Abstract

Examples are important when we attempt to learn something new. To learn problem solving and programming is an acknowledged difficulty. Teaching and learning introductory object oriented problem solving and programming has been discussed extensively since the late 1990'ies, when a major shift to object orientation as first programming paradigm took place. Initially, this switch was not considered to cause any major problems, because of the accumulated knowledge for how programming should be taught. This turned out to be naive. Knowledge gained for the imperative paradigm did not apply well to the object oriented paradigm.

Because of its importance for the field of computer science, introductory programming education has drawn a lot of attention. Most of the research done in connection to object oriented problem solving and programming has been focused on students learning and the difficulty to acquire skills in programming.

Less investigated is the foundation of the educational mission, the characteristics of object orientation and how this is best supported by the educator. There is no obvious agreement of what the basics of object orientation are, especially not from an educational point of view.

In this thesis, two major aspects concerning the teaching of object orientation have been investigated: the definition of object oriented quality, specifically in examples for novices, and educators' views on aspects of object orientation. Based on research of how object orientation is characterised in literature and in software design principles, a set of concepts and principles are presented as a description of basic characteristics of object orientation. These are applied to the educational context, and a number of heuristics, called Eduristics, for the design of object oriented examples for novices are defined. The Eduristics are then used to discuss the flaws and shortcomings of common textbook examples, but also how the object oriented quality of examples can be improved.

To be able to evaluate the quality of examples, we initiated and participated in the development of an evaluation tool. This tool has been used to evaluate a number of examples from popular textbooks. The results show that the object oriented quality of examples is low.

To explore the ways educators view a number of aspects of object orientation and the teaching of it, ten interviews have been conducted. The results of this study show that the level of abstraction in the conceptual model of object orientation among educators is low, and that novices are not given any support for object oriented problem solving.

Sammanfattning

Exempel är viktiga när man ska lära sig något nytt och det gäller även när man ska lära sig programmera. Att lära sig problemslösning och programmering är erkänt svårt och det har föranlett många förslag på vad som är ett bra sätt.

Under 1990-talet skedde en större omläggning i programmeringsundervisningen världen över. Från att ha introducerat programmering i det imperativa/procedurella paradigmet övergick man till att använda objektorientering som första paradigm. Inledningsvis trodde man inte att det skulle skilja sig på något avgörande sätt från tidigare erfarenheter om hur programmering skulle undervisas. Detta visade sig vara en naiv föreställning. Mycket av den kunskap som ackumulerats kring den imperativa programmeringsundervisningen visade sig svår att överföra till objekt orientering. Omställningen har varit mödosam och är fortfarande inte genomförd fullt ut.

Programmering är centralt i datavetenskap, eftersom olika aspekter av programvarukonstruktion genomsyrar det mesta av verksamheten kring datorer. Utbildningsmässigt är en inledande kurs i problemlösning och programmering förutsättningen för vidare studier i ämnet. Detta gör att en hel del uppmärksamhet har riktats mot problemlösning och programmering.

Det mesta av den forskning som finns gjord i anslutning till objekt orienterad problemlösning och programmering har varit fokuserad på nybörjares lärande och problem att komma in i programmerandet.

Mycket lite finns gjort när det gäller själva utgångspunkten för undervisningen om objektorientering, nämligen vad som är centralt i objektorientering och på vilket sätt det ska manifestera sig i undervisningen.

I det här arbetet har två huvudaspekter av objektorientering i undervisningssammanhang undersökts: definitionen av objektorienterad kvalité, specifikt i exempel för nybörjare, samt vilken syn lärare har på olika aspekter av objektorientering.

För att möjliggöra detta har vi undersökt hur objektorientering beskrivs i litteraturen och i vedertagna design-principer som används i programvaruutvecklingssammanhang. Baserat på resultatet av den undersökningen har vi använt en uppsättning koncept och designprinciper för att definiera vad som är karakteristiskt för objektorientering. Med detta som utgångspunkt har vi applicerat definitionen av objektorientering till undervisningssammanget och definierat ett antal heuristiker specifikt för konstruktion av objektorienterade exempel för nybörjare.

Parallellt med detta arbete deltog vi i utvecklingen av ett utvärderingsverktyg för att värdera objektorienterade exempel för nybörjare. Detta verktyg har använts för en större utvärdering av exempel hämtade från populära läroböcker. Resultaten från denna studie visar att exempel generellt sett håller låg objektori-

enterad kvalitet. Vi har också visat att exempel som värderas högt, uppfyller våra heuristiker och att exempel som värderas lågt strider mot desamma.

För att utforska hur lärare ser på objektorientering och hur de resonerar kring strategier för att lära ut objektorientering, har vi gjort tio intervjuer med lärare i gymnasieskolan och på universitetsnivå. Resultaten visar att den konceptuella modellen för objektorientering är mycket enkel i förhållande till den komplexitet som ofta anses kännetecknа paradigmet. Dessutom, ges i stort sett inget stöd för nybörjaren vad gäller att förstå och lära sig problemlösningsansatsen, som ofta upplevs som väsensskild från hur man i normala fall löser problem.

List of Papers

This thesis consists of an introduction to the research area and the following papers:

Paper I: Börstler J., Nordström M., Kallin Westin L., Moström J-E., and Eliasson J., "Transitioning to OOP/Java – A Never Ending Story", In *Reflections on the Teaching of Programming*, M. Kölling, J. Bennedsen, and M. Caspersen, Eds. Lecture Notes in Computer Science, vol. LNCS 4821. Springer, 86-106.

Paper II: Nordström M., and Börstler J. (2010) Heuristics for Designing Object-Oriented Examples for Novices. Submitted to *ACM Transactions on Computing Education (TOCE)*

Paper III Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Kallin Westin, L., Moström, J.-E., and Caspersen, M. E. Evaluating OO example programs for CS1. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 47–52, New York, NY, USA. ACM.

Paper IV: Börstler, J., Nordström, M., and Paterson, J. H. (2010). On the quality of examples in introductory java textbooks. *The ACM Transactions on Computing Education (TOCE)*, Accepted for publication.

Paper V: Nordström M. Educators' views on object orientation. Submitted to *Computer Science Education*.

Paper VI: Nordström M. Educators' strategies for OOA&D. Submitted to *ACM Inroads*.

Paper VII: Nordström M., and Börstler, J. Improving OO Example Programs. Submitted to *IEEE Transactions on Education*.

Reprints were made with permission from the publishers.

Author's Contributions

Paper I: This is the story of ten years of experience of switching from imperative/procedural programming to object oriented programming in our introductory courses. All co-authors collected data, contributed with their experience, and took part in the formulation of a number of educational principles for teaching object orientation to novices. Most of the writing was done by Börstler, Kallin and Nordström.

Paper II: Summarising the major findings and results of my Licenciate Thesis, and further discussions in connection to the evaluation of examples. The major part of the work was done by me.

Paper III: This paper is the result of the initial work to design an evaluation tool for object oriented examples. Seven danes and swedes, all experienced in teaching introductory programming in several paradigms took part in the development of a pilot-tool. All co-authors participated in the development of the assessment tool, and analysis was done collectively. Statistics was done by Lena Kallin Westin. Writing was mainly by Börstler and Nordström.

Paper IV: Based on data collected for an ITiCSE workshop (Börstler et al., 2009), an extended analysis and investigation of different categories of examples was carried out. Co-authors contributed in different areas. My focus in this work was the object oriented qualities, mainly on First User Defined Classes (FUDCs).

Paper V: Based on ten interviews, this paper categorises educators' personal views of Object Orientation. I developed the structure for the research area, planned and performed the interviews. The analysis of the textual data is all my work.

Paper VI: Teaching Object Oriented Analysis and Design seem to be a problem to educators. Whether explicit or not, how we exemplify different aspects of object orientation will affect students notion of how to design their solutions. In this paper a categorisation of used strategies for OOA&D is presented. The structuring of the research area, the planning and the carrying out of the interviews, was entirely my work, as well as the analysis of the textual data.

Paper VII: The use of Eduristics to show how it is possible to discuss object oriented quality and to improve examples for novices. All work has been done jointly by the co-authors.

Other Publications by the Author

The work presented in this thesis is partly based on ideas and work presented previously:

- Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., and Thomas, L. (2009). "An evaluation of object oriented example programs in introductory programming textbooks". *SIGCSE Bull. Inroads*, 41:126–143.

- Börstler, J., Christensen, H. B., Bennedsen, J. Nordström, M., Kallin Westin, L., Moström, J.-E., and Caspersen, M. E., Evaluating OO example programs for CS1*, Proceedings of the 13th annual conference on Innovation and technology in computer science education,* 2008, Madrid, Spain June 30 - July 02, 2008 Pages 47-52

- Börstler, J., Nordström, M., Kallin Westin, L., Moström, J.-E., Christensen, H. B., and Bennedsen, J. (2008) *An Evaluation Instrument for Object-Oriented Example Programs for Novices*, Technical Report UMINF-08.09, Dept. of Computing Science, Umeå University, Umeå, Sweden

- Börstler, J., Caspersen, M. E., and Nordström, M. *Beauty and the beast—toward a measurement framework for example program quality.* Technical Report UMINF-07.23, Dept. of Computing Science, Umeå University, Umeå, Sweden, 2007.

- Börstler J., Caspersen M.E., and Nordström M. *Beauty and the Beast* - openspace on Educators symposium, At the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA'07, Montreal, Quebec, Canada October 21 - 25, 2007

- Nordström, M. *PigLatinJava - troubleshooting examples.* Technical Report UMINF-07.26, Dept. of Computing Science, Umeå University, Umeå, Sweden, 2007.

- Eliasson, J. , Kallin Westin, L. and Nordström, M. Investigating students' confidence in programming and problem solving, *Proceedings of the 36th ASEE/IEEE Frontiers in Education Conference (FIE2006)*, San Diego, California, USA, October 28-31, 2006 pages M4E-22-M4E-27

- Eliasson, J. , Kallin Westin, L. and Nordström, M. *Investigating students' change of confidence during CS1 - four case studies.* Technical Report UMINF-106.13, Dept. of Computing Science, Umeå University, Umeå, Sweden, 2006.

Acknowledgements

It is a privilege and a great pleasure to thank the many people who made this work possible.

First and foremost, my colleague and for the last three years my supervisor, Jürgen Börstler. Your enthusiasm, humor, creativity, and very strong mind to get things finished, has pushed me through this adventure. *Augen zu und durch!* has become my motto... Thanks to You.

Numerous people have contributed to my research. Jens Bennedsen, Jürgen Börstler, Michael Caspersen, Henrik Bærbak Christensen, Johan Eliasson, Jim Hall, Thomas Johansson, Lena Kallin Westin, Jan-Erik Moström, Jim Paterson, Kate Sanders, Carsten Schulte, and Lynda Thomas are co-authors of one or more of my publications. I have really enjoyed, and benefited, from these collaborations. I do hope to get the privilege to work with You again!

Of course I am deeply indebted to the generous fellow educators who were willing and able to find time to share their thoughts and experiences and to participate in my interviews. I really enjoyed listening to You.

On a more personal level:

To my colleagues at the department, thank you for making the department such a nice place. Solving crossword puzzles helps relaxing, and I think we have gotten really good at it :-) Since I have been with the department for many years, I go a long way back with many of you, and I want you all to know how much I appreciate your company, and your willingness to assist me.

LenaP, to me you are a hero and a shoulder to lean on. You have offered help and advice on so many occasions, and supported me all these years. Thank You!

LenaKW, optimistic, ready to take on any challenge, and extremely generous! Thank You for being close!

Kristina, You have carried me those times I needed it the most... God bless You!

I would not cope without friends that support comfort and encouragement when needed: Bönorna Kristina, Gun-Britt and Britta. Thank You for caring!

I have a large network consisting of my mother Karin, my mother- and father-in-law, Birgit and Karl-Anders, my sister Kina and her family, my brothers- and sister-in-law, and their families: they provide me with a large caring family, and a lot of birthdays.

To my closest family who keeps me down to earth, and provides me with real life: I owe you for putting up with me the last year, months, weeks, and days of this work. Torbjörn, Emilie, Johanna, Jakob, Ellen and Frida, I love You!

Finally:

Looking back on this last year, finishing this work, I have realised some similarities with something else I have experienced.

Being pregnant (a condition I am slightly familiar with), resembles finishing a PhD. In the beginning everything is fantastic. You enjoy every minute, and you are privileged with the opportunity to be introvert and dwell upon the circumstances of your condition. You know what is waiting at the finish, but it is still very distant, and almost unreal. Easy to ignore.

Time passes, and suddenly that particular event waiting, seems much more likely to eventually take place. Not soon, but getting slightly worrying. You know that there is some sort of effort involved.

Some more time passes, and now it is approaching fast. Too fast. You realise that it is inevitable. It IS going to happen! Anguish! Increasing anguish...

And then suddenly, you can not stand this anguish anymore, and you start to wish for it to take place, to get it over with. The sooner the better!

It is impossible to envision what life might be afterwards. You don't even care...

Soon I will know what life after THIS experience will be...

BTW, I do not expect it to be even close to meeting my newborn children, THAT was nothing short of a miracle!

Umeå, December 2010
Marie Nordström

Contents

Chapter 1

Introduction

In the late 1990's my department, like many others, decided to switch language for the introductory programming courses. Switching from a traditional imperative language (Pascal) to an object oriented (Java), did not seem like a big deal initially. In the beginning it was merely a change of language. We added objects to a well established line of presentation, starting out with the usual elements introducing programming to novices. Variables, data types, statements, expressions, selection, loops, conditions, "procedures", and parameters, were thoroughly discussed before the introduction of objects. Soon we discovered that the solutions and programs students produced mostly resembled Pascal code in Java-syntax. No wonder, since that was basically what we taught! Now a long journey of stepwise improvements of the course started, and eventually we arrived at an object-first approach. The line of presentation was developed in close connection to assessment and examination, to form a coherent approach to provide conceptual understanding as well as the development of skills. An important component of the educational design for teaching object oriented problem solving and programing, has been the use and development of an explicit method for object oriented analysis and design.

As the reasons for choosing this teaching approach, with an explicit emphasis on objects from the very beginning, became articulated, the demand for certain features in examples, exercise and assignments became obvious. Textbooks introducing object orientation to novices stated that they were "Object first", "Truly object oriented", "Focusing on objects", and similar claims, but still showed examples that were contradictory to characteristics of object orientation, despite the advertised goals. Good examples to support the introduction of object orientation through objects is hard to find, and to design. Part of this problem is the particulars of the educational context. Examples have to be simple, plausible, and exemplary to show the essence of the paradigm, and to serve as role-models for the novice.

The search for suitable object oriented examples, and the search for research to support the design of object oriented examples has been tedious and close to fruitless. This frustration is the reason, and starting point, for my interest in the object oriented quality, and design, of object oriented examples for novices.

1.1 Learning to Program

Programming is not a trivial task to learn (McCracken et al., 2001; Robins et al., 2003). Novices are struggling, and the drop-out rate is high (Bergin and Reilly, 2005). Though largely debated, object orientation is commonly used for introducing problem solving and programming to novices (de Raadt et al., 2004; Schulte and Bennedsen, 2006). How to do this is not straightforward. In addition to the general difficulties, the introduction to programming seem to be more complicated when using the object oriented paradigm, compared to the imperative/procedural paradigm (Sajaniemi and Kuittinen, 2008). There are many suggestions to what might be the cause of this. One reason could be that the decentralised flow of control is more difficult to grasp (Du Bois et al., 2006). Following the execution of statements in object oriented code is more cognitively demanding than a linear sequence of instructions. Another possible explanation is that object oriented languages are more complex than procedural ones (e.g. (Caspersen, 2007)). The paradigm is relatively new, and this might mean that the pedagogy is not mature enough yet, and that this will improve with time. There seem to be no simple explanation for the difficulties of learning to program, and no simple solution!

Computer science education research (CSER) is a young discipline. Setting the grounds for the development of CSER Fincher and Petre (2004) points out that the area is still struggling to find the shape of our literature. They claim that the major part of research papers are "practice" papers. These are descriptive, practice-based, experience papers, and weak on argumentation. An analysis of an objects-early debate on the SIGCSE mailing list[1] shows that the arguments among CS educators are not based on research on novice programmers (Lister et al., 2006). The investigation of claims from the debate, shows that there is no evidence to support them. Commonly believed myths dominate the discussion rather than informed research results.

