies a demand. The descriptions often emanate from an educational point of view or a hobby situation and they mainly give a personal and subjective perspective on programming.

The next category describes software development as working out how a program should be constructed in order to fulfill the expectations or requirements that have been posed. In some cases, the descriptions also indicate a professional attitude, for example there appears a consciousness of the differences between hobby projects and professional projects.

In the third category, software development is described with future aspects in mind and takes a wider perspective than only focusing on the specific application to be developed. This way of seeing involves aspects of software quality that should be considered when software is developed. These aspects are important to make the software understandable, reusable, maintainable and endurable. This way of seeing has an obvious professional point of view.

The fourth category describes software development from a professional and businesslike perspective which is manifested in many ways, such as, you have to actively help customers and end users to understand what they need and formulate their requirements, that there are budget limits for software development projects, and that it may take long time before you can expect to get return on investments in a project.

A summary of the outcome space for the phenomenon "software development" is shown in Table 2.

5.1 Category 1: Creating a program that solves a problem

In this category of description, software development is described as something that solves a problem, to achieve something through building a program, to get a computer to perform tasks for a user, or simply, to write code. The informants describe the phenomenon in general from a perspective that is mostly personal.

Some of the informants describe that they experience

Table 2: A summary of the outcome space for the descriptions of software development.

Category	How *software development* is described
1: Solve a problem	Software development is described, mostly from a subjective perspective, as finding a solution to a problem or creating or building a computer program that solves a problem, that meets a need or realizes an idea.
2: Design a program	Software development is described as finding out which functions and parts should be included in a program to meet the need and how they should be designed. Different design methods are described to achieve the goals.
3: Design for future	Software development is described from a professional perspective as designing for the future; it is important that the software can handle future changes, be reused and that the design is documented so it can be understood.
4: Understanding need and whole	Software development is described from a professional perspective as designing software and understanding what the customers and end users need, what can be achieved, the time frame, the economic aspects, and which methods to use.

software development as a satisfying problem solving process. For example S15 tells that the work invested in solving the problem is rewarded when he finally can see his creation taking shape:

> I see it as, well it is kind of a problem solving that, that gives a reward when you manage to solve it and you can see a result. That is the reward I guess, that you manage to solve the problem, that you see that you can create something. That is what I think is fun. [S15]

In a similar way, S03 describes software development as a challenge:

> Good question, well, like a challenge, to find a solution to a problem. [S03]

S10 starts to talk about problem solving and then describes a head-on approach to reach the result, which is to make a program:

> Yes, problem solving, but, well, it is the programming, just build on and go on and go for a result that you want to achieve in some way... [S10]

A similar way to describe software development is to see it as achieving a goal, to solve a task; S09 describes:

> No, I don't know, it's like a part of achieving something, if I could say so. You will get like a goal or something, a task... [S09]

Software development is described by S05 as creating a program that can be used for something:

> You create, you create a, you create something in a computer that a user should be able to use for something. A hard question, well, you create a program. [S05]

S18 describes that software development is about solving the "problems" we are having with tedious tasks in our everyday life by having a computer making the work for us:

> To make computers, or machines in general, ta make machines to do things for us that we do not want to be bothered about ourselves, so that the everyday life gets easier and more comfortable. Well, first you become aware of a problem, it would be great if there was something that could do this because it is so boring to just sit there and do the same thing over and over again... you could have something that does it for you, you just snap your fingers and you make the first step and then everything else is settled because it is deterministic work in some way, and it seems that computers are good at doing that. So it is this kind of things it starts with, it starts with a problem I guess. [S18]

Many of the descriptions in this category make a generalizing use of the term "problem". In some cases, however, it is used in a more concrete context; That someone has a real problem and assigns this problem to another person that will solve it by writing a computer program. For example, S04 describes how you can create a program that solves a problem that a company has:

> ... a company may have, well, that they have some problem that they want to get solved, and that you, well I don't know, that you kind of, from scratch can create like a, well, a program that satisfies the need they have. [S04]

Some of the informants were very concrete in how they described software development, and one of the aspects that appeared was how the software is created through a personal work effort dealing with concrete program source code. S08 describes how he implements an idea into source code:

> It's about realizing I guess, being able to get an idea in, into code you know... If I am going to code something, then I usually starts by having a look at examples of what other people have done... when you look at other

> peoples code you may get an aha moment in your head... or looking at old code of my own... and then you think that it is possible to do it in this way but perhaps I should do it a bit different, and it is in that moment the code forms in my head, kind of... I don't like to plan to much ahead for exactly how I should de it. I prefer to get started and then let it flow for a while – when you get in some basic stuff and then you build on that, kind of... [S08]

S20 gives this summary of his view on software development:

> ... because I see all development of software as solving a problem, or dealing with a need, through the computing power that resides in a computer... [S20]

In summary, software development is understood as problem solving in this category, that is, the challenge or the process of creating code that solves a certain given task. And the descriptions often express that it is joyful and satisfactory to succeed in solving a problem.

5.2 Category 2: Designing a program

In the previous category software development was described as writing programs that solve problems, only the descriptions did not say how to write the programs to achieve this goal. The second category of description includes the first category, but what is added to the descriptions is that they concern various approaches and methods that the informants use to form, that is, design the software. In this category we can observe a slight shift towards the professional context.

Software development is generally described as something that takes place in steps. S09 describes that:

> ... you build something step by step, kind of. Design, and then it is implementing it and, well. [S09]

Some informants gave a fuller description of software design, and did so mainly in two ways. The first way to describe software design is that it is about planning and specifying before you start to write program code, and the other way is to develop an early prototype of the software and then refine it in steps.

Software development is described as planning and specifying in the beginning, S16:

> Well, it is very much work with specifications in the beginning, I guess. [S16]

S11 describes how software development follows a certain order where you analyse and plan in the beginning before you program:

> I would like to say that it is a number of things, there is this software building routine, you do some analysis first, only concerning the function you want, you design the program and in the last phase, the last stage, you program and after that you look for bugs. So that is just it, to plan first, what is it called, design. [S11]

S05 describes that software development requires that you first must plan what to do and how it should be done, and only after this planning you can start creating a design (in this case S05 refers to the graphical user interface design). When the design of the user interface is finished, you can write the code that handles what should happen when someone uses the interface:

> Planning what you want to achieve, and plan how it should be accomplished. Then I usually start with creating the design, what it will look like, and then I try to create the logic that is behind when the user is playing with the design, what it supposed to happen. [S05]

This understanding of how software development takes place is shared by S04 who has own experiences of software development in professional settings. He describes that it is important for him to start with a visual prototype of the program's graphical user interface to be able to get an understanding for what the program should be able to do and what it should consist of:

> I work lots in Photoshop, well, I make up the entire user interface for exactly how it should work, and then some ULM diagrams on top of that. It depends a bit, but, well, I make UML diagrams and then you start to program after that, because then I believe that I get an overview of the entire system and see parts of how, how the parts are supposed to be connected, and such. [S04]

S07 describes that the solution to a problem may be useless unless he first has thought through and documented how the problem should be solved before the program is written, and that this is probably the way it is done in larger companies in the industry:

> I could sit down and start coding it right away, but the solution would probably be useless because I had not thought things through. I would not have been sitting on the top looking out over the problem and thought through how it should work. Instead, the design is a documentation that enables to get back an check – have I really followed this, is it the way I thought... it is important that you spend much time on the first design... then again, the design is not nailed in rock, indeed, it can be changed... I can imagine that this is how it works in companies that do much software development. [S07]

