Ways to Understand Class Diagrams

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

The software industry needs well trained software designers and one important aspect of software design is the ability to model software designs visually and understand what visual models represent. However, previous research indicates that software design is a difficult task to many students. This paper reports empirical findings from a phenomenographic investigation on how students understand class diagrams, UML symbols and relations to object oriented concepts. The informants were 20 Computer Science students from four different universities in Sweden.

The results show qualitively different ways to understand and describe UML class diagrams and the "diamond symbols" representing aggregation and composition. The purpose of class diagrams was understood in a varied way, from describing it as a documentation to a more advanced view related to communication. The descriptions of class diagrams varied from seeing them as a specification of classes to a more advanced view where they were described to show hierarchic structures of classes and relations. The diamond symbols were seen as "relations" and a more advanced way was seeing the white and the black diamonds as different symbols for aggregation and composition.

As a consequence of the results, it is recommended that UML should be adopted in courses. It is briefly indicated how the phenomenographic results in combination with variation theory can be used by teachers to enhance students' possibilities to reach advanced understanding of phenomena related to UML class diagrams. Moreover, it is recommended that teachers should put more effort in assessing skills in proper using of the basic symbols and models, and students should get many opportunities to practise collaborative design, e.g., using whiteboards.

1. INTRODUCTION

Most students with a major in computer science or computer engineering aim to work in the software industry after graduation. Hence, one of the main objectives of university level computing education is to prepare students for professional carriers in companies involved in software development.

Not only does the software industry require people who can solve problems and write code; they must also know how to communicate and be able to discuss problems in teams and be willing to adapt to the company culture. They should be prepared to get into other persons' code for maintenance and be good at documenting what they are doing. The expectations are high.

People who discuss design of software need to represent their thoughts and ideas with some sort of model. The Unified Modelling Language (UML) is a visual model language that fits object oriented analysis and design well [2]. Its rich set of diagram types and symbols is used to model and document many of the aspects relating to software and it is at present the de facto standard in the software industry [19].

The modern software design process involves an exchange of ideas on many abstraction levels, and it is often an iterative process [13]. Not only will the model language and the models support the design process and the implementation of the program. In addition, the collection of design documents is a documentation of the system which will be used and maintained for a long time by several persons.

Since one of the desired outcomes of computing education is that students should be able to reflect on programming problems and design solutions before they write code, they are encouraged to use visual models. One way or the other, UML and class diagrams come as a natural part of teaching object oriented analysis and design. Most modern textbooks on design and programming use UML to visualize concepts and ideas, and students who take a software engineering class will devote more attention to software design processes and UML models. Nevertheless, compared to all time spent on learning programming, much less effort is put into learning how to model software, and previous research shows that students are poor at software design [7].

It is reasonable to assume that, for students, the software design process is very different from coding, especially regarding feedback. Program code must be correctly written in order to execute in most programming languages, whereas it is much harder to verify the syntax and semantics of visual design models. This implies that students can not "self-learn" how to draw "correct" diagrams in the same way they learn programming guided by feedback from the harsh compiler.

Taken together, not at least in the industry, it is crucial that people have a shared understanding of what

software models mean; that the meaning of the symbols used in the design models are unambiguous to the involved persons. Even if it may be impossible to reach this goal for all students, educators should benefit from knowing more about the students' design "toolbox" and how they understand some of the elements of software design.

Indications from research [7], my own experiences of student designs, and the requirements from the industry have made me interested in how students handle and understand class diagrams and their purpose. My experiences of how differently students use or do not use the symbols for whole-part relationships has made me especially interested in how "the black and white diamonds" are understood by learners.

The aim for this study is to empirically investigate how students experience UML class diagrams and some of the elements and concepts involved.

The main research questions in the study are, from the students' perspective:

- What is the purpose of a class diagram?
- What is a class diagram?
- What do the black and white diamond symbols in a UML class diagram mean?

2. THE UML AND CLASS DIAGRAMS

This section is mainly intended for readers who need a brief introduction to UML and object oriented concepts.

Since program code is in many cases far too detailed to use as a basis for thoughts and discussions there has emerged many graphical notation languages that are designed to express ideas on several abstraction levels. UML is frequently used in textbooks and in programming and software engineering courses.

There are various competing development methods that use different graphical notations to represent software components, relations, program flow, et cetera. Three dominating actors, represented by Grady Booch, James Rumbaugh and Ivar Jacobson, the "three amigos", joined and agreed on a unified notation. UML 1.0 was released in 1997 and it is now a standard notation language.

In object-oriented (OO) software, most classes and objects have relations to other classes and objects. OO-theory defines a number of relations between classes including: aggregation, composition, inheritance, association. In the UML class diagram, these relations are symbolized by lines between the boxes that represent classes. These lines appear in different variants: they can be directed (arrow heads), undirected (only a line), dashed (interface), or have special arrowhead symbols (diamond, solid diamond, and closed arrowhead). The relations can also be annotated with relation and role descriptions.

In the static model diagram (class diagram) there is a fairly limited set of symbols that can be used to express how the designer wants to model the software. Relations between classes can be specified and each symbol is associated to a certain concept from OO-theory. In UML an "aggregation" is symbolized by a line with a outlined (white) diamond in the end that is directed towards the "container" and "composition" is symbolized by a solid (black) diamond directed towards the "whole".

The distinction between aggregation and composition (composite aggregation) is not unambigous in textbooks and in research literature, see Section 6, and this is one of the reasons for our third research question in Section 1. Nevertheless, one way of describing both is that aggregation and composition are when objects are parts of something that constitutes a whole that is different from the mere sum of the parts [6]. As an examle, consider a MP3 player with its internal components; the display, the buttons, the case, the circuit board with its integrated components and in order to be a functioning device, it also need batteries – and some tunes. And if the owner, who sees the player as a whole, pushes the play–button, all of the parts will secretly interact with each other and perform the desired action. But there must be a limit drawn at some point. The owner of the player has a close relation to the player but is not a part of it.

Some parts are loosely connected; aggregation – the player can exist without the batteries and tunes which could easily be replaced, whereas some parts are tightly connected; composition – the player cannot exist without its display, circuits and case.

The aggregation is more dynamic than the composition. For example the player "has" batteries and tunes, but the batteries can be removed and replaced and the songs may vary in numbers and titles. And batteries and songs can be moved to other players. When the player goes to the junkyard, the batteries and songs may continue living their lives. The battery and song would then be part of an aggregation which in a programming language should be implemented using some dynamic mechanism, i.e., pointers or reference variables.

However, the player is constructed by (consists of) a case, circuit board, display, et cetera. These individual parts are bound to this very player; as the whole consists of its parts. When the player goes into the pocket, so do the parts. This describes a tighter relation between the whole and the parts, i.e., composition.

3. 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 by 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.