Introductory programming courses, commonly called CS1, are important for many reasons. They may be considered providers of a basic tool-of-the-trade for most areas within computer science. Research on the teaching and learning of programming is thoroughly reviewed by Caspersen (2007) and Bennedsen (2008). General theories of teaching and learning are surveyed, and applied to the area of teaching and learning introductory programming. The results are unambiguous: students have a hard time learning to program, and the major problem is composition, not learning syntax (Caspersen, 2007). Years of experience and a large amount of research have laid the ground for the theory of a model-based approach, strongly supporting the idea of teaching with an explicit focus on the programming process rather than exclusively working with the introduction of different language constructs (Bennedsen, 2008).

The meta-analysis by Valentine (2004), is of some interest for the present work. The analysis is restricted to all articles regarding CS1/CS2[2], published in the SIGCSE Technical Symposium. The contributions are categorised in a six-fold taxonomy:

[1] The ACM Special Interest Group on Computer Science Education (SIGCSE) maintains two moderated mailing lists for announcements and discussion of topics of general interest to SIGCSE members. Subscription is limited to SIGCSE members, and posting is restricted to subscribers.

[2] CS2 is usually the course introducing Data structures and Algorithms.

- Marco Polo: "We tried this and we think it is good". (27%)
- Tools: software, a paradigm or an organizing rubric for an entire course, etc. (22%)
- Experimental: articles that made any attempt at assessing the "treatment" with some scientific analysis. (21%)
- Nifty: assignments, projects, puzzles, games and paradigms. Attempts to find innovative, interesting ways to teach students abstract concepts. (18%)
- Philosophy: discussions along philosophical and educational lines.(10%)
- John Henry: attempting almost impossible, and often unbelievable, approaches in teaching. (2%)

The analysis spans 20 years (1984-2003), and in all, 444 papers were analysed.

The contribution of questions treated scientifically is rather limited, "Experimental" is 21%. The category is regrettably not divided into qualitative and quantitative research respectively. This category is also heavily criticised by Randolph (2007), since it "is so broad that it is not useful as a basis for recommending improvements in practice".

However, being interested in precisely educational research for CS1/CS2, we find the results indicative and the distribution of papers disturbing. Without having quantitative data to state it, a strong impression is however that qualitative research is scarce, at least within this field. A simple test searching the ACM Digital Library for the word "qualitative" within the proceedings of SIGCSE Technical Symposium Proceeding (1984–2003), yields 15 hits (out of 1663). This means that less than 1% of the publications in SIGCSE Technical Symposium Proceeding even mentions the word qualitative. This indicates that most computer science education research, at least the research analysed in (Valentine, 2004), is at best quantitative.

Regardless of methodology and line of presentation, it is well known that examples are important for learning. In the educational context, examples are small and often restricted by the the novice's limited frame of reference. This makes the design of examples difficult, since they must be easy to understand, but still exemplary to act as role-models for the paradigm. The quality of examples and their impact on learning are areas not researched. Before evaluating the impact of different teaching approaches, we must make sure that examples are properly designed to expose the characteristics of object orientation.

1.2 The Problem

There are somewhat conflicting requirements for object oriented principles and examples, when it comes to the specific needs of the educational situation. It is common for educators to have to compromise with respect to ideals and preferred qualities when teaching. Examples should preferably be clear-cut, and isolate a certain feature to be demonstrated. To keep the cognitive load down, it is common to try to keep the number of lines of code down. By keeping the code to a minimum, often with a one-class or one-page limit, some of the important characteristics may be difficult to demonstrate. If we want examples to show novices a realistic need

for object oriented software development, reasonable contexts are hard to find . We must make problem solving plausible, both in terms of problems as well as in the design of solutions. Novices often struggle with the mapping of domain knowledge to implementation, and this must addressed. Furthermore, novices initially have a limited set of general programming concepts, which restricts the amount of concepts and constructs available for the educator. All of this makes the design of examples difficult.

The object-orientedness of examples has been discussed (Westfall, 2001; CACM, 2002; Dodani, 2003; CACM, 2005), and particularly the very first examples shown can be problematic. It is hard to find suitable first examples that are simple enough, but still object oriented. Just putting code into a class does not make it object oriented. A common template for the first example appearing in popular textbooks is shown in Listing 1.1.

```
public class HelloWorld
{
  public static void main (string[] args)
  {
    System.out.println("Hello, world");
  }
}//class HelloWorld
```

Listing 1.1: First example.

This class is non-typical of object orientation for several reasons. There are, in fact, a number of severe contradictions to the ideas of object orientation:

- the class is not an abstraction
- the class does not model an entity in a problem domain
- there are no objects instantiated
- the class contains only one method, `main`, that is not called explicitly by any client
- the method, `main`, does not represent any service provided, it is not representing a behaviour of an object
- the method `main` is `static` which means that it does not belong to an object, but is common to all the objects of this class
- the only method explicitly called, `System.out.println()`, is static and is not the service of an object, i.e. no collaborating object is instantiated

All in all, this example does not exemplify many characteristics of object orientation.

Besides the problem of finding suitable problem domains and contexts for the introductory examples, it is difficult to illustrate the execution of a program. Creating a lot of objects within a client-program will not necessary show the programming novice what happens when the program is executed. A common first example of a class is `BankAccount`, but for objects of this type of class it is hard to demonstrate the effect of method-calls or state-changes in any natural way.

Because of this, it could be tempting to choose something that might be illustrated graphically on the screen, to see some effect, or trace, of the execution.

There are many, many versions of the `HelloWorld`-example, and it is not uncommon to use graphical elements to make things a little bit more attractive to the novice.

Executing such a program could result in the output shown in Figure 1.1.