S19 describes that it matters if it is hobby projects or if it is a larger projects, because they are different. In a hobby project you can work more directly with the programming, whereas in a larger project, you need to be more careful about the planning before you start to program:

> Then you sit and think about what different parts are needed, modules or classes... and I think about how they are put together... but when it is kind of a hobby project, you can

> start to type the code sooner than you are supposed to. Usually you should plan more and that. Like now, in the thesis work, I planned very carefully. We planned for two weeks and we had a whiteboard and we came out with a lot of ideas and details, and it went really damn good actually. So it depends if it is a small project or a large project... [S18]

Likewise, S17 have a similar understanding that there are differences between working professionally and devoting oneself in a hobby project. In professional contexts, a design phase is needed, where the developers plan what to do before they implement the software. The description also adds that the program should be tested and that potential bugs should be corrected:

> Well, it depends on if you work professionally or only with a hobby system, but in any case you must have a design phase where you in some way sketch more or less carefully, sort of, what you are going to do, and stuff. It is very much about design, implementation, and then testing and debugging and that... but if you only are doing a hobby project at home it becomes more of that you are not so careful and not so formal, sort of. [S17]

Several informants describe a different way to develop software. It is about working with prototypes that you gradually change instead of planning a complete solution before starting to write the program code. S12 says:

> Yes, but my way of developing the programs is to... you make a prototype first and then you have it in mind when you try to design and improve, and in the design book we studied, it was supposed that you should make prototypes and then rewrite the code and keep on doing it all the time. Instead of having a gigantic UML diagram and then follow it... the basic idea is that you make prototypes and they you improve the code and make a prototype from the improved code. [S12]

S16 describes that a new methodology has come that differs from older methods. Using the new method you produce a usable prototype that you continuously develop and refine in short intervals. This is far from the waterfall method, where everything is completed in a current phase before shifting to next stage. To S16, the new methodology is preferable because he does not like documenting – nor specifying:

> It is very much scrum at the moment... it is a rather new concept... besides, you get a... continuous prototype that is usable, sort of, unlike... the big companies are using... more this waterfall method where you heave things to next stage and next stage... This is more round, because you work around all the time, sort of, weekly instead... the waterfall method is very directed towards writing documentation... or specifications. Naturally, it is good to specify so that you understand how it works... but I think the trial and error method is more fun, sort of, that you program instead of sitting there and thinking out how it all should work. You always have a picture in the head how you want it to work between the different parts anyway... [S16]

Also S13 describes a prototype driven iterative model and focuses above all on that the programmers are going to be able to take part in the entire process:

> I think you work in iterations, that you make a first software and you present it and see, are we on the right road? ...that you can go around in the iterations, that you develop all the way through, so that is the project method you can say. I have learned many process methods but I think this is the most reasonable, that you develop programs so that the programmer can follow all the way too. Surely, the programmers are participating from the beginning, and also maybe in the discussion of what is possible and so on. [S13]

The descriptions, or understandings, in this category of description, can be summarized as that there is a process that is about working out, or designing, a solution to a problem following some kind of methodology where things are carried out in certain steps in a certain order. The methods described are either an iterated step-by-step refinement of a prototype or a two-step model where you first plan for the code and then you implement it.

5.3 Category 3: Designing for the future

In the previous category of description, software development is described as a more or less structured and methodical process that focuses on producing a computer program. The third category of description, designing for the future, includes the understanding in the previous category and adds descriptions that open a new dimension of software development, namely an awareness of a number of qualitative properties the design should have in addition to being the solution to a specific problem. In this category, the focus is on inner quality requirements on the software, including that the design should be extendable, durable, reusable and maintainable.

S09 describes that it is important to have a well thought-out (inner) design, facilitating the possibilities for further development of the program:

> If someone would like to develop this further, it is important that I have thought-out the design in a way that they can see what I in fact have done and where they can add to it and how... well, to have kind of a structure on the construction itself... that it is not a heap of boards in which I am hammering nails at random, but rather that I actually think that this is a wall, here is a floor, the wall is dependent, or that the roof is held by walls, and such. [S09]

S01 describes how his understanding of software development has changed from being only writing the code

into now include that you need to make a good design that makes the programs durable and changeable:

> This is something that has really changed over the years. Because, when I used to hack some code at home in front of my computer, in principle it was like that, that you started from an old project and improved it, or you started from scratch and in principle sat down and wrote code right away. In recent years I have realized that it is not a practicable way if you are making, above all, durable programs that can cope with changes and can be separated in different paths along the development process. So, I have first of all started to realize the need for good design, good program design, good code structure, and I feel this is really something that the education has helped me with, that I did not know before. [S01]

Early in the interview, S11 mentioned the concept design and was later asked to talk about what good design means. He explains that his understanding is that it is a durable design that is possible to understand and build further on. He describes how this is based on that the components of the design are separated in well defined cells with only a few connections and that this facilitates exchangeability:

> Good design, that is a durable design that is possible to, that is, first of all, easy to understand and secondly, it should work when you build additions to the system and then it should also be relevant, you should not program more than what is needed, you know. And then it is cohesion and coupling... that relevant stuff is put together and these relevant things has very little connection to others because you have clusters, kind of. Well, but the typical tourist sale thinking, you know, that you have small cells and if one goes down it is easy to exchange it, kind of. Roughly, that they do not have so many connections to each other. That's right, that the same things can mange, code that does the same things should be put in the same classes and should have low coupling to other classes so they are easy to exchange, not many-to-many relationships. [S11]

S12 talked about an agile process where it was important to develop a prototype quickly, but he would like to go back and refactor the code in a later stage to make the code reusable in the future:

> Well, I guess it is above all to make the code simpler and shorter and, a bit of this future thinking, doing it all with modules and plugins and such. [...] Well, that you think: can I use this code for something else? In that case you can break everything down in smaller methods to be able to reuse the code later. Perhaps you make more of these frameworks or libraries, and then you do the ordinary code just for the specific purpose you are having. [...] Well, it is to be able to reuse it in later projects. [S12]

S01 describes the importance of "seeing between the lines" and be able to see the whole; that things will change in the future and that you have to design software in a way that it can handle changes:

> Right now, I'm taking a class... and the purpose of the class is to learn to evaluate and make good designs ourselves... it is very important... because it is easy to learn to do the handcraft, the mechanical handcraft, but perhaps not the preparation for it... already in the beginning when you sit down with at specification or a requirements list, that you kind of see the whole, and also notes that it may come in things between the lines that change the project in the future. And that you have to think about future fields of application for the program you are developing, to make it easier for one who continues later on... that you have a core that is very flexible... that you can build on and extend in many different directions. [S01]

According to S02, there are two different levels of design. The lower level is very important and it deals with how the program is built internally with the purpose to be maintainable, changeable and correctable. The higher level is all about how the program can be usable and understandable for a user:

> Well, on the lower level you must decide how, how the program itself should be constructed on the, that is to say on the slightly lower level which only the programmer cares about and the user will not see at all. And that part of the design can be very important, because if you design it poorly, then it can be extremely hard to maintain later. If you want to make changes, bug fixes or what ever, it will be, could be enormously hard if you design it wrong [...] and besides, if later on you do a bad design at the higher level where the user uses the program, it can be completely wrong when you use the program, that it cannot be used because of its bad design; you simply don't understand how to use it. So there are two different aspects to think about there; that you design the underlying layer so that it is possible to change and that you design the higher layer so it is possible to understand what you are doing, what you are supposed to do in order to get something done. [S02]

In summary, this category of description comprises the understanding that software development is solving a problem by designing the program methodically and ensuring that it is constructed in a way that makes the work effort spent on design and coding reusable in a future perspective; that the program is readable for future programmers; that it is possible to exchange components and make changes. Software development is

described as designing software with respect to various quality criteria which lay beyond the horizon of what functions the specific application should provide. Software should be planned before it is written – or should be refactored. Software should be reusable, modifiable and endurable for future needs.