In one part of the interview, the informant was presented with a number of design diagrams in UML. The interviewer encouraged the informant to discuss these diagrams and talk about what kind of information he or she could get from them. Furthermore, the informants were asked to speculate about how they would implement the design in a programming language. The four UML static class diagrams represented the same model in four different levels of explicitness; the first diagram showed 18 class boxes named 'A' through 'R' without method names or attributes. Unannotated relation lines were drawn between the classes, such as inheritance, association, aggregation and composition; the second diagram introduced more meaningful class names; the third diagram added multiplicities to the relations; and the fourth diagram introduced annotations to the relation lines, such as roles and relation descriptions. The diagrams are further described in Appendix A.

A phenomenographic approach was used to analyse the interviews, which means that the transcribed interviews were examined to find expressions of "meaning" of various phenomena related to class diagrams. 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. Next section introduces Phenomenography and describes how it was applied in this study.

4. 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. [14]

Experiences from earlier studies had shown that different 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 [5] 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. [25]

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 set of "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 [15]. 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. [16, p.126]

4.1 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 phenomenon, e.g., class diagrams, 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 [20]. 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 phenomenon 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 by 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.

5. RESULTS

The data analysis of how the informants describe UML class diagrams was done on three different levels. The first level is related to the first research question in Section 1 and is focused on how the informants describe the purpose and use of class diagrams in software development contexts – a macro perspective.

The second level of analysis is focused on how the informants describe the class diagram and its internal structure as a phenomenon per se.

The third analysis level takes a micro perspective and is focused on how the informants understand the diamond symbol(s) in UML diagrams and their connection to the object oriented concepts aggregation and composition.

5.1 Ways to see the purpose of class diagrams

This section presents the results of the analysis of how the informants described the purpose of class diagrams; how they use them or how they think they are used. During the analysis it was discovered that the informants, on a collective level, focused mainly on three different aspects: (1) How the class diagram helps to understand an existing program or how you can document an existing program, (2) how the class diagram supports the design process and how you can document a design, and (3) how a project team can use class diagrams interactively as they discuss, develop and agree on their designs. The resulting phenomenographic outcome space consists of three qualitatively distinct categories of description. A summary of the outcome space for the phenomenon "use of class diagrams" is shown in Table 1. The following sections describe, for each description category, how the informants experience or understand the purpose and use of class diagrams, by highlighting and discussing selected quotes from informants.

5.1.1 Category 1: Code

In this description category, the use of class diagrams is described as a way to document, describe and overview existing program code. In this way, the class diagrams makes it easier for a programmer to understand the code of a software. In fact, the diagrams describe code and can even be seen as code.

One way to see the purpose of class diagrams, among the informants, is that class diagrams help people to understand the code structure of a software or a system. S01 describes that they give a graphical overview:

> ... so that would make it much easier, I think, and I am not particularly experienced, it would make it easier for me that never have got into a, well, a new system; to have a graphical overview of how components are connected and such. I like UML. [S01]

Table 1: A summary of the outcome space for the descriptions of how class diagrams are used.

Category	How the use and purpose of class diagrams are described
1: Code	Class diagrams are used as a documentation of existing program code which is good for someone who wants to learn about the software and perhaps is going to make changes to the program.
2: Design	Class diagrams are used as a way to develop software designs. It is a tool that can be used to model a solution to a problem and therefore it also documents existing or non-existing program code.
3: Dialogue	Class diagrams are used as a way to design software and are used as a means for a dialogue with team members in a dynamic design process that will end up in program code. They are perfect to use as a tool to develop, discuss and test models on a whiteboard together with the team.

S07 describes that the diagrams may not be so useful unless you also have the code. The diagram is a documentation of the code and if you have the code you can make a diagram that gives an overview of the code:

> ... the UML diagrams are maybe more useful if you have the code first, before you are going to develop something, develop a code further so to say. So it just felt that if I get a UML diagram from someone, it would not be as worthwhile as if I get code from someone and then apply a UML diagram. And the UML diagram is used, well, as a documentation for oneself really, well, a brainstorm, a list, you know, a plan for what it looks like. It feels that it is more rewarding to get the code first and from that make the UML diagram – compared with what you get out from only the UML itself, if you get what I mean. [S07]

S04 mentioned UML in connection to software development. When he was asked if this was part of designing the software, he answered:

> Yes, I guess it is important to have it in the documentation. [S04]

This answer indicates a view that UML diagrams are more part of the documentation of a system than part of the development process.

S09 describes that the diagrams are closely related to code, that they in fact can be regarded as code:

> ... due to the fact that the experience I have from UML is almost exclusively in relation to writing code, so by old habit, I see it as code. [S09]

In summary, this way to understand the use of class diagrams is mainly that they document or represent code in a visual form (non-textual).

5.1.2 *Category 2: Design*

This category describes that class diagrams are used to develop and model software designs before starting to code. Class diagrams can be used to sketch a solution in a top-down manner and can be refined gradually. The design model can then be implemented into code which will also make it a documentation of the code, why this understanding also includes the understanding in the previous category.

In the informants' descriptions there is a specific understanding that UML diagrams are used when people design a software system and that there are more than just class diagrams in UML. For example, S02 says it supports a top–down approach to the design process:

> ... I think that you do design with UML rather much actually. That you describe some different steps for how calculations should be handled with, well, on some different ways that you can use in UML. Not necessarily only class diagrams because there are some different models to use, so I think that it is used pretty much when you want to sketch what you want it to look like. And then it can be done very roughly, that you only have a module somewhere but for the time being you don't care how things look like inside the module. Instead you start on a very high level and work yourself downwards all the time. I believe that is a pretty good way to make designs. I think this i how you do it as well. [S02]

For S14 it is good if you visualize a design first to get an understanding of the parts – instead of coding in the dark:

> ... yes, but it may be good and if you are going to create a system, that you draw before how you think and how it should fit together to get an understanding, not just sit and code in the dark, for example, that, well, and then it shows how they fit together hierarchically, which these classes can access and, yes, such things. We have touched upon that a bit more lately, in order to get, then you get more grip on what is needed and what is not needed. Otherwise, you may perhaps think that you need ever so many things just because you think it should be so complex. [S14]

S02 starts with drawing the design in UML before the coding and there are software tools that can be used for making the design diagrams:

> As I have done now, for example, in the side projects, I usually work in, in an IDE called

> NetBeans. And there you can draw a UML diagram. So already there in the design phase you draw a UML and, and think about what you want it to look like, that you start kind of. I usually tend to start at the stage that I begin with the UML, and then I generate code from the UML, and start from there, when we are going to start to code. [S02]

Designing with UML, S20 says, is appropriate when you deal with large designs in professional contexts; however, in a small project it might be too time consuming. The class diagram is a structure that describes the design of a software and programmers can use it to understand what they are supposed to do:

> Oh, yeah, but I still think that in a professional context, I think that UML, then it might be useful. Because when you teach and when you build something with friends and so on, then it may be that you spend too much time on the UML if everyone understands what kind of thing they are supposed to do. But in a professional context, it's very important that you really have one, well, great design and very defined design, and then, it is also easier as a programmer to develop things if you get this kind of chart so you can see, then you know a bit more what to do. [S20]

S20 explains that a visual representation helps people to really understand a design and what they should do:

> It requires graphic representation to really understand, because I think everyone, all students are more or less visual, so it helps a lot. And then in a professional situation, it is also a great way to instruct programmers how they should begin their work, instead of having a meeting with the designer for some hours in which you write your own notes, you get a diagram showing the key relationships, and when you have done so much you can do from it, then you can then proceed with more details. [S20]

In a similar way, S09 describes how the diagram helps a designer to form a structure that helps others to understand which parts they should work with:

> Yes, strictly speaking it would be much like, well, that together you prepare some sort of, no, not together necessarily, that there is someone who has an eye on the project as a whole that makes a UML that, or well, some sort of structure from which the others can undertake pieces, so to speak. [S09]

This description category can be summarized as follows: Class diagrams are used to design and model software solutions before the software is implemented.