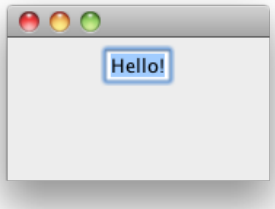

Figure 1.1: Execution of the `Greeting` class

This output was generated by the `HelloWorld`-example shown in Listing 1.2.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Greeting extends JFrame
{
  private JTextField textField;
  public static void main (String[] args)
  {
    Greeting frame = new Greeting();
    frame.setSize(300, 200);
    frame.createGUI();
    frame.setVisible(true);
  }
  private void createGUI()
  {
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    Container window = getContentPane();
    window.setLayout(new FlowLayout() );
    textField = new JTextField("Hello!");
    window.add(textField);
  }
}//Greeting
```

Listing 1.2: Second example.(Bell and Parr, 2010)

In this example there are some further confusing issues:

- the class is extending another class. No explanation of what this means is given in the accompanying text.

- the class is creating an instance of itself. If we claim that everything is about objects, it does not seem reasonable that something that do not exist can create an instance of itself.
- the class has only one method, and it is private!, to create a simple graphical window (which is rather complex in Java)
- there are many unfamiliar things: libraries imported, classes used, methods for advanced features of frames, constants not defined within this class etc.
- to a novice it must seem complicated to generate a simple output on the screen, requiring all those window-handling operations

It is hard to see the purpose of the abstraction `Greeting`. The name (which is really important to convey the essence of the object), indicates that greetings can be created and used. This raises several questions. Why would anyone need greetings-object, what is the context in which these objects would appear? What kind of behaviour would clients assume of objects of this class? One reason for classes is to be able to instantiate many objects of a certain kind, but in this case it is hard to see the need for several objects. They would all look the same, generate the same static output, and clients can not interact with them. Showing the methods of these objects, reveals that there are no public accessible methods, so anyone creating a `Greeting`-object, can not interact with it, and it does not generate any output. Inspecting the class itself, the novice is offered many methods that are difficult to associate with a greeting, since `Greeting` is inheriting from `JFrame`, see Figure 1.2.

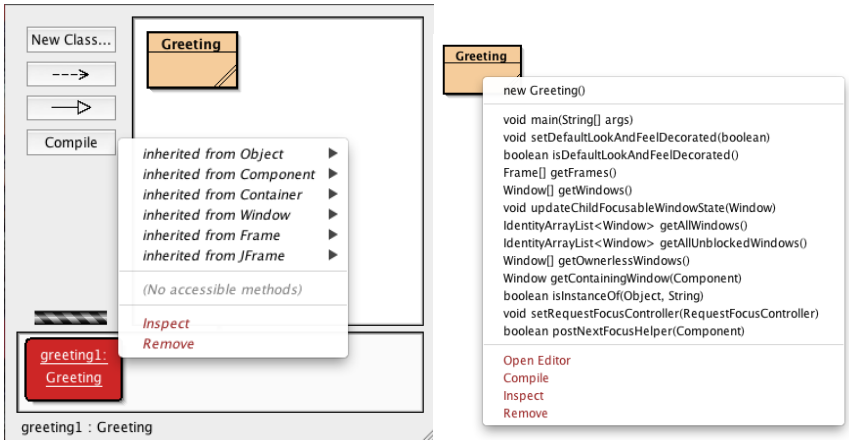

Figure 1.2: Available methods in `Greeting` class (screenshots from BlueJ).

These examples may seem trivial, with obvious and non-obvious deficiencies, and/or non-object oriented. Superficially they might be placed in another context, or look slightly different, but the approach and functionality resembles `HelloWorld`.

1.3 Outline of this Thesis

The research presented here is addressing the subject of object oriented quality in introductory object oriented programming education. The thesis consist of an introductory part and seven papers. The content of the thesis is organised as follows: Chapter 2 gives an introduction to the general research on learning from examples. In Chapter 3 some of the research on teaching and learning object orientation is reviewed.

In Chapter 4, we discuss the definition of object oriented quality, both in general and in the specific context of teaching novices. Based on this work, heuristics to be used in the design of examples for novices are then discussed in Chapter 5.

Since educators are the ones presenting examples to novices, and their personal views on different aspects of object orientation will affect their presentation, it is important to listen to them. The analysis of their stories is presented in Chapter 6. Chapter 7 contains conclusions and directions for further research. Finally a summary of the papers is given in Chapter 8.

The seven papers included in the thesis (I-VII) address the question of object oriented quality from different perspectives. These perspectives and their treatment in the papers respectively are illustrated in Figure 1.3 and shortly described below.

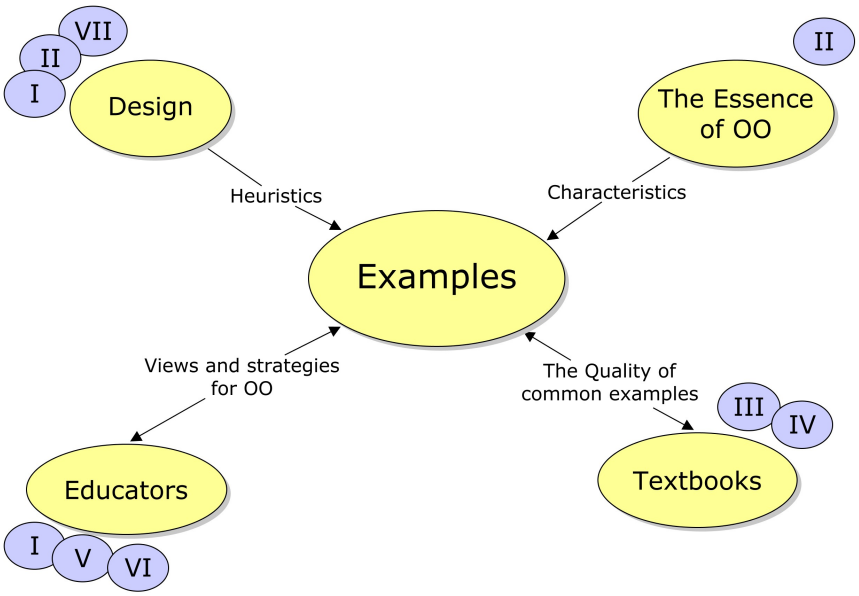

Figure 1.3: Different aspects of examples, and related papers in Roman numerals.

The Essence of OO

To be able to discuss object oriented quality of examples, it is necessary to state what the essence of object orientation is. The characteristics was established based on an investigation of how object orientation is defined literature, in terms of concepts and design principles used by the software developing community. This

is discussed in Paper II - Heuristics for designing OO examples for novices.

Design

Based on the investigation of object oriented characteristic, the results were then applied to the educational situation. The particular needs of a novice being introduced to object orientation was taken into account and a number of heuristics for the design of object oriented examples for novices were developed. The discussion on a proper mindset for teaching object orientation and the consequences for examples is initiated in Paper I - Transitioning to OOP/Java – A Never Ending Story, and the design of examples is specifically discussed in Paper II - Heuristics for designing OO examples for novices and Paper VII - Improving OO Example Programs.

Textbooks

Textbooks are a major source, for educators when searching for examples to use in introductory courses, and for students to find solutions to common programming problems. This makes the object oriented quality of textbook examples critical. A number of textbook examples have been evaluated through a large-scale study, using an evaluation tool designed to capture technical, didactical and object oriented qualities Paper III - Evaluating OO example programs for CS1 and Paper IV - On the Quality of Examples in Introductory Java Textbooks.

Educators

Another aspect of object oriented quality is how educators themselves view object orientation. What are the characteristics considered important to convey? The research question *How educators view OO*, was thematically operationalised according to four themes; the paradigm itself, the concept of an object, examples and object oriented analysis and design. Each theme was investigated from three different perspectives, the teacher's personal view, the teacher's view of students' difficulties and the teacher's choice of methodology to address those difficulties. Data has been collected through ten interviews and qualitative content analysis has been conducted. The results from these interviews are presented in Paper V - Educators' Views on OO, Objects and Examples and Paper VI - Educators' Strategies for OOA&D.

Chapter 2

Learning from Examples

Research on examples is mainly focused on cognitive and learning aspects.

Several studies conclude that people rely heavily on examples for learning (e.g. Pirolli and Anderson, 1985; Lahtinen et al., 2005), but that the level of generality to be transferred depends on the generality of the example solution. The use of examples is manifold: the explanation of a certain phenomenon, the gaining of some skill to solve similar problems, the generic understanding of how this fits into a bigger whole, and is related to other concepts within the framework.

Based on theory and research findings from instructional systems, cognitive science, and developmental psychology, concept-learning can be viewed as consisting of two cognitive processes: forming conceptual knowledge and developing procedural knowledge (Tennyson and Cocchiarella, 1986). Conceptual knowledge entails understanding the operational structure of the concept itself, as well as the relationship to associated concepts, and can be said to be the storage and integration of information. This knowledge is used to develop procedural knowledge, by retrieving knowledge for solving problems. The two processes are interacting in the formation of concepts.

Learning to program is a complex process, and as in many other learning situations novices combine many cognitive activities (VanLehn, 1996). They have to develop mental representations related to program design, program understanding, modification, debugging and documentation. The processes of using examples to learn are: *retrieval*, *mapping*, *application*, and *generalisation*.

> *Applying a principle or example consists of retrieving it, placing its parts into correspondence with parts of the problem [...], and drawing inferences about the problem and its solution on the basis of the problem's correspondence with the principle or example. After applying the principle or example, subjects may generalize it.* (VanLehn, 1996, p518)

When learning and acquiring skills in programming, reading and tracing code is important. The formation of a more complex understanding of concepts, and pieces of code, is made through the development of successively more abstract schemas or plans (Rist, 1989). There are two cognitive processes involved when we are trying to comprehend program code in examples, *chunking* and *tracing* (Cant et al., 1995).

Chunking means recognising groups of statements and labelling them with symbols or single abstractions. This recognition can be performed in levels and

produce a *multi-levelled, aggregated structure over several layers of abstraction* (Cant et al., 1995). Comprehending the chunks is important in this process.

Tracing involves quickly scanning through code in order to identify chunks. Often information about a certain entity is scattered and tracing is needed to collect it. In itself, the process has no connection to comprehension of the traced code.

Cant et al. (1995) use chunking as a model for recognising "program plans", which consists of both an idea of control flow and of variable use. It is therefore crucial to be careful in the design of examples, to avoid adding to the cognitive complexity of reading code. In this respect, choice of identifiers, consistency in the use of syntactical elements, proper and adequate commenting, and decomposition are examples of things that are likely to affect the readability of the example (Börstler et al., 2007). The forming of concepts is an integral part of how we structure knowledge. The ability to abstract common characteristics from instances to classify entities is vital for the cognitive work in building conceptual models, *classification provides "maximum information with the least cognitive effort."* (Parsons and Wand, 1997). Being able to read code is important for several reasons. Reading examples to learn new concepts or features of object orientation, learning standard solutions to standard problems on several levels of abstraction, and bug fixing are some of the every-day activities of a novice, but reading code is problematic to many novices (Lister et al., 2004).

Learning from examples seem to be most efficient when the knowledge gained by studying the example is immediately used to solving a new problem. The effect of different combinations of studying and solving problems has been investigated by Trafton and Reiser (1993). Studying an example and then solving a similar problem, with access to the studied example, seems to be the most favourable way. A suggested reason is that the number of rules formed are fewer, than when the example is blocked from access.

Research also shows that students who use the examples/exercises to elaborate on the conditions and consequences of each step in the example, to explain how and why things function the way they do, perform better. To investigate the role of examples through self-explanations, Chi et al. (1989) compared and contrasted how examples were used by good and poor students. The classification of students was based on the overall results in a given exam. The students' self-explanations were used to reveal their understanding, by measuring whether or not they knew

1. the conditions of application of the actions
2. the consequences of actions
3. the relationship of actions to goals
4. the relationships of goals and actions to natural laws and other principles

The results show that good students met all the above forms of understanding. They used the examples as a reference, usually rereading only a few lines of an example. Poor students seldom explained the example to themselves, and if they did, the explanations were restricted to details and not concerning concepts or

general principles. Furthermore the poor students reread larger portions of an example, than did the good students, in search of solutions. A closer examination of self-explanations shows that they are no guarantee for better learning (Renkl, 1997). Most of the students tend to use passive or superficial explanations. The successful students were characterised as either anticipative reasoners or principle-based explainers.

Worked-out examples is a way of providing the learner with an expert's process of problem solving. They usually include a problem statement and a procedure that shows the approach of solving the problem. The problem solving procedure is typically shown in a step-by-step fashion. The use of worked-out examples is discussed in several papers, and is argued to be more favourable, in terms of cognitive load, than the use of regular examples when acquiring cognitive skills (Sweller and Cooper, 1985; Sweller, 1988; Sweller et al., 1998). Important is also the sequencing of examples and practice problems (Pirolli and Anderson, 1985).

The process of acquiring a cognitive skill through worked examples involves four stages (Atkinson et al., 2000). Initially the learner solves problems by analogy, trying to match known examples to the problem at hand. Then the learner starts to develop abstract declarative rules to be used in problem solving. After much practice, the problem solving becomes more automated and the rules are no longer used, since the verbal memory evolves into a procedural form of memory. Finally the collection of examples and the accumulated practice makes it possible to retrieve solutions directly from memory.

The further development of worked examples introduces fading (Renkl et al., 2002). Fading means successively removing more and more worked-out solution steps as learners transition from relying on examples to independent problem solving. Backwards fading means successively excluding explanations from the end of the worked-out example, which is more effective than removing explanations from the front, and has proved to have an effect, at least for near-transfer problems. The lack of effect on far-transfer problems raised the question whether the fading could be combined with other instructional approaches, to foster far-transfer in particular. To investigate this, fading worked-out examples was combined with prompting for explanations. Although no interaction between the use of fading and self-explanation prompts could be established, both instructional approaches proved to produce an effect that is statistically significant even for far-transfer problems (Atkinson et al., 2003). Neither backwards fading nor prompting induced any significant increase in learning time, and are both simple and easy-to-implement procedures through computer-based learning environments. Theoretically, prompting and self-explanations can be connected to Vygotsky's Zone of Proximal Development (Vygotsky, 1978). The use of prompting is however, as can be expected, highly sensitive to implementation. Some studies show negative results, partly explained by the cognitively inadequate implementation supplied by computer environments (Atkinson et al., 2003).

Similar ideas are seen in research on general instructional design. In this line of research the notion of *best examples* are considered an instructional design variable for teaching and learning concepts. The *best example* should represent an average, central, or prototypical form of a concept, and contribute to the initial encoding of conceptual knowledge . This example should then be accompanied by expository and interrogatory examples, to form procedural knowledge (Tennyson

and Cocchiarella, 1986). Expository instances of the concept, are examples and nonexamples that systematically organize and present the dimensionality of the concept. Interrogatory instances are examples and nonexamples that use questions to present the concept. The development of procedural knowledge is facilitated if the learner is comparing and contrasting expository examples, since they should provide richness to the conceptual knowledge.

An interesting collection of *harmful* examples has been collected by Malan and Halland (2004). They identify four common pitfalls to avoid when designing examples: examples that are too abstract, or too complex, examples applying concepts inconsistently, and examples undermining the concept introduced.

Examples can also contribute to students adapting several types of poor learning behaviour, as investigated in (Carbone et al., 2000, 2001). Poor learning tendencies was initiated by tasks that initiated superficial attention (copy and paste), or impulsive attention (too unfocused, too much). The third learning behaviour was students getting stuck due to inadequate strategies for initial design, coding, and debugging.

Summary

There is research that provides results and theories for the general discussion on the use of examples. But, when it comes to the specific area of learning problem solving and programming in general, and within the object orientated paradigm in particular, the application of these theories has not been researched. We all know that we should present the novices with "good" examples, but the notion of what constitutes a good example has not been discussed. If the examples used, in some aspect, are inconsistent with the general idea of what we are teaching, it does not matter how much work we put into the instructional design. It is therefore necessary to base the design of our examples on explicitly formulated qualities that are in line with the characteristics of object orientation.

Designing examples to illustrate object oriented concepts or features is definitely a craft that demands caution and awareness. Unless carefully designed, the examples may very well be counterproductive. Research makes it clear that the very first examples are critical in providing the first cognitive encoding for the conceptual model.

Chapter 3

Teaching and Learning Object Orientation

Programming is of great importance to the computer science community. Therefore, specific attention is given the particular field of teaching and learning programming. In this chapter some results are presented, for extensive surveys of this field see the thesis work by Caspersen (2007) and Bennedsen (2008).

3.1 Teaching object orientation

Changing paradigm from imperative/procedural to object orientation for the introduction of programming to novices, was initially an underestimated teaching challenge, and classic results on on programming education, e.g. mental representation of programs, the understanding of the notational machine and the overall approach to program design, were assumed to apply equally well to object oriented programming. Sajaniemi and Kuittinen (2008) discuss this lack of research-based approaches to teaching object oriented programming, and conclude that there is no evidence that favours using object orientation as first paradigm. On the other hand, Lister et al. (2006) found no support for the claim that object orientation would be more difficult to learn than imperative/procedural programming.

Educators themselves are an integral component i programming education. It is therefore important to investigate educators' experiences if teaching object orientation and how they perceive the difficulties of it. In an on-line survey among educators, Dale (2006) used an open ended question to ask educators: *"In your experience, what is the most difficult topic to teach in CS1?"*. Out of the 351 responses, four categories of answers were found through content analysis. The categories were:

Problem Solving and Design This category comprised comments describing higher-level thinking, such as: problem solving, algorithm design, abstraction, object oriented problem solving and design concepts.

General Programming Topics Comments in this category focused on specific programming constructs. Arrays, parameters and parameter passing, selection, looping and I/O were topics mentioned.

Object Oriented Constructs The most common concepts in this category were inheritance and polymorphism, but almost all basic constructs were mentioned, e.g. user defined classes and variables.

Student Maturity Educators considered students to be ill prepared. Many mentioned that problem solving was a skill lacking in the student population. One respondent summed it up this way: "Not a single topic, but the general issue of being precise, being explicit, being ordered, being thorough."

The most frequent phrase in the first category, Problem Solving and Design, was just problem solving, and the phrases design and abstraction were explicitly related to object oriented design concepts. However, in the very same group of respondents, 68% reported using no tools or techniques for teaching design (Dale, 2005). However, it is difficult to know what the interpretations of the word design was among the respondents, since 78% considered it very important to teach teach *problem solving and design explicitly* and that it should be demonstrated whenever possible. One explanation might be a conflation with algorithm development, because regardless of the design methodology used, this was clearly a focus for most of these CS1 instructors: 80.3% described it as a thread spread throughout multiple lectures. In the category object oriented constructs, it is difficult to spot any obvious common difficulties, other than the rather general issues of "class", "object", "polymorphism", "inheritance" and so on. There is a need for a more qualitative research into these matters.