5.4 Category 4: Understanding what is needed

The previous categories have more or less described software development from a point of view that emanates from the problem and how the software's code is produced. In the most advanced category, software development is described from a perspective of the whole that is close to a professional approach. This way of seeing, starts with the customer, the users, their requirements, needs and wishes. The customers gets a central position and have great importance for the development process, and it is a part of the developers' job to help the customers to know what their requirements on the software should be. It is also apparent that there are budget and time limitations to be aware of in a project, and after the software has been delivered to a customer it needs to be maintained.

S03 describes that software development is to evaluate the customer's requirements and then find a suitable solution:

> It is to evaluate the requirements of the customer, I guess, and from that give, well, some reasonable solution that follows the requirements. [S03]

According to S02, software development is a long process, and it starts with the requirements from the customer:

> For me, it means all of this long process from the point that you have the requirements until you have delivered, you know. And then it is... after that you must keep on maintaining the program... In the beginning you get some kind of requirements from the person who orders the program... and then you have to decide how you should work... and try to keep it within budget... [S02]

As S04 points out, the first thing is to understand what the customer needs, and there should be a requirements specification:

> Well, that you, first you look at which needs they have, what, what should be in the program so to speak. Maybe some kind of requirements specification is preferable. [S04]

S20 says it is essential to learn what the user needs, and one way to achieve this is to interview the users:

> Software development means, it is, above all it is that you must investigate what the requirements are, what it is that should be solved to start with. Well, find out who is going to use it and try to get some information from interviews, in order that you do not start programming right away... [S20]

If you want to introduce something new to the market, S13 says, software development is much about exploring the needs, the requirements and the contexts of the end-users. Programming is something that comes in second:

> Well, simply it is all about need, that there is a need of something, and then that someone can express that need in a good way so that another person can understand it. And then that someone can handle the wishes and then introduce them to a programmer... a requirements specification that is passed on... for example, that you want something new that does not exist on the market... it is required that you make user investigations... to see if there should be a system at all... you test on different people.. you check the context and the environment where the system is going to be used... see what needs the people have, if there is a need on the market at all... there are design principles... when it comes to ergonomics, well, context, environment and things like that. Simply that you should have a user focus... because programming... comes rather long behind in the hierarchy. [S13]

First of all, it is all about coming to an agreement with the customer about what they want and if they can pay for what the development will costs, says S06. Then you keep in touch with the customer and deliver in portions. The program will need maintenance until it eventually will be taken out of service. Software development also means that you develop personally and learn from your experiences:

> The customer. First you should find out what they want, and then the customer must decide... if it is worth the money... to find out that people really want something. There is when the program development starts, I guess... it is the contact with the customers... and then you have to go through certain cycles... Scrum perhaps, there are some method that says that you should deliver a piece at the time and see to that it works a bit, and then you go back and make changes. I is a long process only to develop the program, but there are also a certain measure of system maintenance... and that goes on more or less until 15 years later when the program dies... Then personal development is involved... mostly it is rewarding. If I make a program I want to believe I have learned something. [S06]

S19 describes software development as a mission in which it is included to find out and formulate what has to be done before you implement and test the system:

> Oh! It feels like a rather big concept, you know. It is everything from the beginning to the end. Well, you have some kind of mission that you are supposed to accomplish, and one

> part of the development is, what should I say, to formulate or investigate what you are going to do, and then to implement it, the thing itself, and test and redo it. [S19]

S14 describes various aspects that you must pay attention to if you for example are going to develop a system for a company. It concerns collecting information from many involved roles related to the project, for instance from the end-users, and then investigate which requirements there are. The end-users are important, but you also have to adapt the project to the budget and time limits:

> Then, I guess, they should contact those who are involved in the different roles; what is behind the technical so to speak to investigate the requirements, as well as those who will use the system to adapt it in the best way. The users are very important, see... Budget and those things are also very important. Well, someone else must identify the requirements for the system and, kind of, start from those basic requirements you have in the beginning and how you can develop it and what the time frame is, and then, well, interview people and see their point of view and how people use the system today. Do these user scenarios and similar stuff... There will be, well, rather many aspects to consider, really. [S14]

S13 points out the importance of having a continuous dialogue between the programmer and the customer so that what eventually is delivered really is what the customer wants. This is why it is important that the programmer participates in meetings with the customer from the very beginning:

> Above all, it is important that the programmer knows what the customer wants... it is this dialogue in the project that must be present all the time, and that is why I think the programmer should be engaged in it from the beginning... when the discussions take different turns, the programmer must be there to say stop it or to encourage... We have seen examples of what can happen when there is one sitting here... and there is a programmer sitting in the other end who says: well, I can give you a house, and they got a house, only it was not that house they wanted to have... I believe it is important that the programmer is in it from the beginning, and is allowed to design and think as well, not only someone who does what another person says, but is involved in a process. [S13]

Altogether, this category of description gives examples of a comprehensive understanding of software development in a professional perspective where the focus is set on the understandings between customers, end-users, developers and programmers.

6. HOW THE INFORMANTS VIEW SOFTWARE MAINTENANCE

The phenomenographic analysis of interviews revealed four distinct categories of ways of perceiving and describing the phenomenon of software maintenance.

In the first category of description, the different descriptions focus on that maintenance is to fix bugs in the code and create patches that handle these errors. These descriptions often express a view that maintenance is a boring job that you would rather not deal with, but the understanding that it can be challenging and stimulating to resolve a bug also occurs.

The second category of description takes on a clear time perspective and a user or customer perspective. After a period of time after delivery of software, then the customer discovers that there is something he or she wants to add or change, but it also may involve errors. Maintenance is, then, to address these concerns.

The third category of description, sees in a way, the software in an even longer time scale. Here you can see software in relation to external factors that can change over time. For example, a new operating system or a new release of a database can force adjustments in the software.

Finally, in the most advanced category of description, software maintenance is perceived as a continuous, ongoing effort to change, debug, add features and customize the software. It is something natural which is always there for better or worse. It is something you would expect to be working with in the future.

A summary of the outcome space for the phenomenon "software maintenance" is shown in Table 3.

6.1 Category 1: Handling bugs

Among the informants in the investigation is clearly a way of perceiving the phenomenon of *software maintenance* as an activity associated with improving a software that has been delivered to a customer, by addressing the errors, bugs, that the user has discovered. S07 explains also that errors are something you would expect of a newly developed software:

> Yes, maintenance of software, that's... Firstly I do not think that you can release a software that is flawless from the start but of course you have to patch it and you have to upgrade it to fix any bugs the user finds. Because it is, ultimately it is he who will use the program most. It is never the programmer who uses their own program, but mostly it's always the customer, so to speak, and it's he who discovers the errors and shortcomings. So, maintenance of software is improvements in the first stage, I guess... [S07]

Several informants describe expressly or implicitly that maintenance is something different from development, for example, that maintenance begins after the development and that it might not be the same people who are involved. S20 points out that maintenance of software means that you collect and manage feedback about errors from those who use the software in the real world and that there are people who have the task of remedying the defects found after delivery has taken place:

Table 3: A summary of the outcome space for the descriptions of software maintenance

Category	How *software maintenance* is described
1: Handling bugs	Software maintenance is about collecting bug reports and taking care of the bugs and patching the software after its delivery to make sure that the system is up and running. The programmer who makes the corrections might have to get into unfamiliar code and look for details.
2: Change and add	Software maintenance is about handling corrections to make sure that the system is running and it is also about implementing the improvements, the changes and the additions to the software that the customers or users probably will ask for a while after delivery.
3: Adapt and update	Software maintenance is about corrections and additions in the software, and it is also about adapting the software to changes in external factors that impact the software, for example new versions of software libraries, data bases and operating systems.
4: Continuous work	Software maintenance is a natural part of the job. It is a continuous work effort that is required in order to keep a software's life long by making adequate changes in the code. It is costly. There can be a contract between the customer and the producer that regulates this work.