5.1.3 *Category 3: Dialogue*

In the third category, informants describe how they use class diagrams interactively together with the other team members in the project. The purpose can be to get an understanding or agreement of what constitutes the software as a whole and which parts a developer are responsible of and their complexity. S07 describes:

> The UML diagram is an aid during a development process so that you can get the different, that you have a person A who makes class B, and you have, yes, and so on, different people doing different things, and that they can cooperate in a way and understand the entire program and understand the major issues affecting them. Then I think UML diagrams are great, while the code is good if you have to go in specifically in a certain part and see: how can we improve this, or what is here? [S07]

Some informants describe how class diagrams can be used in vivid brainstorming processes. S02 talked about his experiences of working out a coarse–grained sketch of a design on a whiteboard together with his team:

> When we have worked in groups, we usually, at least at school anyway, we usually sit in a room when we design our programs. We take an empty room and then we write down what we want to have on the blackboard. We brain-storm which classes are needed and what kind of thing we will try to do, and then we sketch a very rough UML for what it should look like. When we have done it, we touch it up a bit, remove things that may not be needed, divide something that, that is, we divide some class into two parts if needed, and so on. So that's what we usually try to do, that we try to outline a rough UML, basically, and then, based on that, we work to produce what is wanted. [S02]

Informant S13 describes that he and his team-mate were fortunate enough to have access to a whiteboard of their own during the entire project period, and how this was very useful. But they had not thought about making a backup:

> Since it is object-oriented programming, there were of course many objects. There were about 14 different ones or something, so, we had to have a whiteboard in there and so, but every time we came there you could look at that and you always discovered something. It was really, really good actually, because we never erased anything. It was one day the cleaning maid had erased it ... Yes, [laughter], it was close to murder in the house but it was just to start drawing all over again, and it was quite useful because then we got maybe some other, we could not exactly remember what it looked like, but now we have begun to photograph the whiteboard. [S13]

Several informants describe that it is a good idea to develop software designs in collaboration with the

project team; however, collaborative work can also be a source of problems that are hard to handle. S01 points out a problem that he experienced when his group designed a solution to an project assignment. After long discussions, the group agreed on a solution model (using class diagrams); however, when they started to implement the code, the subgroups started to do their own solutions that deviated from the design documents. And they did not go back and discuss this with the entire group and the design documents were never changed. Everybody knew this, but nobody wanted to start a new design discussion.

> It depends a bit on the project you are working on, and how well you have done your design, because what may look very fine on a whiteboard and very logical and good, may in the end, even if you only have missed a detail in the program flow, it may be very complicated later, because then you are really locked by the image you had from the start, so. There is also a big disadvantage to have a clear picture before you start, because, especially if you work in a group it becomes easy that you, you may not modify the common picture, because I think then you can destroy for someone else, who might have been, who, who think strictly in accordance with the image which we drew up together, and if you keep rigidly fixed in the original image then, then the program is not getting better from that, but you may have to sit with the group again and, and, or as soon as you discover gaps in the common image, that you go back and redraw it. [S01]

This category described the use of class diagrams as being a tool for communicating and discussing about software designs during the software development.

5.2 Ways to see what class diagrams are

The second analysis of the transcriptions was looking for qualitatively different ways to understand and describe what class diagrams *are*. The result is an outcome space with three qualitatively different categories of description: (1) The class diagram is a visualization that shows classes as boxes with instance variables and methods and the diagram shows how the classes are connected to each other, (2) the class diagram is a visualization that shows classes and various types of relations between classes, and (3) the class diagram is a visualization that shows hierarchic structures of classes.

5.2.1 Category 1: Classes and connections

The descriptions in this category characterize class diagrams as a visualization that shows classes and connections between classes. It gives information about the classes' properties and what they "do". The class diagrams used in the interviews represented classes without specifications of attributes and methods and the internals of the classes was not addressed explicitly in the discussions; however, the understanding that class diagrams can show this information was mentioned indirectly, e.g., S02 who cannot see what the classes "do":

> ... I can not really see what the different classes do, of course, but I can just see how the relationship to them looks like. [S02]

or directly, e.g., S17, who mentions variables and functions:

> ... anyway, I do know this much about Java diagrams that you should have both variables and functions in the class diagram ... Yes, then the important thing is what types the variables have ... and then, if you are going to implement this, you must of course also have the functions, so you know what it is you should do... [S17]

Informant S14 describes how he uses class diagrams to define which classes to use and how they are "connected" but uses no specific symbols to discern the character of their connections.

> I have not used these particular UML diagrams, you know, with the different [symbols] ... but we have, you have drawn kind of connections from which classes you should use and roughly how they should fit together. [S14]

S14 also describes that the diagram shows classes and their connections. In this way, S14 says, it is possible to see how the flow goes between classes:

> ... we think we need to sketch up how, kind of, yes, like class models or whatever, giving relationships, and showing how it, yes, how all, all the flows must go, so to speak, so that it becomes clear to the one who looks at it... [S14]

When presented with the class diagram used in the interview, S08 describes how it shows relations between classes; how they communicate and since the classes do not specify methods or properties, it is not possible to implement the design.

> Oops, the first thing I think of is just that this only shows relationships between, between like classes, and how they communicate with each other in some way, and, you would surely not be able to implement anything from this, but, if you would have another sketch that shows you how to implement, then maybe we could apply this, discern where to start, and which parts I can put aside to develop at the end, and what part I need to develop in order to test other parts. [S08]

S17 expresses a similar view:

> I can see that there are different types of information that is primarily focused on the communication between these boxes, for they [the classes] all look exactly alike, they just have different names... [S17]

S17 continues to describe different forms of arrows, but cannot tell what they mean:

Table 2: A summary of the outcome space of descriptions of UML class diagrams.

Category	How class diagrams are described
1: Classes and connections	It is a diagram that shows classes with their internal methods and attributes, and it also shows an overview of the connections or flow between the classes.
2: Related classes	It is a diagram that describes classes and their connections. The connections, or relations, can have a number of fundamentally different features.
3: Hierarchies	It is a diagram that shows classes and how they are related in abstraction hierarchies involving classes, instances, interfaces and sub-structures, such as inheritance structures and aggregations.