Thompson (2008) performed a phenomenographic study on practitioners' ways of describing the design characteristics of an object oriented program, or how a program is implemented. The interviewees expressed design characteristics of an object oriented program, and the results were categorised in five categories: *is language dependent* (e.g. the use of specific language features makes the program object oriented), *uses paradigm constructs* (e.g. objects, object identity, state, behaviour, interface, message passing), *uses generic constructs* (e.g. abstraction, composition, delegation, encapsulation), *applies generic design objectives* (e.g. cohesion, coupling, maintainability, robustness,) and *expression of thought process* (e.g. emphasis on the thought processes and the ways of thinking that are behind the programming paradigm). The higher level categories do not ignore technology aspects but see them as taking a subordinate role. Only the two highest levels concerns abstractions, while the lower ones models real world objects. The critical aspects found were then used as a guide to the analysis of a number of textbooks introducing object oriented programming. The objective was to determine whether textbooks utilise similar aspects to those found in the empirical work with practitioners. The conclusion is that most textbooks do a better job in discussing the nature of an object oriented program, what is called the "what" aspect, then they do in addressing the "how" aspect.

Educators that include object oriented topics in their introductory course see less learning difficulties regarding abstract concepts, than those who excludes object oriented topics. Schulte and Bennedsen (2006) showed that the presence of object oriented elements in a course, forces a more conceptual approach than elements of an imperative nature do. This makes it important to discuss teaching object orientation from an abstract and conceptual perspective. Teaching the paradigm must be emphasised, but is often seen as competing with elementary programming

constructs. McConnell and Burhans (2002) examined how the coverage of basic concepts in programming textbooks has changed. They noted a shift in the amount of coverage of various topics with each new programming paradigm, and with object orientation a decrease in subprograms, but also a decrease in basic programming constructs.

Another area of research is how to design courses. Different approaches have been suggested, and this is summarised by the initiative taken by IEEE and ACM to describe the body of knowledge in computer science and to formulate a common curriculum for computer science education, CC2001 (ACM, 2001, 2008b). In CC2001 three different ways to teach introductory programming are suggested: imperative-first, objects-first and functional-first.

The subject of teaching approaches, with a survey of relevant research, is thoroughly treated by Bennedsen (2008). The result of this work is the suggestion for a model-based line of presentation, based on four principles: *object from day one*, *a balanced view of the three perspectives on the role of a programming language*, *a systematic way to implement a solution*, and *explicit focus on the programming process*. To achieve this, they show many examples, they explicitly use UML and they emphasise the programming process. Role-play is used to illustrate the concepts, using everyday life situations. UML is used to describe concepts and their properties, and supplies a vocabulary for communicating classes. The model-based approach uses a read-modify-write approach to introduce a certain concept.

Some more detailed guidelines for the design of introductory courses can also be found in literature. Kölling and Rosenberg (2001) suggest eight guidelines: 1: *Objects first*, 2: *Don't start with a blank screen*, 3: *Read code*, 4: *Use "large" projects*, 5: *Don't start with "main"*, 6: *Don't use "Hello world"*, 7: *Show program structure*, and 8: *Be careful with the user interface*. In close relation to these guidelines, and based on our experiences, we developed eleven principles for course development: *No magic* (P1), *Objects from the very beginning* (P2), *General concepts favoured over language specific realisations* (P3), *No exceptions to general rules* (P4), *OOA&D early* (P5), *Exemplary examples* (P6), *Easy-to-use tools* (P7), *Hands-on* (P8), *Less "from scratch" development* (P9), *Alternative forms of examination* (P10), and *Emphasise the limitations of computers* (P11). See Paper I for a thorough description, and a discussion on the outcomes of these principles.

Advice for the introduction of object oriented design and analysis is harder to find. The design of small classes is discussed by Fowler (2003). He argues to make a type (class) when objects with some special behaviour in their operations that a primitive type does not have, are needed. The favorite example is money. Money does not behave as floats, and should not be represented by floats. The advice given is *when in doubt, make a new type*.

Providing means to break down the design of methods into smaller steps, Caspersen and Kölling (2006) recommend *The Mañana Principle*. This is to introduce novices to the idea of separating concerns and to use many small methods. *When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later.*

One way to introduce object oriented analysis and design in introductory programming, is the use of CRC-cards (Bellin and Simone, 1997; Biddle et al., 2002; Börstler et al., 2002; Gray et al., 2002; Börstler, 2005). This is an informal way to investigate design ideas, and to try out solutions, but more important, it can

provide an experience of the object oriented way of designing without having to be skilled, or even experienced in programming. Informal roleplay supported by a method for documentation, has the potential of supplying opportunities for the first steps into object thinking (Börstler, 2005; West, 2004).

A very specific concern of teaching object orientation, has been the discussion on whether to teach objects first or not. This has on several occasions been the topic on the SIGCSE mailing list (Bruce, 2004; Lister et al., 2006; Kölling, 2006; Bennedsen and Schulte, 2007). Some argue that teaching objects early has failed, while others pose the question whether this depends on the approach or the teachers. Kölling (2006) argues that teaching object orientation is complex due to two factors; intrinsic complexity, inherent in the paradigm, and accidental complexity, caused by external factors such as inadequate languages, tools, teaching strategies, teachers lack of experience with the paradigm etc. This argumentation is supported by the cognitive load theory (Paas et al., 2003). According to this theory, instructional procedures contribute to the cognitive load of the learner, in both positive and negative ways. Increases in effort or motivation can increase the cognitive resources devoted to a task which is positive for the acquisition of schemas, but badly designed instructional procedures impose a negative cognitive load on the learner. That might be practical details such as having to search for information in several places, or using inconsistent vocabulary in different instructional resources.

In an analysis of one of the SIGCSE discussions, it is shown that many of the arguments used are based on myths, rather than scientific evidence (Lister et al., 2006). This study claims that "many computer science academics lead double lives, leading their research lives and their teaching lives according to different mindsets". When it comes to research we demand strong evidence, preferably quantitative, but in our teaching profession we are reluctant to build on the works of others. This means developing materials, tools and teaching approaches individually, often based on intelligent guesses about what "works".

From identified characteristics of tasks that lead to poor learning behaviour, Carbone et al. (2000, 2001) make recommendations for how to address the identified problems. To improve tasks given to students, it is recommended to supply the students with "a method of attack", because one of the major reasons for getting stuck, was that the students did not know how to design a solution in manageable components. This also affects the possibilities for bug fixing and the handling of logical errors, often originating from compositional mistakes. A more explicit and thorough introduction to, and use of, object oriented analysis and design would empower novices in approaching problems, as well as in working with their solutions subsequently. It would furthermore draw focus away from coding which tends to dominate in CS1.

Even though student-centric approaches are often discussed in research, educators do not always teach in that way. Results show that teacher attitudes to teaching often put a focus on content and organisation (Pears et al., 2007). As educators, we need to ask ourselves what perspective we should have for our teaching. We have to decide what "success" in an introductory programming course should mean (Lister et al., 2007). If we are aiming for "development in student thinking", then we need to find ways to convey the idea of object orientation as a problem solving approach, as well as the practicalities of implementing the solution.

Focus on the process of program development and the associated

> *strategies, principles, patterns, and techniques is the missing link that we must provide in order to accomplish our mission of educating novices in the skills of programming.*(Caspersen, 2007, pp. 165)

3.2 Learning object orientation

In a survey of research modeling the cognitive aspects of learning to program, Robins et al. (2003) reports on different approaches: programming plans, schemes, programming strategies, and the difference between experts and novices.

The idea that object orientation is about active objects that are able to perform computations and manipulate data that constitute their state, and that communicate with each other, leads to a problem solving approach that focuses on the problem, not the solution (Rosson and Alpert, 1990). This has been taken as a sign of naturalness; that object oriented analysis and design would be closer to the way we think. This seems to be supported by research, at least when studying expert programmers, but it has also been shown that expert object oriented programmers use both object oriented and procedural views of the programming domain when solving problems (Détienne, 1997). Novice programmers do not exhibit the same understanding of the problem domain(Wiedenbeck et al., 1999).

But identifying entities of the problem domain does not necessarily result in the design of autonomous, active objects, since many entities in the real world are passive, dead things that hold some static information. To be able to design suitable objects must therefore be considered an important part of learning object orientation.

One way of improving programming education is to investigate what students do "wrong". Research has shown that students learning how to program with an object oriented approach have difficulties "putting the pieces together"(Spohrer and Soloway, 1986; Lahtinen et al., 2005). Spohrer and Soloway (1986) use the term *construct-based* to illustrate the view of programming that many courses and textbooks convey. Their results show that the majority of bugs in syntactically correct programs are related to plan composition, rather than misconceptions of syntactical elements.

Other results indicate that data flow (transformations which occur to variables as the program executes), and function knowledge (goals that the program accomplishes), are difficult to understand because of the delocalised nature of object orientation (Ramalingam and Wiedenbeck, 1997).

Research shows that one of the major problems for novices is to design a solution for a problem and to express the solution in program code (Rist, 1995; Robins et al., 2003; Lahtinen et al., 2005). This must be regarded highly relevant for object orientation, a paradigm that should foster designs and implementations with a delocalised nature.

There are several investigations on common misconceptions among novices. Among these are object/variable conflation due to single attribute classes leading the novice to view objects as mere wrappers for variables (Holland et al., 1997). Another difficulty is the notion of object state, to understand how the invocation of methods influences the object state (Ragonis and Ben-Ari, 2005).

From psychological research it is known that misconceptions can originate from a lack of control. If unable to make sense of a situation, or an experience, individu-

als identify a coherent and meaningful interrelationship among a set of random or unrelated stimuli to regain control (Whitson and Galinsky, 2008). To internalise new knowledge, we tend to look for patterns and/or relationships among the concepts we already know and the new concept. Novices tend to extrapolate known phenomena, sometimes leading to erroneous conclusions. One example of this is that numbers or numeric constants are the only appropriate arguments to pass for a corresponding integer parameter. Passing explicit values is easier to comprehend, than passing the value of a parameter that does not seem to have any value (Fleury, 2000). This tendency could unintentionally be due to the simplified types of examples we often encounter. The focus might be to show the principle of parameter passing in a method call, and the ambition is to make it obvious what is passed, so explicit values are used. Then when the novice needs to call a method, the use of explicit values seem to be the way to make sure that the right value is passed.

The lack of student maturity identified by educators, is alarming. In a large scale study, problem solving was identified as the most difficult topic to teach (Dale, 2005). Students seem unprepared for their first programming course. *Not a single topic, but the general issue of being precise, being explicit, being ordered, being thorough.* This research is supplemented by the research on student strategies for programming by Eckerdal et al. (2005); Eckerdal (2009). Their results show 35 different strategies grouped into four super-categories: *getting help from elsewhere*, *learning through practising*, *resolving a problematic situation on a more abstract, general level*, and *working their way around*. This further stresses the need to give novices strategies to approach both the design of solutions, as well as to resolve problems that occur during implementation and code development.

3.3 Summary

Computer science education research is maturing, and it is certainly acknowledged that the introduction to problem solving and programming is an educational challenge. Research focusing on CS1/CS2 is building up, and we think that many indications can be found proving that object oriented analysis and design is to a large extent lacking in CS1. Teaching and learning object orientation means more than teaching and learning the syntax of a language with object oriented features. It is also true that the educators are an important part of the outcomes of an introduction to object orientation, no matter what the approach is. The impact of vocabulary, the quality of examples and the support for problem solving in the object oriented paradigm must be discussed, along with the formulation of characteristics and the establishment of proper presentations of different aspects of object orientation.

None of the above mentioned studies has made any attempts to analyse or question the object oriented quality of the examples or exercises used. Neither has the educators' personal views on object orientation been investigated. The perception of object orientation is varying among professionals, and it seems reasonable to assume that this is true for educators as well. It is not unlikely that some of the observed problems and difficulties are due to the way object orientation is presented, in terms of adherence to object oriented characteristics and principles. It is vital to decide on *what* to teach, before analysing *how* to best teach it.

Chapter 4

Object Oriented Quality

If we are to discuss the object oriented quality of examples, we must start with the basic properties of object orientation. However, there is no canonical definition of object oriented problem solving and programming. Because of this lack of commonly agreed upon characteristics, we explore how object orientation is portrayed in literature, and in software developing practices. The basis for the present work has been an investigation, and examination, of concepts and design guidelines commonly promoted and used in the field.

4.1 Core Concepts and Design Principles

In the early years of object orientation, Nygaard and Dahl used ideas from ALGOL to name entities objects and to establish characteristics for a new language, Simula (Nygaard, 1986). They stated that the basic concept should be *classes of objects* and that the *subclass concept*, should be a part of the language, Simula 67. The term object-oriented programming is derived from the object concept in this language. Stroustrup (1995) stated that *The fundamental idea is simply to improve design and programming through abstraction.* The concept of objects in Simula 67 was the basis for the term object oriented programming, coined by Alan Kay, the designer of Smalltalk, coined the term object oriented programming, and considered it */../a new design paradigm /.../ for attacking large problems of the professional programmer, and making small ones possible for the novice user* (Kay, 1996).

Among other sources that discusses characteristics of object orientation, are ACM's basic requirements for a computer science degree (ACM, 2001),(ACM, 2008a). Based on cognitive research, the notion of critical concepts in object oriented programming has been developed by (Mead et al., 2006; Meyer, 2006). Further indications of the characteristics of object orientation can be found in studies of frequent object oriented concepts in literature (Henderson-Sellers and Edwards, 1994; Armstrong, 2006).

To find out what concepts that could be considered central to the paradigm, we investigated the literature which resulted in a set of concepts characterising object orientation: *Abstraction*, *Responsibility*, *Encapsulation*, *Information hiding*, *Inheritance*, *Polymorphism*, *Protocol/Interface*, *Communication*, *Class* and *Object*.

Abstraction is often mentioned as a central concept. Abstractions are said to

be the driving force of object oriented design and the key to defining the objects, as the building blocks of an object oriented solution to a given problem (Devlin, 2003; Kramer, 2007; Meyer, 2001; Parnas, 2007).

But concepts in themselves are not enough to provide an good understanding of object orientation. We also need to know how to use the features captured by the concepts, to develop well designed software. The practices proposed and used by the software developing community reveal what is considered good object oriented design, and they would therefore provide insights into the characteristics of object orientation.

Design-advice is given by different researchers/practitioners, and they come in many different forms; heuristics (Johnson and Foote, 1988; Riel, 1996; Gibbon and Higgins, 1996; Grotehen, 2001; West, 2004), rules and guidelines (Bloch, 2001; Wick et al., 2004; Garzás and Piattini, 2007), code smells and refactorings (Opdyke, 1992; Fowler et al., 1999; Mäntylä, 2003; Mäntylä), patterns (Gamma et al., 1995), object oriented software metrics (Chidamber and Kemerer, 1991; Purao and Vaishnavi, 2003; Lanza et al., 2005) etc. Some are general principles and some are very detailed do's and don'ts.

Searching the literature we have found that most object oriented design "advice" are captured by the principles collected and/or formulated by Martin (2003). They are grouped into three categories: Class design, Package cohesion and Package coupling. When teaching novices, the major focus is on class design, packages are rarely introduced in an introductory course. The class design principles are:

SRP – The Single Responsibility Principle Each responsibility should be modelled by a separate class. A class should have one, and only one, reason to change.

OCP – The Open Closed Principle A module should be open for extension but closed for modification.

LSP – The Liskov Substitution Principle Subclasses should be substitutable for their base classes.

DIP – The Dependency Inversion Principle Depend upon abstractions. Do not depend upon concretions.

ISP – The Interface Segregation Principle Many client specific interfaces are better than one general-purpose interface.

Since these principles are dedicated to class design, there is no principle focusing on the collaboration among classes. Collaboration is an important component of object orientation and to incorporate this aspect we found it necessary to include two more principles:

LoD – Law of Demeter Do not talk to strangers. Only talk to your immediate friends.

Favour object composition over class inheritance. One of the basic ideas of the Gang-of-Four design patterns.

We believe that these principles describe what can be considered the general idea of object orientation, at least the parts that can be related to the introduction of object orientation to novices.

The set of concepts and the principles are related, and complement each other, Table 4.1.