> Software maintenance means that you also get... one thing is that you have a good way of collecting comments and bug reports from the users, that you can get reports from when the program has been used for its real purpose so to speak. And that you have a number of persons that has as their job to, well, update the program. First, I think of removing bugs that may have been shipped, but also perhaps to add, well, new features and that. But first of all I believe that it is about removing bugs that are discovered afterwards. [S20]

Moreover, it is common that the view of software maintenance, in the sense of dealing with bugs, is also linked to the perception that maintenance is not as interesting as the development process. For example, S08 says that in order to correct errors in a program you may need to familiarize with code that you have not written yourself, and that feels awkward and is not so stimulating:

> Software maintenance, it feels like, software maintenance, the phenomenon feels like something tiresome. It feels like much of bug corrections, it feels like that. You just get it in your head, you know, that it feels like something cumbersome, only because you probably have to get into other peoples code, if it is not your own, and I don't know. It feels like it is a pretty, probably a much less stimulating part of the coding stuff, some way, well, I don't know. [S08]

Furthermore, S08 describes that maintenance is very boring because it is about to engage in correcting errors and searching for small details:

> I don't know, but it feels like maintenance of, maintenance of code, no, but it feels like, then you are supposed to correct lots of, lots of errors and it feels like, that if you are doing something that is very messy when you are coding, you will get to a limit when it starts to be, when it has become so messy that you really are getting tired of it. And it feels like if you are going to maintain something, then you will start right there from the beginning. Then you are sitting with something that is damn boring and you are kind of trying to look for things you are going to change. But I don't know, if perhaps I should talk more about maintaining a program in the sense of developing it further. I don't know, but that does not feel like maintenance. Maintenance, to me it feels only like correcting and it feels like the absolutely most boring part of coding there is, and just that you have to, it is all about sitting there looking for those tiny details, and search for errors and that, yes. [S08]

However, as S15 says, this is perhaps a task that you get into the bargain:

> Yes, no, but I guess It could be, it could be interesting to be part of the whole process from the customer and then to implement it and then deliver, but about the maintenance, bugs and such are not as interesting, but it might be included, maybe you get it into the bargain. [S15]

And even though the common perception is that it's more fun to work on developing new things than to fix bugs, S15 tells that it may be satisfying to solve a problem so that the program works better:

> Yes, but it is to fix bugs and ensure that everything runs on, isn't it? So I guess it is, well, now I have not much experience with

> it but, but then it will be more like to keep the system running. It's almost more fun to come up with new, like further development, the development of new things. But then, although it can of course give a satisfaction if you manage to resolve a bug so that it works much better then of course there is also a kind of satisfaction. [S15]

6.2 Category 2: Change and add

The second category of description widens the perspective in the sense that maintenance is described as something more than only beeing a bug correcting activity. This wider perspective introduces the understanding that clients may come back in the future and ask for changes in the software, or make suggestions for adding new features to it.

One of the informants' ways to describe software maintenance in this category is by saying that programmers add new features to an existing and delivered software, or to correct and modify some parts of the software. S12 and S09 says:

> Yes, that's everything from testing for bugs to correction, and implementation of new features. [S12]

> ... it is not just fixing bugs, but you need updating and extending, I guess, but that's also part of maintenance. [S09]

S05 says that failures and errors are discovered only after the user has had time to really use the software, and it is only after a while that the user comes back with suggestions for changes in the software:

> Maintenance, yes, if, after a while when the user has really used the software, they discover little bugs and want improvements and additions. Then the programmer can maintain it and fix errors and add things that they want put in. [S05]

S03 says with a similar outlook that software maintenance may be about to change the software through the introduction of optimizations and extensions; that the client wants to improve something that actually works, but needs work even better:

> Yes, optimizations, extensions. Now, we have not done much to expand programs, but I can imagine that there is much about it ... Yes, I think it's very seldom it is that, if you have, if the customer is satisfied from the beginning, comes back and says that this was not quite what we had expected, but it's probably more that they want something more, or something changed after using it for a while. [S03]

S02 and S18 describe that new requirements on the functioning of software may arise afterwards; that the user gets ideas for changes that would improve the user-friendliness:

> If you have built a program that you then run for a while and decide yes it works as it should, it may of course still be so that more demands on the program can occur. And, in that the requirements occur, it can be characterized as maintenance, adding them, the functions. Thus, for example, that you have a text editor that you can not undo the work in. It's still working, but then, it can be that you want to patch it and make sure you can undo, so of course it can be characterized as maintenance even if the program continues to [work]. It is kind of finished but you want to add something or you want to remove some function, then you maintain it. Or if you find a new bug that you want to fix. It may well be that new bugs will be created when you've fixed a bug. [S02]

> Fix bugs, find them. When those who use the program realize that, yes this is good, now, now I want to be able to do this a little different because it is difficult to specify ... because you may sense that it was perhaps not quite that but something like it, then you get to change those little things and also to make it easier to use. [S18]

S06 starts his description with telling that the operation of machinery must be ensured, but decides that it is not included in software maintenance, which he then describes in a way that is similar to what is narrated previously. But in addition to describing how customers want to expand the program with things that they were not aware of before they got the chance to test it, he also explains how the new features in turn can lead to new errors to be addressed:

> Maintenance of software, it's partly to check on the machines running the software actually works and if there are programs to be running around the clock you need to have back-up and stuff, though, it's perhaps more hardware maintenance... I have been doing very little, but I suspect it is about, implement new features in the software that the customer was not aware that they needed until afterwards and then, and then clean up all deficiencies are due to them new functions and it can also be that, I think, there is probably some extreme cases where you'll find things that can break in the software, then it is kind of logical errors, but I don't think you actively are looking after them in the same manner. [S06]

S10 describes in the following quotation that maintenance of software is about *patches* and improvements. Later, he gives his description of what he means by patching where he says that software in some way must be prepared to deal with patches, so that some parts of the software should be replaceable:

> It could be, for example, patching and improvement [...] How to update a portion of

> the code or the entire code, but then one must of course have a pretty good design... If you are only updating a certain part you must have arranged communications within the program that can handle replacing parts, and equally if you are going to build extensions, it requires even more, that it deals with additions too, so there is another level of abstraction. You have to move from that it just works into making it work with, or think out, additions. [S10]

The way of seeing software maintenance is that it is about correcting the so-called bugs is included by the understanding that it is about to introduce changes, improvements or additions to the software. We could see this in the quotes in this section and, furthermore, we can reason logically and conclude that both approaches is to modify a software program, but what is described as the correction of bugs is a more limited approach.

It is worth noting that both S03 and S06 say that they have not been doing much with expanding programs.