> ... and then there are of course different kinds of arrows, filled and non filled, dotted or, what is it called, not dotted, straight lines, and you can also see, if you think that this is classes or modules or anything, you can see which ones are most important or most central. G here, for example, is in fact, seems in fact to be in the middle of the chart, and also has many arrows towards it, and above all here, well, it is difficult to say when you do not know what the arrows mean or what it is all about, so I can probably not, well. [S17]

Informant S12 describes that diagrams can be understood as a combination of classes and flow:

> ... if you should do this smart, you can of course draw, I usually make a mixture of class diagrams and flow charts, it shows, well, from this class and down to that class and it does this, and then it goes to that class which does that. [S12]

S16 describes how the connections between classes can express a dependency between the classes:

> I think the thicker arrows seems like, I do not know how to explain them, but the arrow from A to B, for example, that is a thick arrow, and I think it feels like A is very dependent on B, for example, that it gets something big, I mean something important from B. While B and C, here it is more like a relationship, you know, because, well let's say relationship because I don't recall. [S16]

In summary, a class diagram is described as a overview visualizaion, showing a number of classes, some of which have connections to each other

5.2.2 *Category 2: Related classes*

This category of description includes the previous category and opens up a new dimension to how class diagrams can be understood; the connections between the classes in the diagram, or relations, can be different in their semantic nature. S20 tells how the class diagram is a way to show the relation types and that there are many different types that can be expressed in UML, but he also says that he is not very familiar with the notation for these relations:

> As far as I know, the UML is, it's a way to show relationships between different entities, and there are lots of them, I imagine, in UML, a great many different types of relationships, so many, and it is something that I, frankly, never learned, or so much of anyway. [S20]

S20 uses an information perspective and explains how relations can mean different things; in which direction the information flows, how information is shared and how parts are created by other parts:

> If you have two elements that have a relationship, how can they, how can the information go between them, what information is common in those elements, and, or you could also say how the parts have been created by others really. It gets pretty abstract when talking about what a relationship is. It seems more like, well, the only thing described by the relationship is that it is some form of information flow that goes from one element to another. Or if they have both in some way, then you have, in fact, inheritance from a class is that you copy information, well, so, I imagine some kind of information flow when I think of relationship ... my experience is that in classes, when you program some kind of application, mostly you, what you're watching mostly is, is which way the arrow points, in which direction the relationship points, and perhaps what, what symbol you have on the element, if it is a circle or a square. [S20]

Many informants talk about inheritance as a relation between classes in class diagrams. S14 describes that classes involved in an inheritance relation have similar properties. Moreover, he describes that information exchange between classes is a relation:

> It may be that they, if they inherit, then they have equal, or not equal but similar properties, or values in themselves, but then if, if they have a different type of relationship, it may surely be to collect information from each other, so to speak, if it is to be presented in any way, and then of course it can go in different directions as well, and it depends on how they are connected with each other. [S14]

S17 explains what inheritance between classes means and adds the understanding that it can provide for a polymorphic behaviour:

> Maybe I thought of different types of inheritance and these, what's the name, interfaces, well, abstract classes, and such ... Inheritance, a class that inherits another class will get its parameters and functions, which in Java means that you may have, poly, what's the name of it? I have forgotten what it's called, that functions have the same name, but behave differently. [S17]

The previous quote touched upon interfaces and in another explanation of different relation types, S20 explicitly describes that interfaces are involved in another relation type; the implementation relation between a class and an interface:

> If this is a picture of object orientation, then a relationship may be that what the arrow points to is a class inheriting an existing super class. It could also be an interface, and here is another interface, or that it is a class that implements an interface. Is there is more? That is, well, what comes to mind within object orientation at the moment. So there are, well, there are many relationships like that. [S20]

Talking about relations between classes, S19 describes how class diagrams show inheritance relations involving classes and interfaces; however, looking at the diagram, he discovers a contradiction; a containment relation between two familiar graphical components, which makes him slightly confused:

> Inheritance is a relationship, I suppose, and to implement an interface is a relationship for sure, so that is imaginable, indeed. But when I start talking about this now, I think that there is someone who has put a JLabel in a JPanel and then it's relationships between objects and not relations between classes, so then there are both, in that case there are both objects and classes in this diagram, and then I get a bit confused. [S19]

Then, S19 shifts his perspective and opens up for a different interpretation of what can be represented by a class diagram:

> Or maybe this is not only inheritance, rather it is some sort of diagram of something that actually is built, so that there is a UserPage that has a JPanel in which someone has put a button, a label, and a text field. But I do not know if there is such UML diagram type, though I suppose there ought to be. [S19]

S19 is asked to explain his view and he describes that he thinks of class diagrams as being on a conceptual level – not on a design level:

> Yes, that is how I imagine the word class diagram. I think it's called something else when you describe some kind of, what shall we call it, implementation, or well, something, a system you have made. So I imagine that the class diagram only means how the classes fit together, I mean conceptually with inheritance and implementation of interfaces, not how you actually have a certain instance of your – a system that is kind of made, but perhaps that is the case if taking other class diagrams. [S19]

S07 says it is possible to understand how the classes are related and how they should be programmed if you have proper names on the classes and if you are familiar with the system. Nevertheless, S07 sees that some classes are contained by another class:

> Well, if you get real names there, and are somewhat involved in what the system is supposed to do, you can see how different classes should fit together, how they should be programmed. Well, like this class G for instance, it should contain J, H, K ... So this is a UML diagram then, as I see it, and it is a class diagram, you can say, and it helps programmers to see which relationships there should be, you know. [S07]

S01 explains that it is easier to understand why classes are related and in which way they are related if the classes have names that explain their nature. The class diagram gives an overview; however, it requires more information in order to make it more understandable:

> First ... it gives a general view. Now, much more information than what is written here is needed, such as explanatory class names, because when you have good names for things, it is usually very much easier to understand a relationship. Take for example what I said earlier, if H should be a student and K a course, then it would be much easier to see the structure and why they must fit together, or how they fit together. [S01]

Summary: A class diagram is described as a diagram showing a number of classes, and some of the classes are conncted, related, to each other in different ways. The informants mentioned a number of relation types, e.g., inheritance, implements, information flow, and containment.

5.2.3 *Category 3: Hierarchies*

In this category, the descriptions are focused on that the class diagram shows, or should show, classes structured in vertical hierarchies where the important thing to illustrate is the structure itself, in addition to the individual classes. It includes the previous categories because the described hierarchies includes connected classes and their relation types. Some of the informants talked about hierarchic inheritance structures and aggregation structures.