Table 4.1: The relationships among object oriented Principles and Concepts.

Principle	Concept: Abstraction	Responsibility	Encapsulation	Information hiding	Inheritance	Polymorphism	Protocol/Interface	Communication	Class	Object
SRP – The Single Responsibility Principle	X	X	X				X		X	
OCP – The Open Closed Principle	X	X	X	X?						
LSP – The Liskov Substitution Principle					X	X				
DIP – The Dependency Inversion Principle					X	X				
ISP – The Interface Segregation Principle					X					
LoD – Law of Demeter								X		
Program to an interface, not an implementation			X	X	X	X		X		
Favour object composition over class inheritance	X	X								

For a thorough discussion on this research, see (Nordström, 2009) or Paper II.

4.2 The Object Oriented Quality of Examples

With the characteristics of object orientation as starting point, it becomes possible to discuss object oriented quality of examples for novices. To investigate the possibility to measure the quality of examples for novices, we developed an evaluation tool (Börstler et al., 2008a). The instrument was piloted by six experienced educators on five examples (Börstler et al., 2008a,b). Based on this pilot study, the instrument was redesigned, and in the resulting tool, the factors evaluating the object oriented quality of an example are:

Reasonable Abstractions (O1): Abstractions are plausible from an object oriented modelling perspective as well as from a the perspective of a novice.

Reasonable State and Behaviour (O2): State and behaviour make sense in the presented software world context.

Reasonable Class Relationships (O3): Class relationships are modelled properly (the "right" class relationships are applied for the "right" reasons).

Exemplary OO code (O4): The example is free of "code smells".

Promotes "Object Thinking" (O5): The example supports the notion of an object oriented program as a collection of collaborating objects.

For each quality factor in the evaluation tool a list of typical problems is provided to exemplify the quality addressed. This list is distilled from the literature on student problems or misconceptions, and common violations of the acknowledged general principles of object orientation.

With this tool, a large-scale study was performed (Börstler et al., 2009). The examples for this study were chosen to be representative of a wide range of examples from introductory programming textbooks. Each example was to be the first one in a text, exemplifying certain high level concepts or ideas. Three major categories

were used: *First user defined class* (FUDC), *Multiple user defined classes* (OOD) and *Control structures* (CS). The aim was to have a broad and representative coverage of textbooks, with respect to popularity, coverage, presentation style and pedagogic approach. In this study we received in total 215 valid reviews by 25 reviewers. An extended analysis was made on the 21 examples that received $\geq$3 reviews each (191 reviews in total) (Börstler et al., 2010).

One interesting results of the evaluation of the quality of object oriented examples, is that the general impression of an example tends to degrade after the evaluation of the example. The fact that there are specific items to evaluate, seems to draw attention to details not initially spotted, and thereby develop the way we view certain features, or the lack thereof, of the example.

In general, the results show that the object oriented quality of examples is low. One reason for this might be that the focus in many textbooks is more on basic programming constructs and syntax-related matters, than conveying the mindset of the paradigm. Dealing with both introduction to programming as well as introduction to object orientation, sets high demands on the design of examples.

Quality is a problematic thing to discuss. On one hand, it might be individual what is regarded as "good" quality. On the other hand, it is often easy to agree on really "bad" quality. Programming is often highly individual in terms of method and style. Most programmers have firm opinions on the topic, and often extended arguments to support their beliefs. This is reflected in introductory programming text-books, and again, it seems to be a question of belief, both of what object orientation means and how to best teach it. Criteria for object oriented quality is lacking.

We have suggested and tested criteria for the evaluation of the object oriented quality of examples for novices. The agreement among reviewers regarding the different quality factors for a large number of examples, shows that it is possible to use such a tool.

Chapter 5

Designing Object Oriented Examples

The importance of good examples is well known. However, it is surprisingly difficult to find examples that are truly object oriented, in the sense that they are faithful to general guidelines and principles formulated to support for object oriented design. One aim of this work has been to investigate the issue of object oriented quality in examples for novices. The goal is to be able to support the design of examples, with focus on the object oriented quality.

The lack of examples can partly be explained by the somewhat different conditions for the actual development of software using the object oriented paradigm on one hand, and teaching/learning object orientation on the other. Object orientation is a problem solving approach designed to handle complex problems. For software development, there are high demands on maintenance, efficiency and reusability. But in the educational setting, there are some more concerns to take into consideration. Novices have a very limited frame of reference in terms of concepts and language elements, and often have no experience of programming in general. This means that the mere idea of execution of statements could be problematic. Because of this, it is necessary to simplify and reduce the examples to make it possible for novices to see the essence of the feature or concept exemplified. When the examples are small, it becomes difficult to illustrate some of the general features of the object oriented approach. Nevertheless, it is particularly important in examples for novices, to emphasise exemplary objects and promote object thinking.

Computer science education research has shown that much of the lack of success in the outcomes of introductory programming courses, lies within the problem-solving abilities. It is therefore necessary to provide novices with a conceptual model that has the ability to sustain throughout the progression to more advanced elements. This can be achieved by thinking of examples as illustrating the process of software development, and objects as being autonomous, collaborative service providers to clients, probably other objects. As part of a problem solving approach, the context supports in making the example plausible as showing at least a fairly realistic need for software development.

To support the design of introductory object oriented examples, we propose 5 Eduristics as described in the following subsections. They are based on a thorough

review of the literature on various aspects affecting object oriented quality as summarized in Figure 5.1. For a detailed description see (Nordström, 2009) or Paper II.

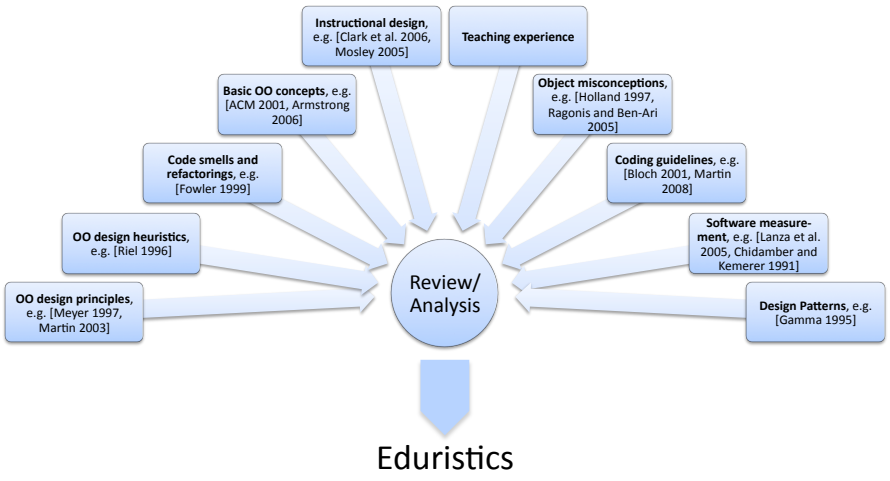

Figure 5.1: Investigating the characteristics of object orientation

5.1 *Eduristics* for the Design of Object Oriented Examples

The Eduristics aim at supporting a clear focus on abstractions, and objects as autonomous, encapsulated, collaborating objects supplying services to clients. Please note that the Eduristics, as described below, have been slightly revised compared to the first version published, to make them more focused and consistent.

1. *Model Reasonable Abstractions*

Since abstractions play a major role in object orientation, this is maybe the heuristics that is most basic, and to some extent would be sufficient on its own. A reasonable abstraction means that there must be a problem presented that, to a novice, is likely to appear in software. It is often the case that we have to make simplifications for a small-scale problem, to make the problem appropriate in size and complexity. However, it is necessary to strive for non-artificial classes and objects. It must be possible to imagine a client using objects of this kind, and the objects must model some entity in the problem domain. Encapsulation and information hiding must be emphasized.

- Abstractions must be meaningful from a software perspective, but also plausible from a novice's point of view.
- Do not put the entire application into `main`, and isolate it from other application classes.
- No God classes.

2. *Model Reasonable Behaviour*

One risk in the small-scale situation, is to oversimplify the behaviour of objects. We have to design examples with objects simple enough, in terms of syntactical elements and programming concepts, for a novice to understand with her/his limited "vocabulary". Nonetheless is it crucial to avoid trivial or artificial behaviour, because it may distract the novice from understanding the basic concept of behaviour in an object. Modifying the attributes with set- and get-methods is not an example of behaviour, but rather of external manipulation of the object. Preferably the behaviour is separated from the internal representation of the state of the object, and clearly connected to the problem domain. Artificial behaviour is often the result of choosing real world objects, without a context supporting that the abstraction is justified to solve a certain programming problem. A car has the ability to for example move forward, move backwards, start, stop and turn left and right, but in isolation these abilities are artificial. It is difficult to discuss what start means in terms of behaviour.

- Show objects changing state and behaviour depending on state.
- Do not confuse the model with the modelled.
- No classes with just setters/getters ("containers").
- No code snippets.
- No printing for tracing, use `toString` to communicate any textual representation.

3. *Emphasize Client View*

When we design small-scale examples, we have to think about how to support the novice in object-thinking. Taking a clients' view when designing a class gives important indications for designing classes. This also makes it necessary to supply a context, to be able to think about clients. It is also crucial to define the responsibilities and services of an object separately from the internal representation and implementation of the responsibilities. Leaving the implementational details out from the design thinking will promote object thinking and make problem solving easier for the the novice. Discussing what a client would/should expect in terms of consistency and logic will most likely extend an example, but will empower the novice in terms of analysis and design.

- Promote thinking in terms of services that are required from explicit clients.
- Separate the internal representation from the external functionality.

4. *Promote Composition*

Extensive use of primitive types, or `String`, for attributes, makes the idea of collaborating objects hard to illustrate. Often, it creates problems to use simple types to represent a complex responsibility. Using, for example a `String`-object to represent a date, or an `int` to be responsible for the age of a person, does not emphasise that objects have the responsibility to provide services, instead they become closer to being containers for values. Furthermore, making an effort to

design examples where objects collaborate, and delegate responsibilities to other objects is beneficiary for the novice to acquire a proper conception of object oriented problem solving.

Inheritance is an important feature of the paradigm, but it is considered difficult to learn and is therefore given a lot of attention. Since novices have a very limited repertoire of concepts and syntactical constructs, it is difficult to show the nature and strength of inheritance. Inheritance is often used to exemplify reuse, but this feature of object orientation can also be demonstrated by composition. To show the strength and usefulness of inheritance, behaviour must guide the design of hierarchies and specialisation must be clear and restricted. Subclasses should be structurally related, but separated by behaviour. An example of this could be balls in a small game-context. Maybe green balls explode, while red balls multiply when hit. This could be a case for inheritance. Instead of having one class `Ball`, that has to check the state (colour) and decide upon different actions due to the state, this could be implemented through inheritance. For a proper use of inheritance, it is important to respect the *The Liskov Substitution Principle*. This principle promotes polymorphism, but restricts the relationship between the base class and the derived class. What can be expected of an object of the base class, must always be true for objects of the derived class.

- Emphasize the idea of collaborating objects, use object for attributes to demonstrate the distribution of responsibilities.
- Do not use inheritance to model roles.
- Inheritance should separate behaviour and demonstrate polymorphism.

5. *Use Exemplary Objects Only*

We have found it common that practicalities of examples threaten to violate the basic characteristics of objects. Even if the abstraction is well chosen, based on the clients view and the context makes the proposed design plausible, we may unintentionally contradict the general intention of an example. To show the idea of objects, we should strive for "many" objects present in the small-scale example. It is also important to be explicit, i.e. using explicit objects whenever possible. Static attributes and static methods can confuse novices. Including the `main`-method in an abstraction means breaking the concept of abstraction and encapsulation. Having a `main`-method that creates an object of the class itself is confusing. It must be confusing, that something that does not exist can create an instance of itself. The method `main` is an exception to object orientation for many reasons. The invocation is done without any object being instantiated (static), it is not called by any explicit object, the invocation is not visible in code (system defined invocation) and it is not the implementation of any behaviour that the class is responsible for.

- Promote "object thinking", i.e. objects are autonomous entities with clearly defined responsibilities.
- Instantiate multiple objects of at least one class.
- Do not model "one-of-a-kind" objects.

- Make all objects/classes explicit, e.g. no anonymous classes and explain where objects that are not instantiated explicitly come from.
- Make all relationships explicit: avoid message chains. Objects should only communicate with objects they know explicitly (Law of Demeter (Lieberherr and Holland, 1989)).
- Avoid shortcuts.

5.2 Evaluating the Object Oriented Quality of Examples

Even if we were to agree precisely on what the characteristics of object orientation are, it is still a matter of personal style and preferences how we implement them. In collaboration with other experienced educators, we set out to investigate whether the quality of object oriented examples could be defined and subsequently measured in some way. Thorough discussions, based on teaching experience, the view of object orientation, and the conditions for teaching, resulted in a first suggestion for evaluation criteria, formulated as quality factors in three categories (Paper II, Börstler et al. 2008b). Based on the results of this pilot-study, and further literature-studies (Nordström, 2009), the definition of example qualities was further developed. Besides the object oriented qualities described in Section 4.2, the checklist/tool did also consider technical and didactical quality factors. Two technical qualities were considered. T1: *Correctness and Completeness*, this means that the code is bug free and the example is sufficiently complete. The second technical quality was T2: *Readability and Style*, and concerns the readability of the code in terms of consistent formatting and style. The didactical qualities were evaluated by three quality factors. D1: *Sense of Purpose*, students must be able to relate to the example's domain and computer programming must seem a reasonable way to solve the problem. D2: *Process*, an appropriate programming process is followed/described. D3: *Well Balanced Cognitive Load*, explanations and supporting materials should promote comprehension; they are neither simplistic, nor do they impose extraneous cognitive load. The object oriented quality factors are described in Section 4.2.

To investigate the object oriented qualities of common textbook examples, a large-scale evaluation of textbook examples was conducted in an ITiCSE working group (Paper III, Paper IV and Börstler et al. 2009). In this study textbooks were classified object oriented and traditional. The classification was based on a careful evaluation of the texts' sequence of presentation and focus and style of presentation. Texts in category *OO* had a clear and early focus on object orientation, and texts in category *Trad* had a more traditional imperative first approach. Examples with comparable properties were chosen, and of special interest was the first example exemplifying a certain high level concept or idea. Three groups of examples were evaluated. The *First user defined class* (FUDC), are examples that reflect the first occurrence of a user defined class in a text. The second group was examples showing *Multiple user defined classes* (OOD). Examples in this group exemplify some kind of design decision/strategy involving several classes. They show how existing classes can be "used" for defining new classes (inheritance, composition) or how designs

can be made flexible (interfaces, polymorphism, ...). Examples in this group can be considered role models for determining relationships between classes. Finally we evaluated examples introducing *Control structures* (CS). The introduction of control structures might be considered contradictory to the purpose of introducing of object orientation. Nonetheless, even in object oriented programs there are elements of imperative flow of control, and novices must have some knowledge of the general elements of programming to be able to read, use and maintain available code. We consider these three categories of examples particularly important, since they "set the stage" for how students are expected to think about object oriented class design.

According to the results of this evaluation, based on 191 data points by 24 reviewers, the evaluation instrument shows high inter-rater agreement, and therefore must be considered as quite reliable (for details, see Paper IV).

In Table 5.1 the relationships between the quality factors and the heuristics are shown.

Table 5.1: The relationships among object oriented Quality factors in the evaluation tool and the educational heuristics.

	Eduristic				
Object Oriented Quality	**1. Model Reasonable Abstractions**	**2. Model Reasonable Behaviour**	**3. Emphasize Client View**	**4. Promote Composition**	**5. Use Exemplary Objects only**
O1—Reasonable Abstractions	X				X
O2—Reasonable State and Behaviour		X	X		X
O3—Reasonable Class Relationships				X	
O4—Exemplary OO	X	X			X
O5—Promotes "Object Thinking"	X	X	X	X	X

Examples clearly appreciated by the reviewers in the study described in Paper III, Paper IV and Börstler et al. (2009)) are also in accordance with the heuristics. Examples score high when the issues of the heuristics are upheld, and low when the heuristics are violated. We argue that this confirms that designing examples according to the Eduristics described in Section 5.1 supports object oriented quality in examples for novices. However, this has to be further (Chapter 7).

Chapter 6

Listening to Educators

The research in the teaching and learning of programming focuses mainly on students' learning, for example misconceptions, patterns of behaviour in compiling and correcting errors. However, little is known of the educators and their reasons and choices for approach when teaching object orientation. They are the users and designers of examples, so it is of importance how they personally view the paradigm, and how they present it to novices. Very little is known about the difficulties educators' experience in teaching object orientation. There are, to our knowledge, no studies on the educators' perspectives on object orientation. In this chapter we describe an exploratory study. Its aim is to identify ways educators think about, and deal with issues of object orientation.

We decided to use a qualitative approach with semi-structured interviews to be able to listen to educators talking about the teaching of object orientation in their own words. The main strength of qualitative research is its ability to study phenomena which are unknown, and otherwise unavailable (Silverman, 2006). Surveys and questionnaires with closed-end questions do not reveal the respondents personal preferences for wording, and the reason for giving a certain answer can not be elaborated on. Even open-ended survey-questions are limited in terms of the ability to pursue a certain line of questioning. Since there was no theory available and no way of knowing what kind of information we could expect from educators, we found it difficult to define questions and phrase suitable and answers, to make questionnaires useful.

6.1 Educators' Personal Views on Object Orientation

The lack of previous work in educators' views on object orientation makes this study exploratory and unique. We wanted to listen to the educators and try to identify their basic understanding of the paradigm. It was also interesting to learn something about the conditions of their work in teaching object orientation to novices. This would make it possible to discuss the quality of object orientation, as implemented in teaching.