6.3 Category 3: Adapt and update

This category of description opens up a new dimension to the perception of what software maintenance means. It's about understanding the software as an integral part of a larger system and that the software has dependencies to changes in external parts of the system, or that the software may require a change in the external parts of the system to work better. The informants describe such dependencies and changes in operating systems, databases, graphics libraries and hardware, but also changes in the behavior of users. This way of perceiving software maintenance and the underlying causes for maintenance are more advanced than the previous ones because it involves a more complex view of the software and the context in which it exists. This way of seeing also includes the understanding that corrections, changes and additions to the software needs to be performed.

Here follows a few quotes from informants who provide examples of this approach. S03 and S17, for example, describes how a new version of an operating system can lead to a need for adaptation of a software:

> Yes, it depends on which program it is but it may be that they added a feature, changed the layout of any quick keys somewhere for some function, yes, then it can surely be like that the software must be optimized to a newer version of some operating system or something similar, because it also happens of course. [S03]

> As I see it, it's to search bugs, and receive information and feedback from users and try to adapt the product, and then also to keep the product up to date with changes in hardware and, yes, the software also, So to keep it compatible with new operating systems and such. [S17]

S04 describes how you can get a faster system, for example by upgrading to a new version of the MySQL database manager or indeed look at different parts of the system so as to obtain a more efficient and safer system:

> First you will have like, optimization stuff maybe, I mean both the program itself but if the program may be using MySQL ... that you may upgrade these versions to get a faster system in some way ... external parts in some way, but also ... that you might come up with new solutions to make your system better ... to get a more effective, safer or something, system... Or they want more features, things that you may have missed. [S04]

In a similar manner S10 describes a link between maintenance and external software libraries. In this case that the release of a new version of the OpenGL graphics library had an impact. In the quote the "users" are the programmers who use the graphics library in their software and "those who do maintenance" are those who maintain the graphics library:

> ... the maintenance becomes ... a bridge between the different skills that develops ... OpenGL is a graphics, yes, a graphics library or what to say and there are somethings that surprise me. They had a released a new version and then everything was so different ... It feels like when you saw it the first time that they put a spoke in the wheels of their own ... those who do maintenance and improve things, they must convince those who use it, perhaps more than the first time ... [S10]

S16 gives example of that software may need to be adapted to changes in the hardware to effectively take advantage of hardware improvements:

> ... there will be new, I mean, a physical hardware that must be handled in the program or else it kind of crashes, and perhaps it is so if you update your computer into a giant computer and use it only for a tiny little process, and it's also a bit sad because when you have so much power in your computer you want to use it well. [S16]

S14 describes how a system might have to be updated and adapted to changes in the scale and scope of the business, *including* taking care of bugs in the software:

> Maintenance of software, it is, well to update and adapt ... Yes, depending on whether you, what needs you have. If a small business expands, you might have to adapt the systems and the software you have to be able to use it optimally, but, yes, but then it might, yes, maintenance, yes, if anyone, I guess it also includes that if you discover bugs and such, that you take care of them, yes. [S14]

S11 describes how the software in the long term may need to adapt to changes in system load, and that parts of the system need to be updated and adapted to new versions of external software libraries:

> Maintenance of software, I have absolutely zero personal experience ... But, that's classic like that, you have a system that can handle a certain load, and suddenly they found themselves 20 years into the future and it is triple, or quadruple the load, and you realize that it really is not working. These algorithms are no good, often perhaps it is in the hardware, but anyway you can of course detect that it is functionality that is missing and must be added, and maintenance is for adding that to get the system to continue functioning. .. But to maintain the code, well, I don't know, replace old crummy methods with new more effective ones, old crappy libraries with new shiny ones. Possibly there, one can call it maintenance possibly, to streamline, yes, maintenance to streamline... [S11]

S13 explains that software maintenance may relate to changes in people's habits, for instance how they handle payments, and that it is about modifications of an existing system – not building a new system:

> When you need to do it is, well, simply when it is not needed to make a completely new system. When there is a system that is well structured in its fundamentals, but which may need modifications to make it fit into today's society, or how to say, today's time, if there has been any change in the world that requires that a system must be changed ... We may take trains for example, train tickets, people don't have cash anymore but they have cards, that's one example, I mean, there is a change in time, that people no longer have cash, parking meters ... [S13]

In some of the quotes that have been used to describe this category, one can also deduce an understanding of that software may have a long life and that you can see it in a time perspective. The next section describes this in more detail.

6.4 Category 4: Continuous work

This category of description gives expression to an understanding of software maintenance as a long-term commitment with the intent to ensure that the software may have a long life. In some of the informants' descriptions, there is an understanding that maintenance is a completely natural part of the work with software or at least that it is something they can expect in the future workplace. The descriptions often contain words such as "continuous", "maintenance", "responsibility", "always", "keep up" and "costly ". This view includes the previously described, but then adds a clear idea of that a program needs to be maintained over a long time.

S16 describes that a computer program is not complete in that it has been delivered, that maintenance is something continuous:

> So, maintenance is of course something continuous, but in fact a program develops, just because you're done, have finished the program and delivered it, it's not a finished program, in fact, for there will always come new things you'll need to take care of, for example, new users who do things that you have not thought of before... and a continuous maintenance is, it is of course to handle the new, yes, the new conditions simply. Even when there is new software, when, there is nothing which requires that the program can deal with a new motherboard, for example, or a new graphics card. [S16]

A similar approach is described by S19 who says that you will never be finished with the software:

> That is when, when you are finished, in quotes, and, yes, it is never finished, because first you will always find errors you have made, so you have to constantly correct them and ensure that there is nothing new that breaks, and test to make sure it was correct, and well you know, the client may want, have identified something they want fixed, and then you should see to that it gets repaired and check that it works. Then that goes around, and then I suppose it is if the client asks for new functions in the program, that is probably also part of maintenance, and ensure that the documentation is, or that you have it, or update it all the time. [S19]

Initially S07 describes software maintenance in terms of bugs, patches and improvements, but after a while he gets into a different way to describe maintenance which has not previously been mentioned, that the operation of a system can generate large amounts of data that needs to be cleared:

> ... I do not know if it belongs to the software maybe, but cleaning up databases, tidying up kind of, and maintain in that sense. There might be, there could be garbage in the system which need not be there, that you need to remove, and then you may well wonder if maybe it is an improvement to just remove it so that it is not stored at all, but perhaps there are reasons, that there is an old system that works towards the data, so that is also software maintenance, to clean up in database structures, to fix these peaces, because if you get untenable database, it will also slow down the program. [S07]

S04 describes his own experience of maintenance. After having developed a "website" for a large company, there was a constant need to maintain the page and work with changes to it:

> Yes, but it's almost always like that when you are doing a website, I think. I've made one to a company here in ... a large company... All the time it has, you must constantly maintain the website... yes, but now we need, we would

like to have this and that, well then you have to add that... Websites, perhaps they are a different compared to Java programs, pure programs, I think. That websites in particular... yes, there are much changes in websites. [S04]

S09 describes software maintenance as a continuous work effort. Even if the software meets its goals, it will never be complete and perfect:

> Actually, it would probably be to have some sort of continuous bugfix, so to speak. Just because the code is finished and you have achieved your goals it does not mean that it is complete, I think. Rather, you have to constantly maintain it, so to speak, check if there are vulnerabilities in it perhaps, and then fix them. Because a code will never be entirely perfect so to speak, at least not something very big and extensive, so yes, continuous testing and dealing with misses and such things. That it is maintenance for me anyway. [S09]

The informants were asked to tell about how they think they will be affected by maintenance in the future. And in response S09 says that you can have a job that includes software development, but also a responsibility of the product after it is developed and then maintenance will be something you will deal a lot with, or "get stuck with" :