Sometimes these structures are referred to as trees, and in the first presented diagram, S02 sees an inheritance tree that includes most of the classes:

> ... it seems that we have some base class here that B inherits from. B in its turn uses some kind of interface here from which we have two implementing classes, B and E. Further on, we have two classes L and F that inherits from E, and then it keeps on in that way further down the tree. We have some class F that has a sub–class G, and G in its turn has three sub–classes, J, H, K, well, and in their turn they use some other classes and then we can traverse deeper down in the tree until we get to N and O here farthest away. [S02]

S02 finds the diagram a bit confusing because it does not consistently follow a hierarchic vertical structure. It seems that the classes are expected to be organized so that their vertical positions correspond to different abstraction levels:

> It is not easy to understand this UML in regard of how the arrows go. We have some arrows that go up, some go down, some go sideways, so it is not really, it is not quite so clear in which level all classes are in ... it feels like, if an arrow goes down, then it is like a lower level. Which means that you have abstracted something, and therefore it gets a bit weird to draw it upwards, because you don't abstract upwards, but you do abstract downwards and then it becomes a bit weird, that you add arrows going up. And sideways is ok, sideways is ok because then it is at the same level so to speak, but when you draw upwards it becomes a bit strange. Even if it is possible to do so, it gets a little weird. It is probably done so here, otherwise the arrows would go around the other classes and then it would look very bad, so I guess this has been done only for show. [S02]

When S02 is asked what would make the diagram easier to understand, the answer is that some classes could be split, and that the diagram could be re–structured into something like a tree:

> ... you could divide certain classes in several other classes, for example, that you divide UserPage in two different, possibly that you restructure the diagram slightly into a more tree like diagram... [S02]

S17 talks about inheritance symbols in the diagram and reacts on inheritance between classes on the same vertical level in the diagram:

> Then it has to be inheritance, I guess, and in Java there are also interfaces, different types of inheritance and relations. But I have a bit of a hard time to understand those [classes] on the same level here. [S17]

And when asked if it is more common to see the classes in class diagrams structured as hierarchies, S17 answers:

> As trees, yes. [S17]

S01 sees two kinds of structures in the diagram: inheritance structures and aggregations. He is also concerned about the complexity of structure of the connections and suggests a vertical restructuring:

> Let's see now, this looks like a classical UML class diagram with inheritance structures on one hand and aggregations on the other – that classes are included in other classes ... and quite spontaneously it feels like the connections between all of these classes are way to complicated. Surely, one could restructure it or rebuild the classes to get a clearer structure. Above all when you, arrows shouldn't, the diagram is not well–structured. For example you should draw inheritance hierarchies vertically instead of horizontally and ... between Q and R the inheritance hierarchy is rather going in the wrong direction in a graphical sense, and this is something that I recognize that I struggle with when you are sitting there with your diagram. That you want both that the arrows should not overlap each other and at the same time that you want the hierarchy to be kept intact... And that many classes have very many connections with other classes here indicates that you could rebuild the system a bit, to be more, where the classes would have more well defined responsibilities. [S01]

S01 suggests an even more hierarchical restructuring; that you can separate the diagram into components in sub–diagrams and use these components in a top–level diagram:

> This is what I have done myself in many cases, to divide complicated systems into several sub–systems, and maybe not have everything in the same [diagram] ... and then you would have only four components ... [S01]

To summarize this category, class diagrams are described as visualizations that show, or should show, hierachic structures of classes.

5.3 Descriptions of diamonds and aggregation

The third analysis investigated how the informants experience the black and white diamond symbols in class diagrams. The result shows that the informants described the phenomena in four qualitatively different ways: (1) That the two classes connected with this symbol are related,(2) that one of the related classes "has" the other class, (3) that one of the classes contains the other or that it consists of the other, and (4) that the black and white diamond has different meanings where the white diamons represents an aggregation and the black diamond represents a composition.

The categories are described in the following sections and a summary of the outcome space for the phenomenon "class diagram" is shown in Table 3.

An interesting observation is that without meaningful class names, most informants were uncertain and vague in their descriptions of what the diamond symbols meant. When the next version of the class diagram was presented to them (the one that introduced

Table 3: A summary of the outcome space for descriptions of the diamond symbols in UML (aggregations) – qualitatively distinct categories of description.

Category	How the meaning of the diamond symbol(s) is described
1: Relation	The diamond symbols mean that two classes are related to each other, that they are connected someway.
2: Has	The diamond symbols mean that a class is related to another class and that the class "has" another class, which includes ownership and control.
3: Consist/contain	The diamond symbols mean that a class contains another class or that it consists of other classes. A class "has" another class "inside" itself.
4: Aggregation/composition	The diamond symbols distinguishes between aggregation and composite aggregation. The composite aggregation (black diamond) has restrictive requirements for existence and access to its parts, whereas aggregation (white diamond) is more dynamic.

real class names instead of the letters 'A' – 'R'), some of the students who previously gave superficial descriptions started to draw more advanced conclusions about the relation symbols from the surrounding context.

5.3.1 *Category 1: Relation*

The black and white diamond relation symbols are – relations. The first category describes the black and white diamond relations in a way that reveals a vague understanding of the phenomenon. The answers are often tautological (the relations are relations). It seems as if the informants know that it means something special, but that they do not remember exactly what it was. Many of the informants were uncertain about the diamond symbols. This is a typical way to describe this uncertainty:

> Well, I see in which ways the information goes, with the arrows. Then I don't, I'm not sure, I don't really remember what the filled and the not filled mean, but it is, well, they have relations between them ... [S14]

5.3.2 *Category 2: Has*

In this category, the diamond symbol is described as a relation between classes, meaning that an instance "has" one or many other instances. One common way to explain what "has" means is to express it in terms of implementation details; to say that the instance that "has" the other instance, holds an instance variable that refers to the other instance. When S02 looks at one of the white diamonds in the diagram, he concludes:

> It seems like B has some kind of instance of C, but then you cannot determine if it is a list or if it is only a single attribute of C. In any case B has some kind of instance variable that refers to an instance of C. [S02]

S02 seems pretty confident in his interpretation, whereas S19 is more uncertain. At first he says that he don't have a clue, but then he gives a description that is similar to what S02 described; the meaning of the white diamond is that a class "has" another class; however, what the black diamond means is a mystery.

> I don't have a clue about these arrows with a diamond in one end and an arrowhead in the other. Maybe it means that B has a C somehow, which means that one of B's – what is it called? – fields or instance variables, is of class C. I think that is what it could mean. Then there are arrows here with a line and a black diamond in the other end, and I don't have any idea what that means. [S19]

To S18, the expression that two classes are related means that they are loosely connected and there are two forms of loose connections; either that one instance "uses" another instance – or that one instance "has" another instance.

> Well, you know, relations are loose, for me anyway, but they are very loose in the world of classes, and then I guess, well, they have some kind of connection where one uses another, or the other uses the first – or it has someone, an instance of the first, or several, maybe. [S18]

S18 explains that the difference between "uses" and "has" means that there is a difference in equality of the instances in a hierarchic sense. In comparison, "uses" is more equal than "has". In the has–relation, there is an owner and something owned and the owner decides what the owned must do. S18 explains:

> Like an automobile that has a tire, it can even have four tires if it is a good car [laugh], then the tire may not be aware that it is an automobile that has got it, but it is still there and does its task; it spins and does what it is supposed to do, no matter where it is. Perhaps it was a bike, if we suppose it was the same tire; it still keeps on doing the same thing regardless of who uses it, kind of. So, it is some kind of relation that the automobile has a tire, or the bike has a tire and the tire does what tires do; it always do it in the same way... Has – then it is more of an hierarchy in some sense, the one rules over the other, but "uses" is more on the same level, the first

> class asks the other class about something in the world, or the system you should say, and then it may choose: "No, I don't like you, I am not going to answer that question" and then the first has to accept that, but if it is that the first "has" the other, it must do it, in some way. It gets instructions to do this and that and then it blindly follows the instructions. But it can also be when you "use it", that it ... if you have a process that uses another process for example, then it can, I don't know, perhaps it is busy, it is perhaps only a recommendation: "Could you do this, please – it would be feasible?" [laugh], and such stuff: "No, I don't have time, I'm busy", for example. [S18]

In summary, the diamond symbol is described as representing a "has"-relation meaning that one class has some kind of ownership or control over an other class. Some informants describe that a class has an instance variable that refers to another class.