This particular research is thematized according to four themes; the paradigm itself, the concept of an object, examples, and the problem solving process (Object

Oriented Analysis and Design, OOA&D). Each theme is viewed from three different aspects, the educators personal view, the educators view of student difficulties and the educators choice of methodology to address the specific issue.

The reason for choosing this structure was an attempt to separate the different components concerning the design of examples. The way examples are chosen or designed is probably affected by the preferences of the educator. Therefore it was interesting to explore how the respondents were describing their personal view of object orientation, as a starting point. We were also curious to collect information on what the educators had perceived as difficult for the students. These two aspects should be important for the educators' choice of strategy or method to teach the themes respectively. At the same time their choice of teaching approach should contribute to the understanding of their personal views.

The structuring of the area, used as interview guide, is illustrated by the matrix in Figure 6.1.

	Teacher's personal view on concept	*Teacher's view of students difficulties*	*Teacher's choice of methodology*
	Characteristical	**Problematic**	**Teaching-practice**
Paradigm (OO)	What are the characteristics of OO? What is most important to stress?	What about OO is most difficult to internalise?	How is OO presented, as paradigm?
Concept (Object)	Ideal objects, how are they defined?	What is perceived as difficult about objects?	How does a displayed object typically look?
Examples	What is characterstic of a good example?	What makes an example difficult for students?	How are examples chosen and/or designed? What characteristics are prioritised?
Process (OOA&D)	What is characteristic for the problem-solving approach?	What do students find difficult in OOA&D?	How is OOA&D introduced and practised?

Figure 6.1: Interview guide

This interview guide was not shown to the interviewees, and the questions are only illustrations to the interviewer of what to listen for. The primary use of the guide was to secure that all interviews touched upon the same aspects. This was accomplished by gentle prompting by the interviewer, if necessary.

6.2 Respondents

In all 10 interviews have been conducted, 6 with teachers from upper secondary school (teaching students at the age 16-19) and 4 with educators at the university

level. The reason for having participants from two levels of education is that educators at the university would most likely be the teachers of teachers working in upper secondary school, and therefore highly influential on the teaching in upper secondary school. See Appendix A for a description of the Swedish school system.

In this study, the sampling of interviewees is convenience-based, and not based on any statistical grounds. Teachers from upper secondary schools have a busy schedule, and in retrospective we are thankful to and appreciate the teachers who were interested in devoting some of their time to be interviewed.

When sample size for qualitative studies is discussed, it is often stated that the quality of information obtained per unit is the most critical measure. Sample size is difficult to determine and a general recommendation is to proceed until analytical saturation is received. Another recommendation for this semi-structured interviews is to include about six to ten participants (Sandelowski, 1995; Morse, 2000).

The demographics of the respondents are shown in Figure 6.2.

ID	Degree	OO	School	Size	Exp
R1	T	Self	USS	M	18
R2	T*	Academic	USS	S	2
R3	Bach CS+T	Academic	USS	S	11
R4	T	Academic	USS	L	11
R5	T	Academic	USS	L	13
R6	T	Self	USS	M	13
R7	PhD IS	Academic	U	M	11
R8	Master CS	Academic	U	S	11
R9	Bach. IS	Academic	U	S	16
R10	PhD CS	Academic	U	L	5

Figure 6.2: Demographics of respondents.

ID All interviewees are identified by an simple code, R1-R10.

Degree Knowing that the recruitment of CS-teachers for upper secondary school is difficult, it was interesting to collect information on the formal degree of the respondents. Degree-abbreviations: T=trained teacher, CS=Computer Science, and IS=Information Systems. T* is on his/her way to a teachers degree, but not graduated at the time of the interview.

OO Furthermore, we collected information on how the interviewees had gained their competence and skills in object oriented problem solving and programming, whether they had formal academic training or were autodidacts.

School The first six respondents work in upper secondary schools (USS), and the last four lecture at university-level (U).

Size It is always a risk that small institutions have more restrictions on their courses, e.g. having students from very different programs in the same class, which may affect the teachers working conditions. Therefore, we made an

effort to have Small (S), Medium (M) and Large (L) size schools/universities represented in the population, which was successful.

Exp The last column of Figure 6.2 shows the respondents' experience, in years, in teaching programming (*Exp*).

It was hard to find any women teaching object orientation, so we are grateful to have one woman among the respondents.

All, except one, of the upper secondary school teachers (id R1-R6) are trained teachers in math and/or physics. Another common background is to have a bachelors degree in some major subject and then to add courses for the fulfillment of a teachers degree. Interviewee R3 is typical in this sense, but at the same time non-typical, since a CS-degree is uncommon among upper secondary school teachers in Sweden. This variety in the background of teachers in upper secondary school is probably due to the fact that Computer Science is not recognised as a subject within the teacher educational system.

Even though eight out of ten respondents have received academic training in computer science, none them have any pedagogical training specifically for computer science. The academic training is completely within the traditional subject of computer science. One of the university lecturers in this study earned a PhD in Chemistry before switching to CS.

The lack of professional CS-teachers in Sweden, makes it common for schools to assign science teachers, without formal CS training, to teach programming courses. They are often autodidacts, and on many schools the only teacher in this subject.

6.3 Interviews

All interviews were conducted at a place chosen by the interviewee. The interviews were recorded using a digital voice recorder, and the length of the interviews ranges from 45 minutes to 1 hour and 16 minutes. All the interviews were conducted by the author and they were all done in Swedish.

Every interview started with the interviewer asking the interviewee to describe his/her background and how he/she came to be teaching object orientation to novices.

The transcription was done verbatim using the program Transcriva. Some of the interviews were transcribed by the author, and for the remaining interviews, the transcription was directly supervised by the author. The transcripts were all proofread by the author, and any discrepancies, unsolved obscurities or misinterpretations, corrected by the author. Finally, all quotes used (e.g. in Paper V and Paper VI) have been translated by the author.

For every interview, all 12 aspects shown in Figure 6.1 are touched upon, thus generating very rich data.

6.4 Analysis

The analysis has been done using qualitative content analysis (Hsieh and Shannon, 2005; Forman and Damschroder, 2007). Content analysis is a widely used qualitative research technique, particularly in health studies (Graneheim and Lundman,

2004; Hsieh and Shannon, 2005; Forman and Damschroder, 2007; Elo and Kyngas, 2008). Current applications of content analysis show three distinct approaches: conventional, directed, or summative. They are all used to interpret meaning from the content of text data. The major differences among the approaches are coding schemes, origins of codes, and threats to trustworthiness. In conventional content analysis, coding categories are derived directly from the text data. With a directed approach, analysis starts with a theory or relevant research findings as guidance for initial codes. A summative content analysis involves counting and comparisons, usually of keywords or content, followed by the interpretation of the underlying context (Hsieh and Shannon, 2005).

In this study the conventional approach has been used, because of the lack of previous studies on educators' views on object orientation. The primary objective is the manifest view of object orientation investigated through the different aspects presented in Figure 6.1.

Once the transcripts were done and proof read, each was transferred from a word-document to a spreadsheet-document. Reading through the text, statements were condensed/concentrated, in a separate column. To be able to return to the original record for any statement, at any time during the analysis, they were all given an identification tag [*id_ row*], where *id* is the respondents identification (see Figure 6.2), and *row* is the row number of that particular transcript, see Figure 6.3.

199	00:14:34.97		
200	Jaa, det törs jag nog s... säga. För att ....jamenn om jag säger att jag skulle ha en grundkurs i programmering där vi använder C som som språk, då skulle jag inte vara så här väldigt ängslig över att studenterna redan första dan skulle förstå hur man använder sig utav såna här s... struct... struct-strukturer	Imperativa språk mindre känsliga för exempel	[R7_200] Imperativa språk mindre känsliga för exempel
201	Marie		
202	00:15:07.87		

Figure 6.3: Condensing and providing unique identification tags.

This way the time-stamps of the statements in the transcripts provided easy access to the audio-files. This allowed for the context of a certain statement to be easily recovered, in case there were any uncertainties of how to interpret the statement.

All condensed statements, without identification tags, were collected in a single document, and then they were analysed, and each concentrated statement was labeled as contributing to one or more of the 12 aspects, and/or "other.

Statements were left unlabelled if they were addressing important, but unrelated issues, not concerning the themes and aspects of the interview guide in 6.1.

Next, 13 columns were added to the spreadsheets containing each interview respectively, the first twelve for marking any of the 12 aspects in Figure 6.1, and the last column to mark other interesting comments in the text.

This way it was possible to filter out all statements marked as contributing to the information on a particular aspect, see Figure 6.4. After filtering out all tags belonging to a certain aspect, e.g. *Educators personal view on object orientation as paradigm*, in all the interviews, the next step was to start looking for any patterns or themes among them, Figure 6.5.

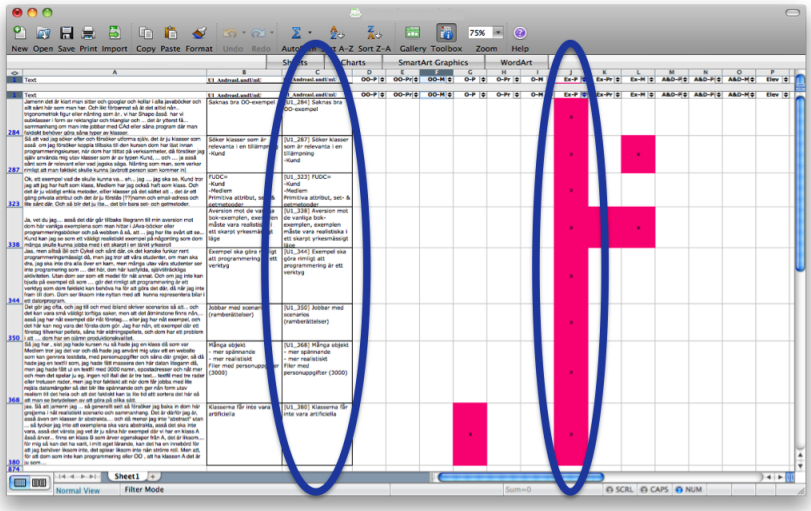

Figure 6.4: Using filter to collect statements for a certain aspect.

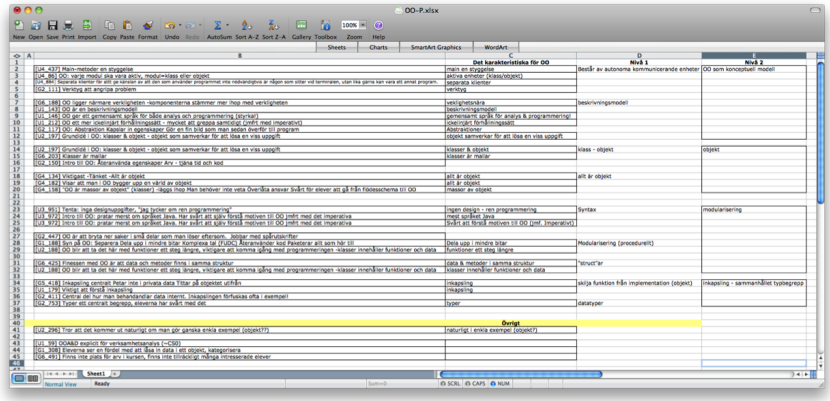

Figure 6.5: Looking for themes.

According to Forman and Damschroder (2007), the coding allows the data to be rearranged in analytically meaningful categories. The concentrated statements were organised into thematic categories, sometimes in several passes and levels, to achieve a suitable level of abstraction. Working with the statements, some appeared not to fit in with the aspect analysed, then the interview and the corresponding transcript were closely studied again, to see if the classification should be altered. This work was done for all aspects discussed in this chapter.

During the analysis both the audio files and the transcripts have been processed many times.

6.5 Results

The specific research questions investigated are:

- *How can educators' views on OO be characterised*? (Paper V)
- *How can educators' views and strategies for teaching OOA&D be characterised*? (Paper VI)

The aspects investigated and the related paper is shown in Figure 6.6.

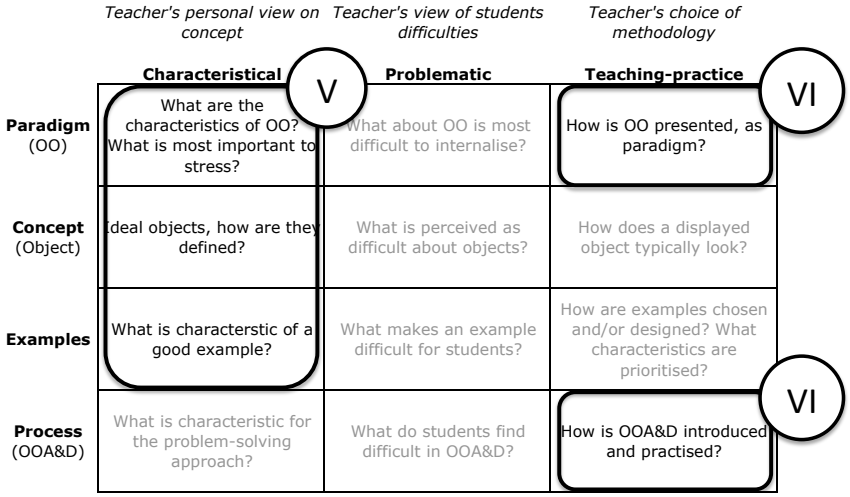

Figure 6.6: Aspects investigated.

Personal views

Based on the process described in Section 6.4, the following three aspects were analysed: *Educators' personal view on the characteristics of object orientation*, *Educators' personal view on the concept of objects*, and *Educators' personal view on examples*, see Figure 6.6.

The resulting categories are organised from a conceptual perspective, moving from abstract to simple, as shown in Figure 6.7.

With some exceptions, the interviewees consistently expressed views that were based in programming rather than conceptual, for the themes *Object orientation* and *Object*. In the case of *Example*, the majority of interviewees expressed an urge to use situated examples. Object were often chosen to model things from every-day life. Viewing examples as illustration for problem solving, or being determined by data to be handled, were exceptions.

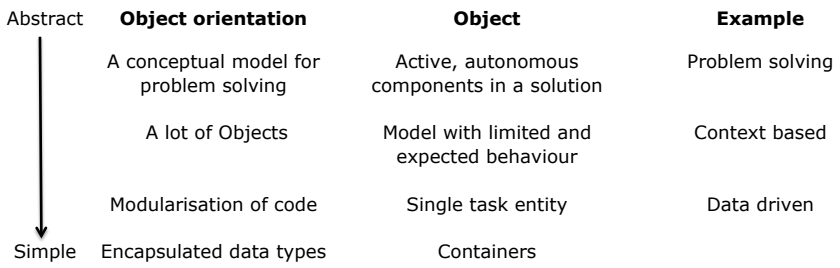

Figure 6.7: Categories of views on *Object orientation*, *Objects*, and *Examples*.

Strategies for Teaching

In this part of the study, we were looking at the aspects *Educators choice of methodology for introducing OO* and *Educators choice of methodology for teaching OOA&D*.

Introducing Object Orientation

One of the research questions was to find out how educators addressed the problem of introducing the general idea of object orientation. This is addressed by the aspect *Educators choice of methodology for introducing OO* in Figure 6.1.

Three categories of strategies for the introduction of object orientation as paradigm has been identified from the interviews. They can be characterised as: *Building a world of objects*, *Induced by contexts, Databases and concepts*, and *Not addressed*.

Introducing Object Oriented Analysis and Design

Introducing students to object oriented analysis and design can be done explicitly or implicitly, or not at all. The emerging categories for the methodological approach are: *Explicit*, *Implicit* and *Not at all*. The Explicit and Implicit categories were further divided into sub-categories, see Figure 6.8.

Explicit	Implicit
Lexical analysis	Scenariobased
Design Patterns	Metaphors
Design reasoning	Fomal notation (UML)
	Object-rich contexts

Figure 6.8: Strategies employed for OOA&D.

Only two of the educators, both university lecturers, explicitly addressed the issue of a structured approach to object oriented analysis and design. However, they discussed it mainly from an ideological point of view, and did not to any extent implement it in their teaching. The reason for not emphasising analysis and

design was partly blamed on lack of time, and that there were " more central issues to be addressed".

The majority of educators who exhibited some kind of support for object oriented analysis and design, only demonstrated the problem solving approach indirectly, through their own practices.

The practices without support for how to chose and design classes, can be termed: *Data driven*, *Objects supplied*, *Physical objects* and *Design supplied*. In these cases the students get to decide on very few, if any, classes of their own. The design is given by the lecturer, either through libraries, in some some kind of class description, or in words. In these cases the focus was entirely on getting the structure of a class right, using methods as a way to modularise the problem. Several of the interviewees, mentions that they are accepting solutions that works, rather than trying to get the novices to formulate a solution that is object oriented. Some of them expressed concerns about discouraging the novices if they would pursue the object oriented qualities of suggested solutions.

6.6 Discussion

It turned out that many of the interviewees were reluctant to discuss object orientation in abstract or conceptual terms. It was, for example, almost impossible to discuss, in any explicit way, what kind of characteristics they appreciated in examples. Most of them seemingly had preferences and things they avoided, but had never formulated a more explicit point of view. Some of the respondents mentioned lack of time, and available fora for discussing issues concerning teaching object orientation.

Research shows that object oriented analysis and design have a hard time making its way into the curricula of introductory programming courses. Novices struggle with finding a way to compose a solution out of smaller pieces of sub tasks, "putting it all together", see for example (Lahtinen et al., 2005). This is confirmed by the results of these interviews. There is very little evidence of a practical, explicit support from the educators, to aid the students in gaining such an understanding of object orientation.

In the study discussed in this chapter, the level of abstraction in teaching object oriented programming seem to be related to whether the teaching starts with general programming components of imperative nature or not. The presence of object oriented elements in a course requires a more conceptual approach, than courses that are purely imperative (Schulte and Bennedsen, 2006).

One interpretation of our data is that educators do not seem to have formulated explicit criteria for how to present and address different concepts in object orientation. Implicitly, some of the interviewees do seem to have guidelines that guide their choice of examples for the introduction of object orientation as a paradigm, but none of them had any conscious and articulated criteria for this. However, there is a correspondence to be found among the three themes in Figure 6.7. Indications of a conceptual view can be found in all aspects, and those who did express a more abstract view, also tended to use it for all three aspects.

Returning to the two questions posed:

- *How can educators' views on OO be characterised*?

- *How can educators' views and strategies for teaching OOA&D be characterised?*

There is a large variety in how these educators talk about object orientation, and their approaches to teaching it. They are all very dedicated to their task of teaching novices, show a lot of enthusiasm and take on the responsibilities of teaching with utmost care. They all put in a lot of work to assist their students to acquire skills in programming, but some express frustration over the seemingly weak results in terms of students' capabilities at the end of the course. However, their teaching is in general more focused on syntactical issues, and to solve imperative algorithmic problems, than on the understanding of the basics of object orientation. To some extent, this could be explained by the educators' weak understanding of the paradigm themselves, and lack of explicit approach to teach the fundamentals of it. Strategies for teaching object oriented analysis and design are, in general, absent. This seem to drive a more procedural approach to solving problems, where objects become containers, or records of data, in combination with methods for the access to and manipulation of attributes.

6.7 Trustworthiness

The question of trustworthiness in qualitative research is not unproblematic. In the tension between qualitative and quantitative research, qualitative methods are often accused of failing to achieve the common criteria of adequacy or rigor in scientific research; reliability, validity and objectivity.

In one of the basic texts on inquiry based research, Lincoln and Guba (1985) rephrases the issue as: *How can an inquirer persuade his or her audiences (including self) that the findings of an inquiry are worth paying attention to, worth taking account of?* They discuss four factors for the trustworthiness of qualitative research, relating to the traditional criteria of rigor in quantitative research (shown in parenthesis below).

1. Truth value (internal validity) : Credibility is used as criterion for truth value. A qualitative study is credible when it presents descriptions or interpretations such that people can recognise the experience after having only read about it in the study (Sandelowski, 1986).

2. Applicability (external validity) : Qualitative researchers argue that *every research situation is restricted to a particular researcher in interaction with a particular subject in a particular context*, and that generalisability is an illusion (Sandelowski, 1986). Sample sizes are typically small and subjects are often selected because they can illuminate the phenomenon studied. This makes the question of generalising less appropriate. Lincoln and Guba suggests the criterion fittingness, which means that the findings can "fit" into contexts outside the study situation and that the audience regards its results as meaningful and applicable in terms of their own experiences (Sandelowski, 1986).

3. Consistency (reliability) : Reliability is considered a necessary precondition for validity in quantitative research. For this factor Lincoln and Guba suggests the criterion auditability to be the criterion relating to the consistency

of qualitative findings. Auditable means that the "decision trail" used by the investigator can be clearly followed by another researcher. Given the data, perspective and situation, it should be possible to arrive at comparable, but not contradictory results.