> Yes, that is, if you have any job that involves developing where you also have some sort of responsibility for the product afterwards, then maintenance is something you certainly get to do a lot. Say you are developing an application, there is of course some requirement that you also address misses and security gaps, and then patches it, and such. So maintenance is probably something you will always get stuck with, so to speak. [S09]

Moreover, S07 describes how a company can contractually commit to maintain a software product and thus has an obligation to attend to the program:

> And then that maybe you, some companies may require a maintenance contract for the product in question. It depends of course on what kind of software... That the company has an obligation to maintain the program as well. [S07]

S10 says that the reason for maintaining software is to allow the software to keep up with developments:

> Yes, to keep up with the latest developments, usually they would like to do that, of course, so that it becomes useful, it feels like there is like a curve where certain sensitive areas has a very steep curve on the things such as getting old with time, so they must of course be maintained more. [S10]

S12 gets into the problem that all of the changes and additions eventually, after a long period of maintenance, may lead to "chaos" in the code, and at that point it might be time to do a complete review of the code:

> There are some companies that are doing it forever [laughs] I do not know, perhaps it is wise to hold on until it starts to get too chaotic, then you can of course try to do some refactoring and come out with a new version that has a better code and better structure and is faster. [S12]

In summary, the most far-sighted approach to software maintenance among the informants, is that software simply needs to be constantly cared for, after it was prepared, for it to be timely, error-free and adapted to external circumstances. These are tasks that can be expected in the future profession.

7. IMPLICATIONS FOR TEACHING

Research on computer science education issues has mainly focused on the problems that many beginners have when they learn to program. Less attention has been drawn to the more advanced students and how well they are prepared for future careers in the software industry. This study was conducted to contribute to the understanding of students' ways of looking at important aspects of working professionally with software in the industry and to provide suggestions on how to use this knowledge in the classroom.

The information booklets on courses related to computer science often stress the vocational aspect – that after the studies, the graduates can be employed, inter alia, as software developers. And when the informants in this study talked about their future plans, it appeared that everyone wanted to engage in various forms of professional development of software in the industry. This was important to know because it strengthens the purpose of the study and makes the phenomenographic results more relevant when you want to use them in teaching. Overall, it is reasonable to assume that students would be disappointed by a purely scientific approach to computing. Nevertheless, computer science educations have been criticized for having taken on too much of a profession-oriented attitude with elements of Software Engineering [13].

A reasonable question then becomes whether and how to maintain a scientific focus while trying to prepare students for a profession? The industry trend is sensitive and will probably always feel that education is lagging behind and will try to influence training. New popular programming languages and new working methods for the development of software are incorporated in the training, but when students have entered the labor market, perhaps the interest in these particular skills is gone.

In my opinion, all students in various computer science programs need to become aware of a number of important aspects of professional work with software; however, I am not saying that we should give them purely vocational training. I assume that we still will use traditional teaching, but I argue that we can take

advantage of the results from this study. The outcome spaces can be carefully analysed to find dimensions of variation which can be used to help learners discover new ways of seeing.

7.1 Variation Theory

This study stems from the basic phenomenographic premise that there is a limited number of qualitatively different ways to experience a learning object in a certain group of people. Moreover, phenomenographers mean that an important prerequisite for learning is the ability to discern critical aspects of the learning object. Variation theory shares the basic concepts of learning with phenomenography and provides a theory for how to give conditions for learners to identify critical aspects and thereby get a richer understanding.

Above all, what should be learnt must exists in a context that is meaningful to the learner, that you have the proper relevance structure [9, p.140, p.155]. Then instruction can be enhanced by helping students to discern the critical aspects of a particular learning object. However, a mere listing of facts is not enough to ensure a rich understanding of complex phenomena; the learner must be aware of the different aspects involved and how they interact. This can be achieved by introducing carefully selected variations in what the learner takes in through his or her senses [9, p.145, p.152].

The idea is to highlight relevant *features* of a learning object, for example by altering the "value" of some aspects while others are kept unchanged. A *dimension of variation* is spanned by all the possible values for some associated property or aspect of the learning object. Empirical research on teaching has been able to discern four different patterns of variation that use different combinations of invariance and variability: *Contrast*, *Separation*, *Generalization* and *Fusion* [10]. In addition, the variations will help the learner to break the natural attitude, which is required to start a reflection [9, p.148].

By analyzing the phenomenographic outcome spaces it is possible to identify critical aspects of learning a specific learning object, and then to find the corresponding dimensions of variation. Previous studies give suggestions for teaching based on phenomenography and variation theory [2, 15]. In addition [15] reports successful results from a pilot study that used the suggested variations and then assessed the learning experiences.

In the following two sections, results from the present study are used to identify key aspects needed to get a rich understanding of software development and software maintenance. Similar to [15], the description categories are examined to identify dimensions of variation which could be used in teaching and some proposals for explicit variations are suggested. In this case, however, it is the professional perspective that determines which aspects and dimensions of variation that will be addressed.

7.2 Variations for software development

Based on how variation theory describes the mechanisms for facilitating that learning takes place, the following discussion analyses the outcome space for ways of understanding "software development", outlined in Table 2, with the aim to identify dimensions of variation and suggests how to use the dimensions to help students to see important aspects. The first two categories of description do not particularly focus on professional aspects of software development, and hence, they are not further addressed here. Category 3 and Category 4 represent a richer understanding of software development seen from a professional perspective. In fact, these categories connect to advanced ways of seeing software maintenance, which indicates an integrated understanding of the professional context. The suggested dimensions of variation for Category 3 and 4 are listed in Table 4.

From a professional perspective, Category 3 addresses an important aspect of software development: that developers should design solutions that are sustainable in the long term. One reason that it can be difficult to realize the importance of this aspect of software development is the relative lack of positive response compared with the satisfaction of making a program work. It requires a comprehensive understanding of the conditions for software development as a business over long time to understand the point. The reward comes when it is time to develop something new and it is discovered that many of the components that have evolved in the past can be reused, or when old deployed software easily can be adapted to cope with new realities. This requires well-designed and readable code, and proper documentation.

There are several potential *scenarios* of what may happen in the future which are values in a dimension of variation that can support and enable the understanding represented by Category 3.

In addition, there are many aspects of software quality that associate to software sustainability. Four important software *quality* dimensions involved in software designs are: (1) code adaptivity, the ability to adjust code to new circumstances, (2) code reusability, to what extent existing pieces of software can be used in, e.g., other projects, (3) design understandability, the degree of comprehensive design documentation, and (4) code readability, the possibility to understand how the source code works. Hence, each of these qualities can be varied in separate dimensions of variation.

Now it is time to discuss how the variations in the five proposed dimensions of variations can be done. The idea is to expose students to different scenarios and a software application with varying design qualities. However, since this may be too complex, the number of simultanious variations should be kept as few as possible. Therefore, what the application *does* should be held invariant while the *quality* aspects of the software design should vary – but not all at the same time. Hence, it is suggested to use two different scenarios and two examples of different quality values for each dimension of variation. Here is an example of values:

- Future scenarios:
  - From a new requirements specification, it should be investigated which parts can be reused from the existing application
  - Data has been migrated to a new database system which caused the existing application

Table 4: Dimensions of variation related to software development description categories.