5.3.3 Category 3: consist/contain

This category describes the meaning of the diamond symbols in terms of classes containing other classes (instances) or consisting of other classes. S05 uses both terms when he talks about a white diamond relation:

> Ok. B consists of a class C, or many C – you don't know about that, and a class D inherits C, so in principle one could say that there is a container B that contains many D. [S05]

S06 uses several ways to describe what the diamond symbol means. The first is "contains", the second is "owns", the third is "lie in". The fourth descriptive term is "be aggregate in":

> And in some way, B contains a C, B owns a C. I have no idea what the arrow that points to the C does, but C lies somewhere in B. C is an aggregate in B, so B owns a C somehow. [S06]

S08 could not describe the diamond symbols when he looked at the first class diagram – the one with only symbolic class names; however, this changed when he was presented with the next version. In this diagram he could see some classes that were familiar to him; for example, the class JPanel that implements a drawing surface in which a programmer can put other graphical components that should be displayed. Now he could use his experiences and knowledge about Java Swing components to figure out what the diamond symbol should mean:

> ... well, a JPanel can contain a JLabel and this JPanel seems to contain a JTextField and it seems to contain a JButton too, so it feels, in that way this open diamond should mean that "this" contains "that" in some way. [S08]

As we have seen earlier, S13 is not certain about the meaning of the white and black diamonds; however, he says that the class H contains the class K and that the class J belongs to the class G. The interpretation is then that the white diamond means containment and the black diamond means belonging to a group:

> ... or H contains K, they have some form of relation, connection – now I introduced another word, connection – but, well relation, while a relationship might be that you belong to a certain, they, belong is maybe a better word; that J belongs to G – and then it is belonging to a group, maybe ... [S13]

The reason for why this category is more advanced than the previous category ("has") is that the descriptions relate to concepts that adds to a deeper understanding, such as consist of (whole/part), containers, aggregation, and belonging to a group. In addition, we can see that the "has"–property also fits for all of the listed concepts, and therefore this category includes the previous.

5.3.4 Category 4: Aggregation and composition

In this description category the black and white diamonds are described as symbols for two similar, but yet different forms of whole/part relations where the black diamond is a "harder" relation and the white diamond is a "softer" relation.

> The difference between filled and not filled – what are they – rhombs? And the distinction between them is a bit – I have read about it and I must always come back to it once I – when I sit there – to define what the difference is, because as I see it, it is fine as a hair. Anyway, in both cases it is about having references to other objects, but in some cases the connection is a bit harder, and that is represented by these filled [black diamonds], in the sense that this cannot exist without the other. Perhaps you could say that if G is a car and J is an engine, then G cannot work without J; whereas, if H is a student and K is a course, the student can function without the course, but it is good if it has got it, for example. [S01]

S10 was asked if he had reflected upon the difference between the black and white diamonds:

> Yes I have, I was a bit confused the last time I thought about it too, but I know that the filled, then, lets see, then these, for example J, H, K must exist to make it possible for G to exist. And this non filled, the slightly weaker, that it may contain ... in those aggregations that have non filled diamonds you can have vectors or lists, and in those with filled diamonds you must have, I haven't done this for a very long time and therefore I don't remember how to do it, if you do something clever to prevent it from existing if something in the collection it should have is missing. [S10]

Consequently, one interesting thing that may be different for the two forms, is what happens when the top

level object in an aggregate is created. S07 explains that when you create a composite (top level) object, its parts are created at the same time, which is not the case for an aggregate.

> This thing with composition and aggregation is a bit hard to define. The way I see composition, is that it is kind of as soon as you make this object, then these are also created, in the other classes. Aggregation, then it is that when I use this method, I create one of these only to pass it along or to use that class for example. [S07]

When asked to describe composition, he answers:

> That a class is dependent, that it must have, well like a car for example, a car needs a steering wheel, tires, and so forth to be precisely a car, and therefore the car is dependent of being able to use the steering wheel and use the car, kind of, the parts. [S07]

And later when he is reading code and are trying to draw a class diagram, he returns to this discussion:

> Lets see, then it is to be hoped that we have a human, a brain then and it is initialized immediately; as soon as the class is loaded you have a new brain, and this is pretty cool you may think, that is called "brain" and as I interpret it, the human consists of a brain, and then we have a composition, a filled diamond below to human brain. And since it is created immediately and because there are no "add–more–brain–method", I suppose that ... and besides it initializes immediately and therefore has a brain regardless if it wants one or not... [S07]

Implicitly we can understand from his "thinking aloud" that if the brain was not initialized in the constructor and if there was an "add–more–brain–method", the human–brain–relation might be an aggregation (white diamond) instead of a composite aggregation.

When asked if he sees a difference in the interpretations of the white an black diamonds, S13 replies in terms of mandatory and voluntary relations:

> Yes I do, because, well, if we start with UserPage, then it is very much to be put in there and then you understand that the things that the arrow or line goes to, simply must get in there as I understand it. And then the white are very voluntary, I would say... [S13]

Some informants experience that the black diamond implies that the parts included in the aggregation are "hidden"; this means that the parts in the composite aggregate are not accessible as separate parts from someone outside the aggregation. S12 said he was a bit unsure about the black diamonds, but when he was asked to talk about them again he said:

> Well, it could be an aggregation that is hidden or something. [S12]

And when S18 talks about how the parts in a composition are created, he also mentions that the parts may consist of parts in their turn, and that the top–level class does not have to know about them (that they could be hidden and abstracted):

> The first thing the car does is to make itself some wheels – or the first the Volvo does is that it becomes a car, and then when it has become a car, it becomes a Volvo, and when a car is created it creates its wheels and initializes them in some way. And then perhaps the wheels create their nuts. But the car does not have to know about the nuts, the car does only have to know about the wheel, that [the nuts] is something that the wheel uses. [S18]

This description category introduced the understanding of the differences between the two forms of diamonds symbols, the black and the white, and their relation to the concepts of aggregation and composition.

6. RELATED WORK

Understanding class diagrams is part of learning to design OO software. McCracken points out the complex nature of software design; it deals with ill-structured problems without well-defined start and end states, and there is no feedback during the process [18]. Eckerdal et al. [7] studied students' designs using data from an international multi-institutional study [23]. It was shown that 62% of the 314 informants could not design software at all, only 2% produced complete designs, and the design quality correlated with the number of completed computer science courses. In my opinion, this indicates that there is a potential to improve students' design skills and one piece in this puzzle may be to emphasize the ability to express and interpret design solutions using UML and class diagrams.