4. Neutrality (objectivity) : For this factor Lincoln and Guba suggest confirmability as the criterion of neutrality. This is achieved when truth value, applicability, and auditability are established. Contrary to quantitative criteria, it is important to reduce the distance between investigator and subject, and to eliminate artificial lines between subjective and objective reality, to enhance the meaningfulness of findings (Sandelowski, 1986). Engagement rather than detachment is viewed as beneficiary in the search of the meanings individuals give, or derive from their experiences.

Validity

In a more contemporary synthesis of validity criteria in qualitative research Whittemore et al. (2001) propose a repertoire of validity criteria in qualitative research.

The proposed criteria for validity are based on a synthesis of the work of many scholars, and it is stated that the concept of validity is illustrated through the explication and differentiation of primary criteria, secondary criteria, and techniques. Primary criteria are necessary but insufficient in and of themselves. Secondary criteria provide further benchmarks of quality, being more flexible as applied to particular investigations. Techniques are the methods employed to diminish identified validity threats. The criteria should be used within the context of a particular investigation.

> *Because qualitative research is often defined by uncertainty, fluidity, and emergent ideas, so too must be the validity criteria that give credence to these efforts. Therefore, it is logical to extend this flexibility to the determination of the most appropriate validity criteria for each investigation.* (Whittemore et al., 2001, p. 528)

Reliability

The concept of reliability is often considered a precursor for validity. With a manifest content analysis, reliability is considered necessary but not sufficient condition for validity (Potter and Levine-Donnerstein, 1999). Coding is for capturing surface features of the data, and the agreement among coders should be high due to low requirement of judgement. Tests for reliability in coding are *Stability*, *Reproducibility* and *Accuracy*. Stability is the degree to which a process is unchanging over time. This requires a test-retest procedure, to see if coding is stable when performed at different occasions. This test is the weakest due to the risk of coders remembering their coding. Reproducibility is the degree to which a process can be recreated in different settings. This test requires a test-test procedure where the same content is analysed by different coders. If coders produce the same coding pattern, the data is regarded as reliable. Accuracy is the degree to to which a process yields what it is supposed to yield and in this procedure coders' judgement are compared to a standard. Accuracy is the strongest reliability test available, but

not always attainable because in a manifest analysis a standard can not always be established. The major threat to reliability in manifest analysis is fatigue (Potter and Levine-Donnerstein, 1999).

Trustworthiness of this Work

The researcher is always present in qualitative research, and it is inevitable that he/she influences the respondents, the procedures and the findings. Interviewing is a social relationship, and each relationship reflects the personalities of the researcher and the participant, and the ways they interact (Seidman, 1998). Establishing a good relationship must be balanced with the awareness of the unsymmetrical relationship between the interviewer and the interviewee. Interviews are always a dialogue between the two persons involved. There are many things that might influence this dialogue, and this can not be controlled. A conscious choice was to try to use a neutral language during the interviews, to avoid intimidating the respondents with a language more formal than the one they would choose themselves to discuss object orientation. However, this might also have been counterproductive and influenced them to use the same wording, instead of their own vocabulary. The object oriented vocabulary has not been a major part of the analysis, and great effort has been made to listen to descriptions rather than exact wording.

My point of departure is not unbiased regarding the subject, since object oriented quality in examples for novices has been an interest of mine for more than ten years, and the focus of my research for the last three. This research has revealed that the quality of examples in textbooks must be considered low, and our experience is that students who have taken classes on object oriented programming in upper secondary school have been surprisingly unaware of the basics of object orientation. This has raised questions regarding the view of object orientation among educators in general. My personal engagement in, and experience of teaching object oriented programming for many years is both an asset, as well as a threat to objectivity with respect to this study. The conditions for establishing a friendly, trustful relationship with the respondents have been good. So have the possibilities to understand the interviewees, because of the familiarity with the subject. However, this familiarity may also be considered an obstacle, when it comes to viewing the data with an open mind, and without preconceptions. The results of this study is therefore my interpretations of my respondents interpretations, of their views on object orientation, in that particular situation, and at that particular moment.

One limitation of the study is the relatively low number of interviews, which means that the requirement for saturation of data can not be guaranteed. The results have not been triangulated by any complementary study of other sources, neither have they been verified by the respondents through member check, which according to Lincoln and Guba (1985) is one important way of contributing to trustworthiness.

The method of manifest content analysis is less formal than e.g. grounded theory and phenomenography, and lacks the theoretical base necessary to form any theory describing the findings. The reason for choosing qualitative content analysis in this study, is to perform an initial and exploratory investigation, on which it might be possible to continue the research. The research questions does

not deal with any interpretations of experiences, or processes in human life, which would suggest the use of a more theory-based method, e.g. grounded theory.

Since the study presented here is not based on an existing theory, the categories have not been decided on in advance. The aim has been to explore the way educators think about object orientation as open minded as possible.

It is my belief that his work should be considered reliable since the process is stable over time, which has been established by several coding runs by the single coder involved in this work. The process can be recreated in a different setting, which has been validated by a test-test procedure with a second researchers coding the same data with only minor, and insignificant, differences. About 17% of all statements were randomly selected and classified. The classifications were then compared to the ones made by the author, and any differences resolved. The major part of differences in classifications, was due to the interpretation of the aspects of the theme *Examples*. For some instances the validating researcher had misinterpreted the aspects. The theme *Examples* was concerning the educator's view on examples in general, any observed student difficulties with problems, and strategies for choosing examples, and not the specific examples chosen. A few labels were changed, and a few statements had additional labels compared to the original classification. None of these changes made any difference for the thematical results. No standards have been available in this work, so the matter of accuracy is not applicable.

The question of validity in qualitative research is a matter of standards to be upheld as ideals (Whittemore et al., 2001). In Table 6.1 criteria for these standards, as well as techniques for upholding them, are shown. Criteria and techniques addressed in this study are marked. By supplying a rich amount of quotations the results are transparent and allow for evaluation on credibility and authenticity. The research process has been tested and the author's long experience and training in counseling skills, working for many years as a student counselor, makes it plausible that the author is concerned of giving voice to all participants, and is sensitive to differences among participants. Therefore the criteria for credibility and authenticity can be regarded as fulfilled.

The design of the study is conscious and articulated. Furthermore, the operationalisation of the research area is structured, data collection decisions are presented, and verbatim transcriptions are provided, which implies thoroughness.

The analysis decisions have been clearly stated, and the categorisation is validated by another researcher. The presentations provides an audit trail, and quotations are supplied to illustrate findings. The researchers perspective is stated, and thick descriptions are provided.

Table 6.1: Primary criteria, Secondary criteria and Techniques for demonstrating validity (Whittemore et al., 2001). Criteria and techniques addressed in this study are marked.

		Assessment	Addressed
Primary Criteria	Credibility	Do the results of the research reflect the experience of participants or the context in a believable way?	X
	Authenticity	Does a representation of the emic perspective exhibit awareness to the subtle differences in the voices of all participants?	X
	Criticality	Does the research process demonstrate evidence of critical appraisal?	
	Integrity	Does the research reflect recursive and repetitive checks of validity as well as a humble presentation of findings?	X
Secondary Criteria	Explicitness	Have methodological decisions, interpretations, and investigator biases been addressed?	X
	Vividness	Have thick and faithful descriptions been portrayed with artfulness and clarity?	X
	Creativity	Have imaginative ways of organizing, presenting, and analyzing data been incorporated?	X
	Thoroughness	Do the findings convincingly address the questions posed through completeness and saturation?	?
	Congruence	Are the process and the findings congruent? Do all the themes fit together? Do findings fit into a context outside the study situation?	X
	Sensitivity	Has the investigation been implemented in ways that are sensitive to the nature of human, cultural, and social contexts?	X
Techniques for demonstrating validity	Design consideration	Developing a self-conscious research design	X
		Sampling decisions (i.e., sampling adequacy)	X
		Employing triangulation	
		Giving voice	X
		Sharing perquisites of privilege	
		Expressing issues of oppressed group	
	Data generating	Articulating data collection decisions	X
		Demonstrating prolonged engagement	
		Demonstrating persistent observation	
		Providing verbatim transcription	X
		Demonstrating saturation	
	Analytic	Articulating data analysis decisions	X
		Member checking	
		Expert checking	X
		Performing quasistatistics	
		Testing hypotheses in data analysis	
		Using computer programs	X
		Drawing data reduction tables	X
		Exploring rival explanations	
		Performing a literature review	X
		Analyzing negative case analysis	
		Memoing	
		Reflexive journaling	
		Writing an interim report	
		Bracketing	
	Presentation	Providing an audit trail	X
		Providing evidence that support interpretations	X
		Acknowledging the researcher perspective	X
		Providing thick descriptions	X

Chapter 7

Conclusions and Further Work

In this work we have investigated two major aspects of object orientation in the educational context: the definition of object oriented quality, primarily focused on examples, and the nature of educators views of object orientation. To be able to address the issue of object oriented quality, the characteristics of object orientation have been investigated, and a set of Eduristics (educational heuristics) for the design of object oriented examples for novices has been defined. Using the evaluation tool described in Paper III, it has been possible to examine the state-of-the-art of common text book examples. The result of this investigation is discouraging, showing low scores, particularly for first user defined classes. Based on the results of this investigation, we have shown that the examples that score high on object oriented qualities, are upholding the Eduristics.

The learning outcomes of education are to a large extent dependent on the educators. They are the ones designing the introduction of object orientation, and the examples, exercises and assignments supporting the presentation. This is the reason for taking an interest in educators views on different aspects concerning object orientation, and the strategies for teaching it. The results of this part of the work, is that the educators in general, have a rather simple conceptual view of object orientation. Object oriented analysis and design is not introduced, or practiced, and the novices are not supported on how to choose and design objects in a problem domain by any systematic approach.

The level of abstraction in object orientation adds to the effort of both the mediator of knowledge, whether in the form of a teacher, a text book or any other supporting material, and the novice, compared to learning problem solving and programming through the imperative paradigm. It is therefore important that we as educators consider the object oriented quality of our mission. In this work we have shown that it is possible to discuss object oriented quality in examples for novices. It is also demonstrated that the object oriented quality of examples in popular object oriented introductory programming textbooks can be improved. We have also illustrated how details can make all the difference in quality, and that some examples can be improved by small changes. Educators from both upper secondary schools and universities show that there is a need for a conceptual approach to teaching object orientation. If the teaching and learning of object orientation is to be successful, in the sense that the novices will achieve a suitable, and sustainable, conceptual model of object orientation, emphasis must be on conveying a proper

view of the paradigm, in all aspects of teaching.

Moving on with this research...

The very first examples in an introduction to object orientation must be regarded as critical, and at the same time, they seem to be the most difficult ones to design. This research has shown that these examples have a particular difficulty in upholding object oriented principles. If we are aiming at teaching object orientation, and not just syntax of a language that supports object oriented features, the first user defined classes (FUDC's) must be further investigated and developed.

We know that examples of good quality, does not contradict the Eduristics. The next step would be to use the Eduristics to design a number of exemplary examples, for educators to evaluate. This way we could supplement the results of this work, in that design according to the Eduristics generates commonly appreciated qualities in examples.

Since we discovered that educators' overall impression of an example changed, sometimes drastically, due to the evaluation process, it would be interesting to have educators use the Eduristics in designing their own examples, or analysing and eventually redesigning favourite examples already used. Through this, it would be possible to investigate if, and in that case how, their perception of object orientation changes due to the use of the heuristics would be useful for educational purposes.

It would also be interesting to investigate if the awareness of the concept of object oriented quality would support novices in learning.

Working with composition early to emphasise the collaborating characteristic of object orientation is important, and further investigations of textbook treatment of collaboration would give valuable insights on the treatment of the subject.

Another important point, is reaching out to practitioners. The practical consequences of the results in this work, have to be disseminated to educators in several ways. Initiating a debate on object orientation and a conceptual approach to teaching it, is absolutely necessary. Furthermore, suggestions of practical ways of implementing object thinking in different educational settings must be developed. Educators, particularly in upper secondary school, work on a tight schedule, and have little time to spend on structural issues, and if suggestions are practical enough this would most likely be appreciated and used. This work would by necessity be conducted in collaboration with educational developers working with upper secondary school.

> *The purpose of empirical research is not only to observe behaviour, but to* think about *behaviour. Empirical science in young domains such as CS education is not so much a process of getting answers as one of finding even better questions.* (Fincher and Petre, 2004, p.23)

Finally

It is my firm belief that the way we teach is formed by the way we think about a certain subject, and the experiences we gather. In my mind object orientation, and the teaching of it, matures with growing experience, and successes and failures in teaching. Developing the way we think about object orientation, and objects,

should contribute to the way we teach, not least in improving the quality of examples and exercises, which are basic tools in teaching. It is hard to imagine that it would be possible in other scientific subjects, for each and everyone to define what the essence of the subject is. Therefore it is important to move away from a trial-and-error way of developing teaching, and instead deploy a more structured, and scientifically based approach.

This is an important issue, and the community of computer science education researchers need to start working on a common framework for teaching object orientation, if we are to change this situation and better support both educators and novices.

Chapter 8

Summary of Papers

8.1 Paper I - Transitioning to OOP/Java – A Never Ending Story

To support the design of an objects-first course we developed a list of principles, to guide course development. These principles were either based on our teaching experience, or the collected advice and experience from the literature in computer science education. The outcome of this approach is evaluated.

Eleven principles to guide the design of introductory programming courses are suggested. They are grouped into three categories: *High-level goals* (P1-P4), *Tools* (P5-P7) and *Pragmatics* (P8-P11).

P1: No magic! Nothing should have to be "explained later", and everything introduced should be transparent. The novice's frame of reference should be respected and new material should

P2: Objects from the very beginning. It is important to promote objects consistently to reinforce the building blocks of object oriented problem solving and programming.

P3: General concepts favoured over language specific realisations. This means that the course should be organised around concepts of object orientation rather than language constructs.

P4: No exceptions to general rules. We must always "do as we say," only use sound and meaningful objects, only show well-designed classes, and certainly not do unnecessary main-programming.

P5: OOA&D early. We have to provide students with simple tools to approach a problem systematically and to evaluate alternative solutions before starting to code. Early OOA&D conveys to the students that responsibilities are distributed amongst the objects that solve a problem.

P6: Exemplary examples. All examples used in classes and exercises should comprise well-designed classes that fill a purpose (besides exemplifying a certain concept or language specific detail).

P7: Easy-to-use tools. To promote thinking in objects, we decided to use an IDE that enforced the work with objects as separate entities, in our case it is BlueJ(BlueJ). Furthermore, we decided to use CRC-cards and role-play to introduce object oriented analysis and design.

P8: Hands-on. Topics should be reinforced by means of practical exercises. Lectures should be followed by supervised in-lab sessions., and for each session, step-by-step instructions and exemplary examples should be provided.

P9: Less "from scratch" development. "Reading before modifying before coding."

P10: Alternative forms of examination. Assessment should support learning, so we use peer-reviewing among students, and a combination of theoretical and practical programming exams was developed.

P11: Emphasise the limitations of computers. Students should learn that computations can produce erroneous or unexpected results due to limitations in data representation, even in logically correct programs.

8.2 Paper II - Heuristics for designing OO examples for novices

Describes the establishment of characteristics of object oriented, based on how they are manifested in concepts and design principles in literature. Defines a number of heuristics for the design of object oriented examples for novices, called Eduristics. These heuristics are validated against the set of concepts and design principles previously established. They are also discussed in relation to the quality factors used for the evaluation of examples, described in Paper IV. Finally it discusses the object oriented quality in two examples, using the Eduristics to show the sometimes small, but significant, details that contradicts the general idea of the paradigm.

8.3 Paper III - Evaluating OO example programs for CS1