Category	Dimensions of variation
3: Design for future	Dimension of future events that can affect the software Dimension of adaptions and adaptivity of designs Dimension of reuse and reusability of designs Dimension of comprehension of designs by documentation Dimension of code readability and ability to read code
4: Understanding need	Dimension of end-user and customer aspects (eg competence) Dimension of teamwork aspects (roles and communication) Dimension of economical conditions and time limits Dimension of domain context and knowledge

to malfunction and hence it must be adapted

- Adaptivity quality values of software:
  - It uses polymorphic structures, protected attributes, and is highly "parameterized"
  - It does not use inheritance hierarchies or interfaces, and is highly coupled
- Reuse quality values of software:
  - Its parts are designed to be generic and are placed in public software packages
  - Its parts are mostly specific to the application and are not separated
- Understandability of the design:
  - It is richly documented by text, listings and diagrams explaining classes, interfaces, relations and libraries
  - It is documented only by the sparsely commented source code
- Readability of the source code:
  - It is organized in packages, one class per file, consistent indention style, commented classes and methods
  - It is put in one single file without any comments, using funny names and following no code conventions

The pairs of values suggested are inspired by the variation pattern *contrast*; that a certain quality is hard to experience without experiencing a mutual exclusive quality.

For each of the scenarios, the possibility to accomplish the task can be discussed, evaluating different combinations of quality values that vary one at the time while the others are held invariant. This is an example of the variation pattern *separation*. The analysis could be a theme for a lecture, it could be handed out as a larger team assignment or it could be a theme running through an entire course.

If a teacher wants to go beyond discussing these aspects theoretically, the consequence would be that at least eight different implementations of the application and several versions of design documentation need to be prepared and this may be too much for both the teacher and the students. One way to reduce the number of combinations is to let pairs of dimensions vary simultaniously, especially if it is not a particular goal to contrast them against each other. For instance, code reusability could be paired with design understandability so that the example of highly reusable code is accompanied by a comprehensive design documentation, and vice versa.

The variations can be done in different ways, but the important thing is that the students get the chance to experience how different qualities of software impact the possability to work on the software in the future, which is crucial for professional developers.

Category 4 represents an overall understanding of software development which includes the relationship with customers, end-users, economical conditions, deadlines, teamwork and communication aspects. Four dimensions of variation can be used to reveal these aspects: (1) the dimension of end-user and customer aspects, e.g., their different needs, circumstances and competences, (2) the dimension of teamwork aspects, such as having different roles and communication, (3) the dimension of economical conditions and other constraints such as time limits, and (4) the dimension of varying domains and domain knowledge needed.

These non-technical aspects, such as taking a customer perspective, are parts of the development process which do not explicitly concern the program code, and this can sometimes be difficult to teach to students who may expect a technical content. Nevertheless, these are very important aspects for a prospective developer to realize and it is not always the case that our students have this understanding:

> Teaching software engineering convinces us that most students, even at senior level, have little understanding of real-world customers using their code to achieve useful work. [17]

A suggestion for helping students understand the complexity of Category 4, is to engage students in larger development projects that mimic professional software projects in different domains. These projects should include people playing different roles, such as end-users, clients, project managers, designers, programmers and economists. Some of the roles, such as clients and end-users, may be played by people invited from the in-

Table 5: Dimensions of variation related to software maintenance description categories.

Category	Dimensions of variation
2: Change and add	Dimension of things that customers may want to add or change
3: Adapt and update	Dimension of external contexts that may affect the software
4: Continuous work	Dimension of tasks, activities and responsibilities expected at work

dustry, others may be played by faculty members and then of course the students are the designers and programmers. In this case the teacher sets the stage and prepares it to ensure that variations in the dimensions of variation will happen. During the project each team is monitored by the teacher who can introduce striking variations that will affect the project team. For example the client can make major changes to the requirements, the indecisive end-user can be replaced by someone with very strong views, or the budget may be cut down. It is important that students get the opportunity to experience a variation of application domains in different projects and this should help them to learn that domain knowledge is essential. The software design process includes the application domain which is not always the case in programming [11].

7.3 Variations for software maintenance

This section examines the outcome space for software maintenance to identify relevant dimensions of variation that are needed to reach a richer understanding. Naturally, learning software maintenance should never be the greater part of a computer science programme, however, it will be rewarding to have experiences and a deeper understanding of maintenance for those who go to careers in the industry.

The outcome space for software maintenance has four categories, see Table 3. The first category represents an understanding that maintenance of software is all about fixing bugs in the program. This way of seeing does not particularly reflect professional circumstances and therefore it is less interesting in this discussion.

In Category 2, the dynamic nature of software and what clients need comes in focus as things may have to be changed or added to the software. The relevant dimension of variation is constituted by typical things that may be changed or added and how this is connected to the software.

Category 3 introduces an understanding that software maintenance may involve adaptations to changes in external parts of the software system, such as operating systems, hardware or third-party products. Thus, the dimension of relevance consists of different external factors that may affect the software and what this implies for the code.

Finally, in Category 4, software mainenance is seen as a continous work effort that comes naturally in professional contexts. An appropriate dimension of variation consists of different tasks, activities and responibilities that are expected from persons who are employed in the software industry. Table 5 summarizes the identified dimensions of variation.

To facilitate a suitable context for the variations related to Category 2 and Category 3, a larger software system could be prepared along with detailed documentation. As an assignment, students should get acquainted with the software and then they should receive a series of maintenance tasks. These tasks should reflect the dimensions of variation related to Category 2 and Category 3. It should be noticed that the understanding of software maintenance represented by these categories is strongly connected to the third category of understanding of software development, "designing for the future" and synergy effects should be expected.

The fourth category describes software maintenance as something that comes naturally with the job. It reflects a professional perspective and an overall understanding of how expensive maintenance is and how extensive it is seen from the perspective of a program's entire lifetime. The variation in this case may be obtained through visits to companies with large legacy systems where students would get the opportunity to see what people do at work, and by arranging inspiring guest lectures.

7.4 Learning through maintenance

Some informants report that they think it is difficult to read others' code. One way to get used to reading code and learn about software is by maintaining old software.

> At least one assignment should include substantial revisions to a previous programming assignment or other moderately large program(s). [17, p.8]

By engaging in troubleshooting, or to make improvements in a system, one can not fail to learn a lot about things, such as how not to program, how to design the next program so that it works better, how to use the company's proprietary APIs, and how the company's custom and practice permeates the way to program.

Spontaneously, many informants said that maintenance, i.e, "to fix bugs", is boring. This is an interesting result in itself. Would a medic student find it boring to deal with diagnosis and troubleshooting?

> Hence, a maintenance engineer should be a highly skilled, intelligent, and creative diagnostician. This requires that universities properly prepare students to enter the maintenance workforce, and that maintenance organizations actively build and maintain their body of knowledge. [6]

Could it be that those informants have their own bad experiences from getting stuck in finding errors in a pro-

gram in pressured situations? Many students have experienced dozens of lines of compiler errors and lots of strange bugs in their programs before they could even run them, which may be perceived as very frustrating and boring. However, troubleshooting, i.e., to make a diagnosis based on how a program behaves, is potentially an exciting task that requires a lot of knowledge and creativity. Organized instruction on troubleshooting and training by solving tasks where students can search for "exciting" errors in prepared code, could perhaps result in a more positive attitude?

7.5 Unawareness of version control

Source code management[3] is important to organizations that handle source code professionally. The system keeps track of all the company's different parts and versions of source code and makes it possible to share work in teams with large software systems. The programmer checks out pieces of source code, works on it, and checks it back in again. And if there are conflicts with others' modifications, these are easy to identify. Naturally, a source code management system is used both for software development and maintenance activities.

Interestingly enough, none of the informants even mentioned the complex issues concerning source code management and version control systems. It is true that some informants talked about "documentation" in general terms, but no one touched this topic. In this aspect computing education has not succeeded to properly prepare students for their future careers.