The present study shows a much varied understanding of the diamond symbols in UML. One explanation may be that UML lacks from precise definitions of these symbols, discussed by [1, 11, 12, 19, 21]. Winston et al. clarify the different semantic forms of whole-part relations in natural languages [26].

Empirical experiments have shown that size and structural complexity are early predictors for understandability and modifiability of class diagrams [9], and layout criteria and algorithms to make UML class diagrams understandable and aestetical are proposed [8, 22]. Eye tracking has been used to learn how people read and understand class diagrams. Interestingly, it seems that software engineers do not focus much on relations between classes [10], which the present study also indicates. Experienced programmers read designs from the center and outwards, whereas the novices read them like a book. Layout, stereotype information and coloring can improve understandability; however, the similar notations for aggregation and inheritance reduce readability [27], which was also observed in this study.

7. IMPLICATIONS FOR TEACHING

The phenomenographic outcome spaces provide valuable information for teachers who reflect on aspects of

learning software design. The insights about students' views are particularly valuable when evaluating education and it is even possible to take advantage of this knowledge in the classroom.

7.1 Variation theory

Variation theory can be applied to phenomeographic results to help students discern the more advanced ways to see various phenomena [16, 17]. This is an approach that can be applied to all outcome spaces.

As an example, we explore the possibility of using variations in order to facilitate understanding of the diamond symbols and aggregations. Three important dimensions of variation that could be seen in the corresponding outcome space that may help to discern the differences and similarities between aggregation and composition are: (1) dynamic behaviour of parts, (2) visibility of parts, and (3) ownership/life length of parts. The identified dimensions of variation allow a teacher to reflect on how to create learning situations that help students to see the important aspects by introducing variations.

Another aspect is that class names give students very strong associations and they give an overview perspective of the design and the relations between different classes in the diagram, e.g., the class names "Whole" and "Part". They may be so powerful that the relation symbols are not noticed because they are redundant in the context. One idea would be to vary the diagrams by omitting the class names, or using fictitious names. Then the relation symbols would stand out and the students would have to reflect on their meaning.

More thorough discussions and concrete examples of how phenomenographic results and variation theory can be used in the classroom can be found in [3, 4, 24].

7.2 Practicing design skills

In the informant collective, the most advanced way to see the purpose of class diagrams is the communicating and collaborating view. However, this view was not nearly expressed by all informants. This implies that if we want to prepare our students for professional design discussions, we must ensure that they get rich experiences from such discussions and that they learn a model language such as UML. This requires that students should work in teams and have free access to group rooms and whiteboards to be able to discuss and produce collaborative designs. We should also make the effort to discuss and assess designs including the formal syntax and semantics of the modeling language, and we should frequently use models in teaching and be explicit about the notation.

In addition to the phenomenographic results, an interesting observation from the interviews is that, overall, the informants were not very confident about UML and concepts such as diamond symbols and variants of aggregation; however, inheritance seems to be a well known concept and nearly all of the informants recognize the corresponding UML notation. This motivates the following question to research: *Is inheritance more important for students to master than object composition?*

8. CONCLUSIONS

The ability to model software visually and understand such models are important skills in the software industry. This paper describes a study of how a group of 20 students in the final part of their computing studies experience aspects of UML class diagrams. The research questions asked how students experience: the purpose of class diagrams, what class diagrams are, and what the black and white diamond symbols mean.

The informants described the use of class diagrams as (1) a documentation of code, (2) a means to make a software design, and (3) a means for an interactive design process together with the team.

The informants described a class diagram as a visualization of (1) classes with methods and instance variables and connections to other classes, (2) different kinds of relations between classes with many different meanings, and (3) hierarchic structures such as inheritance hierarchies and aggregations.

The informants described the black and white diamonds in UML as (1) relations, (2) a notation for when a class "has" another class, (3) a notation for when a class contains or consists of other classes, and (4) as a notation to announce aggregation, but the black and white diamonds discerns between composite aggregation and plain aggregation.

Only a minority of the informants could tell the difference between the white and black diamond, and many informants could not explain what the diamonds mean when they did not have help from descriptive class names. The diamond symbol (and the term aggregation) was often (vaguely) described as a "has"–relationship, but it was more rare that informants described it in terms of a "whole/part"-relation. The "has"–relation was sometimes explained by saying that a class has an instance variable that can refer to another class. This is true, but is also true for other types of relations than aggregation – for example an ordinary association.

Although the concept of inheritance and the related UML symbol were not subject to analysis, it was impossible to escape the observation that the informants could relate to it and they knew how to implement it in Java. S16 for example could specify inheritance when he drew the UML-diagram, but he could not specify other relations except for plain association-lines.

The fact that several informants said they did not know UML very well stresses the importance of improving the learning; in the software industry they will most certainly be expected to use UML or some other modeling language when they meet with colleagues to discuss design. I strongly recommend that students should be able to read designs and express their own design ideas using a basic set of UML constructs. The results from this study gives some implications for how the understanding can be improved. This may require the need to review the whole range of courses to ensure that the students will get the opportunity to learn about software design and design models.

9. REFERENCES

[1] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, and J.-M. Bruel. Formalization of the Whole-Part relationship in

the Unified Modeling Language. *Software Engineering, IEEE Transactions on*, 29(5):459–470, May 2003.

[2] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Longman, Reading, Massachusetts, 1999.

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

[4] J. Boustedt. A student perspective on software development and maintenance. Technical Report 2010-012, Department of Information Technology, Uppsala University, 2010.

[5] 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.

[6] T. A. Budd. *An Introduction to Object-Oriented programming*. Addison Wesley, Boston, 3 edition, 2002.

[7] A. Eckerdal, R. McCartney, J. E. Moström, M. Ratcliffe, and C. Zander. Categorizing student software designs: Methods, results, and implications. *Computer Science Education*, 16(3):197 – 209, 2006.

[8] H. Eichelberger. Aesthetics of Class Diagrams. In *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, page 23, Washington, DC, USA, 2002. IEEE Computer Society.

[9] M. Genero, M. Piatini, and E. Manso. Finding "Early" Indicators of UML Class Diagrams Understandability and Modifiability. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 207–216, Washington, DC, USA, 2004. IEEE Computer Society.

[10] Y.-G. Guéhéneuc. TAUPE: Towards Understanding Program Comprehension. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 1, New York, NY, USA, 2006. ACM.

[11] B. Henderson-Sellers and F. Barbier. Black and White Diamonds. In R. France and B. Rumpe, editors, *≪UML≫'99 – The Unified Modeling Language*, volume 1723 of *Lecture Notes in Computer Science*, pages 550–565. Publisher Springer Berlin / Heidelberg, 1999.

[12] B. Henderson-Sellers and F. Barbier. What is this thing called aggregation? In *Technology of Object-Oriented Languages and Systems, 1999. Proceedings of*, pages 236–250, Jul 1999.