Version control is probably one of the issues that professionals would mention if they were asked about their experiences from professional work with software. In large organizations there are many different versions and branches of programs and program libraries, and that is a completely different thing than dealing with a small program in a student project. The fact that informants are not talking about this aspect is probably because they are not very aware of it at all. However, it is absolutely necessary to master source code version control in serious professional contexts.

This "result" suggests that you should consider using version control systems in programming courses – not only in courses on software engineering. Version control is something that can be introduced into teaching as a learning outcome, but also as a means for learning by letting students, from the outset of their studies, have access to a system that can store their source codes and other documents to be shared with other students, making rollbacks or version ramifications. This would not only teach students about versioning, but also make it easier for their other activities, for example, it would be easier to work in a group with major projects. Or why not let a whole class work together on a more comprehensive software?

Begel and Simon suggest that we should work more on projects with many community members in order to train social and communicative abilities and they also propose that students should be allowed to work with existing software:

> Instead of a greenfield project, a more constructive experience might provide students a large pre-existing codebase to which they must fix bugs (injected or real) and write additional features. Incorporating a management component would be valuable, where students must interact with more experienced colleagues (students who have taken the class previously, who can act as mentors) or project managers (teaching assistants) who teach them about the codebase or challenge them to solve bugs several times until the "right" fix is found. During the development process, students could be asked to log bugs in a bug database, develop bug reproduction steps, and/or triage the importance of the bugs given some planned release schedule. [1]

I suggest source code management should be a natural part of computing education.

7.6 Identification

Students who are interested in working professionally with software need to learn the conditions and culture of the business. In order to deepen knowledge and provide a mixed picture, it may be worthwhile to invite companies to participate in classes, such as field trips, guest lecturers or as clients for student projects [4, 5]. The following quote from one of my informants shows the importance of identification:

> Yes, to my great joy, because I thought the programmers sat in a dark cellar and programmed ... they were deathly pale and had dark circles under the eyes and they were not socializing with anybody ... they worked only at night ... But then we actually had a teacher in this course who came from outside, she worked with Java programming ... for companies ... and she told me that they always work in teams, they helped each other, they changed position in the group so it would not be too monotonous, they worked only during the day, and they kept their 40 hours a week ... I had a very bad picture of programmers ... but she told me quite a lot so therefore my view of the programming profession is positive, because I have heard someone sitting in it daily... she took off one day a week and did other things, she did not get caught in programming, but it was still her main interest, but she had found a good balance, so I felt there is hope ... She was quite alone in her Java role but now she had still other people around her ... I realized that you are not alone and that was great comfort to me, I can say. [S13]

8. CONCLUSIONS

This paper reports results from a study with the general purpose to investigate how students in Computer Science experience aspects of their future profession. The research questions raised were: (1) how students

[3]Other terms used are source control, revision control or version control.

experience software development, and (2) how they experience software maintenance. Twenty informants from four different Swedish universities were interviewed. All of them were in the later half of their educations and some of them were graduating within months.

All informants wanted to work with aspects of software development and they aimed for a job in the software industry after graduation. Working as software developers or programmers was the dominant future prospect, whereas a few wanted to work with "softer" issues such as requirements analysis.

The research questions were answered using phenomenographic analysis of the interviews and the resulting outcome spaces are categorizations of qualitatively different ways of seeing the phenomena.

The informants' ways to describe software development are reflected by an outcome space consisting of four qualitatively distinct categories of description: "problem solving", "program design", "design for future", and "understanding the need". Their ways to describe software maintenance also produced an outcome space with four categories: "handling bugs", "change and add", "adapt and update", and "continuous work".

These results are discussed in terms of which dimensions of variation are needed for obtaining rich ways of seeing professional aspects of software development and maintenance, and some examples are proposed for how to utilize this in teaching.

It was found that not all informants had reached advanced ways of describing the phenomena, especially software maintenance, which mostly was described as dealing with bugs. However, this is not surprising as it probably reflects the relative absence of experience of maintenance in education. It was also observed that version control of source code was never mentioned, either in connection with software development or software maintenance.

The final conclusion is that more can be done to better prepare students for the transition into professional life and the present work suggests a number of ways for teachers who wants to address such issues in teaching. It is hoped that the results presented here will contribute to a widened understanding of the variation in how students experience professional aspects of computing and inspire teachers to reflect on teaching and learning.

9. REFERENCES

[1] A. Begel and B. Simon. Novice software developers, all over again. In *ICER '08: Proceeding of the Fourth international Workshop on Computing Education Research*, pages 3–14, New York, NY, USA, 2008. ACM.

[2] J. Boustedt. *Students working with a Large Software System: Experiences and Understandings*. Department of Information Technology, Uppsala University, Uppsala, Sweden, 2007. Licentiate thesis.

[3] J. Bowden. The nature of phenomenographic research. In J. Bowden and E. Walsh, editors, *Phenomenography*, Qualitative research methods series, pages 1–12. RMIT University Press, Melbourne, 1st edition, 2000.

[4] L. Jaccheri. Software quality and software process improvement course based on interaction with the local software industry. *Computer Applications in Engineering Education*, 9(4):265–272, 2001.

[5] L. Jaccheri and S. Morasca. On the importance of dialogue with industry about software engineering education. In J. B. Thompson and H. M. Edwards, editors, *SSEE '06: Proceedings of the 2006 international workshop on Summit on software engineering education*, pages 5–8, New York, NY, USA, 2006. ACM.

[6] M. Kajko-Mattsson, S. Forssander, G. Andersson, and U. Olsson. Developing CM3: Maintainers' education and training at ABB. *Computer Science Education*, 12(1–2):57–89, 2002.

[7] F. Marton. Phenomenography - a research approach to investigating different understandings of reality. *Journal of Thought*, 21(3):28–49, 1986.

[8] F. Marton. The structure of awareness. In J. Bowden and E. Walsh, editors, *Phenomenography*, Qualitative research methods series, pages 70–79. RMIT University Press, Melbourne, 1st edition, 2000.

[9] F. Marton and S. Booth. *Learning and Awareness*. Lawrence Erlbaum Associates, Inc, Mahwah, New Jersey, 1997.

[10] F. Marton and M. F. Pang. On some necessary conditions of learning. *The Journal of Learning Sciences*, 15(2):193–220, 2006.

[11] W. M. McCracken. Research on learning to design software. In S. Fincher and M. Petre, editors, *Computer Science Education Research*. Taylor and Francis Group, London, 2004.

[12] T. Muhr. *User's Manual for ATLAS.ti 5.0*. Scientific Software Development, Berlin, 2 edition, 2004.

[13] D. L. Parnas. Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, 6(1–4):19–37, 1998.

[14] R. Shackelford, J. H. Cross II, G. Davies, J. Impagliazzo, R. Kamali, R. LeBlanc, B. Lunt, A. McGettrick, R. Sloan, and H. Topi. *Computing Curriculum 2005: The Overview Report*. ACM, 2005.

[15] M. Thuné and A. Eckerdal. Variation theory applied to students' conceptions of computer programming. *Europeean Journal of Engineering Education*, 34(4):339–347, 2009.

[16] E. Walsh. Phenomenographic analysis of interview transcripts. In J. Bowden and E. Walsh, editors, *Phenomenography*, Qualitative research methods series, pages 13–23. RMIT University Press, Melbourne, 1st edition, 2000.

[17] L. H. Werth. Integrating software engineering into introductory computer science courses. *Computer Science Education*, 8(1):2–15, 1998.