[13] C. Larman. *Applying UML and Patterns*. Prentice Hall, Upper Saddle River, NJ, 2002.

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

[15] 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.

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

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

[18] 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.

[19] D. Milicev. On the Semantics of Associations and Association Ends in UML. *Software Engineering, IEEE Transactions on*, 33(4):238–251, April 2007.

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

[21] M. Saksena, R. B. France, and M. M. Larrondo-Petrie. A Characterization of Aggregation. In *In Proceedings of the 5th International Conference on Object–Oriented Information Systems (OOIS'98)*, pages 363–372. SpringerVerlag, 1998.

[22] D. Sun and K. Wong. On Evaluating the Layout of UML Class Diagrams for Program Comprehension. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 317–326, Washington, DC, USA, 2005. IEEE Computer Society.

[23] J. Tenenberg, S. Fincher, K. Blaha, D. Bouvier, T. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. Moström, M. Petre, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Shwartzman, B. Simon, C. Stoker, A. Tew, and T. VanDeGrift. Students designing software: a multi-national, multi-institutional study. *Informatics in Education*, 4:143–162, 2005.

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

[25] 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.

[26] M. E. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science: A Multidisciplinary Journal*, 11(4):407–444, 1987.

[27] S. Yusuf, H. Kagdi, and J. I. Maletic. Assessing the Comprehension of UML Class Diagrams via Eye Tracking. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 113–122, Washington, DC, USA, 2007. IEEE Computer Society.

APPENDIX

A. UML DIAGRAMS

The informants were asked to discuss four different UML-diagrams and what the diagrams modeled.

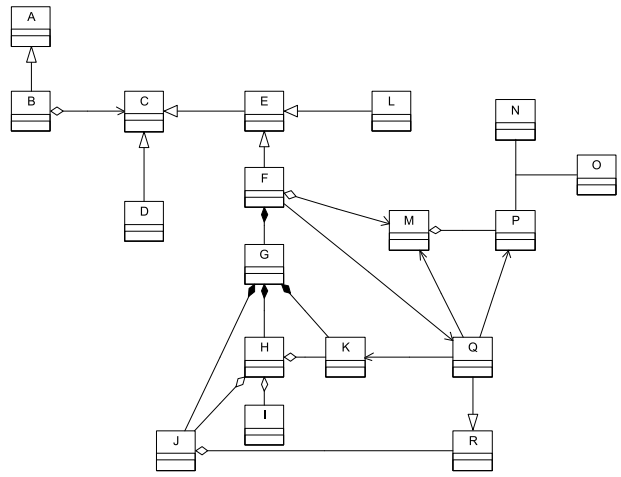

Figure 1: The first class diagram
shows formally named classes and their relations.

The first class diagram contains a number of class boxes labeled with letters (A - R) without information on attributes or methods (see Figure 1). The classes are connected by relation lines whose ends are decorated with symbols: open arrow heads for navigability, closed arrow heads for inheritance, black diamonds for composition, and white diamonds for aggregation.

When the informants talked about this diagram they could not get much help from the context to figure out what the relation symbols meant.

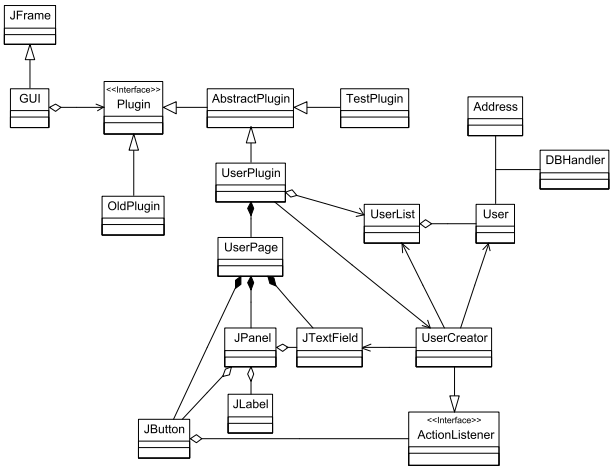

Figure 2: The second class diagram
shows real–named classes and their relations.

The second class diagram keeps the same structure as the previous, however, it introduces "real" class names instead of formal letters (see Figure 2). The class names provide a context making it easier to understand how classes are related. Some of the informants who were uncertain about the meaning of the relation symbols in the first diagram could now make qualified guesses, or confirm their earlier suggestions.

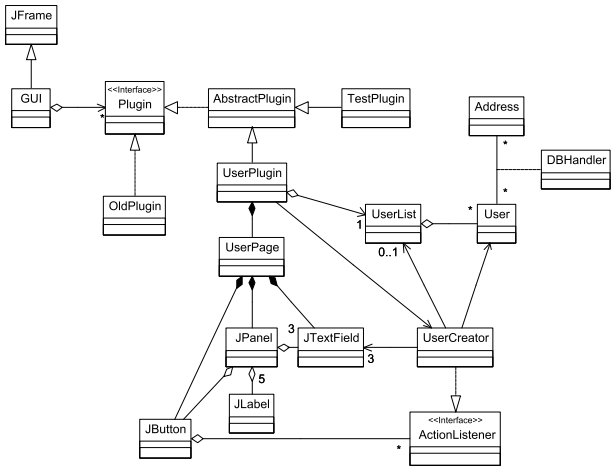

Figure 3: The third class diagram
adds multiplicities to some of the relations.

The third class diagram (see Figure 3) introduces multiplicity information to the relations, which indicates how many instances that may be involved on both sides of the relation. The multiplicity notation is not completely intuitive; however in combination with the class names, most informants could for instance say that the panel has five text fields or that the user list has an arbitrary number of users.

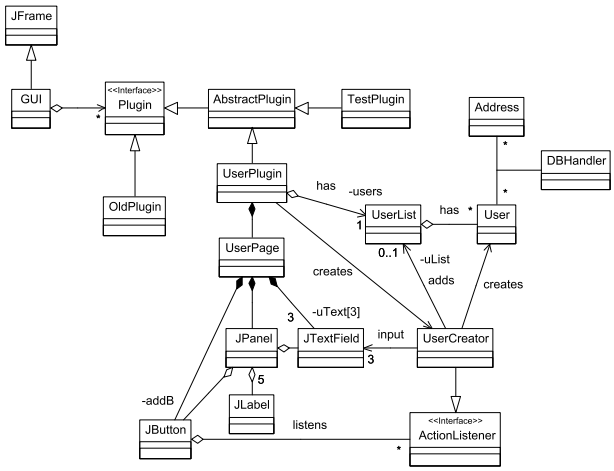

Figure 4: The fourth class diagram
adds roles and relation descriptions to some of the relations.

The last class diagram (see Figure 4) in the sequence introduces some relation names which may tell something about the relations' character. Furthermore, some of the relations have role names at the line ends. In some UML software packages, role names are placed there automatically using the names of the instance variables that are used to implement the relation. In this case a minus sign indicates that the instance variable is private. Most informants were confused by these notations and said that they had never seen this kind of annotations.