This paper contains the results of the first attempt to design an evaluation checklist/tool for introductory object oriented examples. A pilot study of the tool was made with five representative examples covering the following aspects; the very first example of a textbook, the first exemplification of developing a user-defined class, the first application involving at least two interacting classes and a non-trivial (but still simple) example of using inheritance. The concept of quality factor was defined and it was decided to structure the checklist according to three categories; *technical quality*, *object-oriented quality* and *didactical quality*. The results showed that such an instrument is a useful tool for indicating particular strengths and weaknesses of examples, it was evident that the instrument distinguished between examples. However, the analysis also showed that the evaluation instrument presented was not reliable enough for evaluations on a larger scale; inter-rater agreement was too low. Some of the shortcomings were due to semantical issues concerning the criteria, and the lack of instruction on how to grade non-existing features.

Further details of this work can be found in Börstler et al. (2008b).

The development of the tool continued and the resulting tool, and its evaluation and use, is described in (Börstler et al., 2009).

8.4 Paper IV - On the Quality of Examples in Introductory Java Textbooks

In this paper we perform a more detailed analysis of the large data material collected for the ITiCS'09 working group (Börstler et al., 2009). The data analysed in this paper consists of 191 evaluations of 21 examples by 24 raters. The examples were collected from 11 popular textbooks. On average the participating raters had more than 10 years of experience with teaching object orientation specifically.

Reviewers ranked examples in very similar ways, although their absolute ratings could be quite different. The majority of reviewers show a very strong and highly significant correlation with the total average ranking of all reviews. This strongly indicates that our evaluation instrument is reliable.

The participating textbooks were thoroughly classified as being either object oriented (OO), or following a traditional approach (Trad), with a clear focus on elementary programming concepts as loops, primitive data types, expressions, conditions etc., presented before object oriented concepts and components. One result from this study is that although most texts claim to focus on object orientation and follow some kind of object-early or -centric approach, a closer inspection revealed that the texts in the OO category were actually in a minority.

Three different types of examples were evaluated; FUDC: First User-Defined Class, OOD: Multiple User Defined Classes, and CS: Control Structures. Since we only considered examples from popular textbooks, we expected most of the scores to be in the upper positive range. However, as many as 10 out of the 21 examples scored below 10 in a range of [-30, 30] and received an overall final impression ≤ 0. The raters were asked to give an overall impression before and after the evaluation. The overall impression seems to degrade during the review, in particular for the examples that already have a low overall first impression. This indicates that the checklist might help to spot problems that might be easily overlooked.

The FUDC examples are crucial to give a correct first impression of object orientation. One result of the analysis is that the variation in object oriented quality factors for FUDC examples is much greater than the variation for CS and OOD examples. FUDC's are difficult to design because of the restrictions, but it appears that examples illustrating meaningful behaviour display higher object oriented quality than examples which include no or trivial behaviour. This gives an indication of what to focus on in the design.

Interesting to note, is that making a systematic evaluation of examples often changes the standpoint on the quality of the example. There are details of an example, overlooked on a superficial examination, that are revealed when using the instrument. Text book examples should be designed to give a more consistent presentation of object orientation, particular care must be taken to design the first user defined classes (FUDCs).

8.5 Paper V - Educators' Views on OO, Objects and Examples

Reports on an exploration and classification of educators' personal views on object orientation, objects and examples for object orientation. Qualitative content

analysis is used to analyse the textual data, collected as interviews.

The views differentiate substantially in terms of complexity. They are often conceptually simple in relation to the educational aim to convey the idea of object orientation. The categories reflecting the educators views of object orientation, object and examples are related on a conceptual level and are ranging from elementary syntax-based views to abstract, problem solving paradigmatic views in the three aspects investigated. Every-day-life contexts are used more as an introduction to object orientation, while the choices of context for the practical work is slightly different.

8.6 Paper VI - Educators' Strategies for OOA&D

Reports on a classification of educators' strategies for introducing object orientation and for teaching object oriented analysis and design (OOA&D). Qualitative content analysis is used to analyse the textual data, collected as interviews.

The methods for introducing OOA&D are mostly implicit, if present at all. Some educators does not seem to form a view of OOA&D of their own. The overall impression is that students do not get support to understand and practice any object oriented problem solving approach.

8.7 Paper VII - Improving OO Example Programs

Being careful about certain aspects of examples can significantly improve the object oriented quality and avoid unintentional non-object oriented characteristics. Every example has to contribute to the overall mission to support the development of a conceptually correct base for object orientation.

With the use of heuristics for object oriented quality in examples for novices, we examine a typical example of a first user defined class (FUDC). Critical details are illustrated and discussed. This is followed by a more general discussion on common deficiencies in textbook FUDC's. Alternative designs are proposed and finally we suggest a number of suitable abstractions to use in examples along with appropriate contexts.

Bibliography

ACM (2001). Computing curricula 2001. http://www.acm.org/education/curric_vols/cc2001.pdf Webpage last visited: 2008-12-15.

ACM (2008a). Curricula recommendations. http://www.acm.org/education/curricula-recommendations Last visited: 2008-12-15.

ACM (2008b). Java Task Force. http://www-cs-faculty.stanford.edu/~eroberts//jtf/, Last visited: 2008-12-15.

Armstrong, D. J. (2006). The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128.

Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, 70(2):181–214.

Atkinson, R. K., Renkl, A., and Merrill, M. M. (2003). Transitioning from studying examples to solving problems: Effects of self-explanation prompts and fading worked-out steps. *Journal of Educational Psychology*, 95(4):774 – 783.

Bell, D. and Parr, M. (2010). *Java For Students, 6/E*. Pearson International.

Bellin, D. and Simone, S. S. (1997). *The CRC Card Book*. Addison-Wesley.

Bennedsen, J. (2008). *Teaching and learning introductory programming: a model-based approach*. PhD thesis, Oslo.

Bennedsen, J. and Schulte, C. (2007). What does 'objects-first' mean? an international study of teachers' perceptions of objects-first. In Lister, R. and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 21–29, Koli National Park, Finland. ACS.

Bergin, S. and Reilly, R. (2005). Programming: factors that influence success. *SIGCSE Bull.*, 37(1):411–415.

Biddle, R., Noble, J., and Tempero, E. (2002). Reflections on crc cards and oo design. In Noble, J. and Potter, J., editors, *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10, pages 201–205, Sydney, Australia. ACS.

Bloch, J. (2001). *Effective Java Programming Language Guide*. Addison-Wesley, 1st edition.

BlueJ. Bluej homepage. `http://www.bluej.org`, Webpage last visited 2010-11-05.

Börstler, J. (2005). Improving crc-card role-play with role-play diagrams. In *Conference Companion 20th Annual Conference on Object Oriented Programming Systems Languages and Applications*, pages 356–364. ACM.

Börstler, J., Caspersen, M. E., and Nordström, M. (2007). Beauty and the beast—toward a measurement framework for example program quality. Technical Report UMINF-07.23, Dept. of Computing Science, Umeå University, Umeå, Sweden.

Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Kallin Westin, L., Jan-ErikMoström, and Caspersen, M. E. (2008a). Evaluating oo example programs for CS1. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, pages 47–52, New York, NY, USA. ACM.

Börstler, J., Hall, M. S., Nordström, M., Paterson, J. H., Sanders, K., Schulte, C., and Thomas, L. (2009). An evaluation of object oriented example programs in introductory programming textbooks. *Inroads*, 41:126–143.

Börstler, J., Johansson, T., and Nordström, M. (2002). Introducing oo concepts with crc cards and bluej—a case study. In *Proceedings Frontiers in Education Conference, FIE'02*, pages T2G–1–T2G–6.

Börstler, J., Nordström, M., Kallin Westin, L., Jan-ErikMoström, Christensen, H. B., and Bennedsen, J. (2008b). An evaluation instrument for object-oriented example programs for novices. Technical Report UMINF-08.09, Dept. of Computing Science, Umeå University, Umeå, Sweden.

Börstler, J., Nordström, M., and Paterson, J. H. (2010). On the quality of examples in introductory java textbooks. *The ACM Transactions on Computing Education (TOCE)*, Accepted for publication.

Bruce, K. (2004). Controversy on how to teach CS1: A discussion on the sigcse-members mailing list. *ACM SIGCSE Bulletin*, 36(4):29–35.

CACM (2002). Hello, world gets mixed greetings. *Communications of the ACM*, 45(2):11–15.

CACM (2005). For programmers, objects are not the only tools. *Communications of the ACM*, 48(4):11–12.

Cant, S., Jeffery, D. R., and Henderson-Sellers, B. (1995). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7):351–362.

Carbone, A., Hurst, J., Mitchell, I., and Gunstone, D. (2000). Principles for designing programming exercises to minimise poor learning behaviours in students. In *Proceedings ACE'00*, pages 197–201.

Carbone, A., Hurst, J., Mitchell, I., and Gunstone, D. (2001). Characteristics of programming exercises that lead to poor learning tendencies: Part ii. In *ITiCSE '01: Proceedings of the 6th annual conference on Innovation and technology in computer science education*, pages 93–96, New York, NY, USA. ACM.

Caspersen, M. E. (2007). *Educating Novices in The Skills of Programming*. PhD thesis, University of Aarhus, Denmark.

Caspersen, M. E. and Kölling, M. (2006). A novice's process of object-oriented programming. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 892–900, New York, NY, USA. ACM.

Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., and Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science*, 13(2):145–182.

Chidamber, S. R. and Kemerer, C. F. (1991). Towards a metrics suite for object oriented design. In *ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, Phoenix, Arizona, United States.

Dale, N. (2005). Content and emphasis in CS1. *ACM SIGCSE Bulletin*, 37(4):69–73.

Dale, N. B. (2006). Most difficult topics in CS1: results of an online survey of educators. *SIGCSE Bull.*, 38(2):49–53.

de Raadt, M., Watson, R., and Toleman, M. (2004). *Introductory Programming: What's Happening Today and Will There Be Any Students to Teach Tomorrow?*, volume 30, pages 277–282. Australian Computer Society.

Détienne, F. (1997). Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams. *Interacting with Computers*, 9(1):47–72.

Devlin, K. (2003). Why universities require computer science students to take math : Introduction. *Commun. ACM*, 46(9):36–39.

Dodani, M. H. (2003). Hello world! goodbye skills! *Journal of Object Technology*, 2(1):23–28.

Du Bois, B., Demeyer, S., Verelst, J., and Temmerman, T. M. M. (2006). Does god class decomposition affect comprehensibility? In Kokol, P., editor, *SE 2006 International Multi-Conference on Software Engineering*, pages 346–355. IASTED.

Eckerdal, A. (2009). *Novice Programming Students' Learning of Concepts and Practise*. PhD thesis, Uppsala UniversityUppsala University, Division of Scientific Computing, Numerical Analysis.

Eckerdal, A., Thuné, M., and Berglund, A. (2005). What does it take to learn 'programming thinking'? In *ICER '05: Proceedings of the first international workshop on Computing education research*, pages 135–142, New York, NY, USA. ACM.

Elo, S. and Kyngas, H. (2008). The qualitative content analysis process. *Journal of Advanced Nursing*, 62(1):107–115.

Fincher, S. and Petre, M. (2004). *Computer science education research*. Taylor & Francis, London.

Fleury, A. E. (2000). Programming in java: Student-constructed rules. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 197–201.

Forman, J. and Damschroder, L. (2007). Qualitative content analysis. *Advances in Bioethics*, 11:39–62.

Fowler, M. (2003). When to make a type. *IEEE Software*, 20(1):12–13.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Ralph, E. J., and Vlissides, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman.

Garzás, J. and Piattini, M. (2007). *Object-oriented Design Knowledge: Principles, Heuristics, and Best Practices*. Idea Group Publishing, USA.

Gibbon, C. A. and Higgins, C. A. (1996). Towards a learner-centred approach to teaching object-oriented design. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*. IEEE Computer Society.

Graneheim, U. H. and Lundman, B. (2004). Qualitative content analysis in nursing research: concepts, procedures and measures to achieve trustworthiness. *Nurse Education Today*, 24(2):105 – 112.

Gray, K. A., Guzdial, M., and Rugaber, S. (2002). Extending crc cards into a complete design process. Technical report, College of Computing, Georgia Institute of Technology, Atlanta, GA.

Grotehen, T. (2001). *Objectbase Design: A Heuristic Approach*. PhD thesis, University of Zurich, Switzerland.

Henderson-Sellers, B. and Edwards, J. (1994). *BOOK TWO of object-oriented knowledge: the working object: object-oriented software engineering: methods and management*. Prentice-Hall, Inc.

Holland, S., Griffiths, R., and Woodman, M. (1997). Avoiding object misconceptions. In *Proceedings of the 28th Technical Symposium on Computer Science Education*, pages 131–134.

Hsieh, H.-F. and Shannon, S. E. (2005). Three Approaches to Qualitative Content Analysis. *Qualitative Health Research*, 15(9):1277–1288.

Johnson, R. and Foote, B. (1988). Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2).

Kay, A. C. (1996). The early history of smalltalk. pages 511–598. ACM, New York, NY, USA.

Kölling, M. (2006). I object – posting on SIGCSE-MEMBERS mailinglist 2006-11-13. `http://www.bluej.org/mrt/docs/objection.html` Webpage last visited 2010-11-05.

Kölling, M. and Rosenberg, J. (2001). Guidelines for teaching object orientation with java. In *Proceedings of the 5th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 33–36.

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42.

Lahtinen, E., Ala-Mutka, K., and Järvinen, H. (2005). A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 14–18.

Lanza, M., Marinescu, R., and Ducasse, S. (2005). *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc. Secaucus, NJ, USA.

Lieberherr, K. and Holland, I. (1989). Assuring good style for object-oriented programs. *IEEE Software*, 6(5):38–48.

Lincoln, Y. S. and Guba, E. G. (1985). *Naturalistic inquiry*. Sage, Beverly Hills, Calif.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hame, J., Lindholm, M., McCartney, R., Moström, J.-E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull.*, 36(4):119–150.

Lister, R., Berglund, A., Box, I., Cope, C., Pears, A., Avram, C., Bower, M., Carbone, A., Davey, B., de Raadt, M., Doyle, B., Fitzgerald, S., Mannila, L., Kutay, C., Peltomäki, M., Sheard, J., Simon, Sutton, K., Traynor, D., Tutty, J., and Venables, A. (2007). Differing ways that computing academics understand teaching. In *ACE '07: Proceedings of the ninth Australasian conference on Computing education*, pages 97–106, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., Hitchner, L., Luxton-Reilly, A., Sanders, K., Schulte, C., and Whalley, J. L. (2006). Research perspectives on the objects-early debate. *SIGCSE Bull.*, 38(4):146–165.

Malan, K. and Halland, K. (2004). Examples that can do harm in learning programming. In *Companion to the 19th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 83–87.

Mäntylä, M. A taxonomy for "bad code smells". `http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm`, Webpage last visited 2008-10-17.

Mäntylä, M. (2003). Bad smells in software – a taxonomy and an empirical study. Master's thesis, Helsiniki University of Technology.

Martin, R. C. (2003). *Agile Software Development, Principles, Patterns, and Practices*. Addison-Wesley.

McConnell, J. J. and Burhans, D. T. (2002). The evolution of CS1 textbooks. In *Proceedings FIE'02*, pages T4G–1–T4G–6.

McCracken, M., Wilusz, T., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., Laxer, C., Thomas, L., and Utting, I. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulletin*, 33(4):125–180.

Mead, J., Gray, S., Hamer, J., James, R., Sorva, J., Clair, C. S., and Thomas, L. (2006). A cognitive approach to identifying measurable milestones for programming skill acquisition. In *Working group reports on ITiCSE on Innovation and technology in computer science education*, Bologna, Italy. ACM.

Meyer, B. (2001). Software engineering in the academy. *IEEE Computer*, 34(5):28–35.

Meyer, B. (2006). Testable, reusable units of cognition. *IEEE Computer*, 39(4):20–24.

Morse, J. M. (2000). Determining sample size. *Qualitative Health Research*, 10(1):2–3.

Nordström, M. (2009). He[d]uristics – heuristics for designing object oriented examples for novices. Licenciate Thesis, Umeå University, Sweden.

Nygaard, K. (1986). Basic concepts in object oriented programming. *SIGPLAN Not.*, 21(10):128–132.

Opdyke, W. F. (1992). Refactoring object-oriented frameworks. Master's thesis, University of Illinois at Urbana-Champaign, USA.

Paas, F., Renkl, A., and Sweller, J. (2003). Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist*, 38(1):1–4.

Parnas, D. L. (2007). Use the simplest model, but not too simple. *Communications of the ACM - Forum*, 50(6):7–9.

Parsons, J. and Wand, Y. (1997). Choosing classes in conceptual modeling. *Communications of the ACM*, 40(6):63–69.

Pears, A., East, P., Mccartney, R., Ratcliffe, M. B., Stamouli, I., Kinnunen, P., Moström, J.-E., Schulte, C., Eckerdal, A., Malmi, L., Murphy, L., Simon, B., and Thomas, L. (2007). What's the problem? teachers' experience of student learning successes and failures. Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Koli National Park, Finland, November 15-18, 2007.

Pirolli, P. L. and Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian journal of psychology*, 39(2):240–272.

Potter, W. J. and Levine-Donnerstein, D. (1999). Rethinking validity and reliability in content analysis. *Journal of Applied Communication Research*, 27(3):258.

Purao, S. and Vaishnavi, V. (2003). Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221.

Ragonis, N. and Ben-Ari, M. (2005). On understanding the statics and dynamics of object-oriented programs. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, pages 226–230.

Ramalingam, V. and Wiedenbeck, S. (1997). An empirical study of novice program comprehension in the imperative and object-oriented styles. In *ESP '97: Papers presented at the seventh workshop on Empirical studies of programmers*, pages 124–139, New York, NY, USA. ACM.

Randolph, J. J. (2007). *Computer science education research at the crossroads: a methodological review of computer science education research, 2000–2005.* PhD thesis, Logan, UT, USA. Adviser-Julnes, George.

Renkl, A. (1997). Learning from worked-out examples: A study on individual differences. *Cognitive Science*, 21(1):1 – 29.

Renkl, A., Atkinson, R. K., Maier, U. H., and Staley, R. (2002). From example study to problem solving: Smooth transitions help learning. *The Journal of Experimental Education*, 70(4):293 – 315.

Riel, A. J. (1996). *Object-Oriented Design Heuristics.* Addison-Wesley.

Rist, R. (1995). Program structure and design. *Cognitive Science*, 19(4):507–561.

Rist, R. S. (1989). Schema creation in programming. *Cognitive Science*, 13(3):389 – 414.

Robins, A., Rountree, J., and Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2):137–172.

Rosson, M. B. and Alpert, S. R. (1990). The cognitive consequences of object-oriented design. *Human-Computer Interaction*, 5(4):345.

Sajaniemi, J. and Kuittinen, M. (2008). From procedures to objects: A research agenda for the psychology of object-oriented programming education. *Human Technology*, 4(1):75—91.

Sandelowski, M. (1986). The problem of rigor in qualitative research. *Advances in Nursing Science*, 8(3):27– 37.

Sandelowski, M. (1995). Sample size in qualitative research. *Research in Nursing & Health*, 18(2):179–183.

Schulte, C. and Bennedsen, J. (2006). What do teachers teach in introductory programming? In *ICER '06: Proceedings of the second international workshop on Computing education research*, pages 17–28, New York, NY, USA. ACM.

Seidman, I. (1998). *Interviewing as qualitative research : a guide for researchers in education and the social sciences*. Teachers College Press, New York, 2. ed. edition.

Silverman, D. (2006). *Interpreting qualitative data : methods for analyzing talk, text and interaction*. SAGE, London, 3., [updat.] ed. edition.

Skolverket (2010a). The swedish national agency for education—-homepage. `http://www.skolverket.se/sb/d/353` Last visited: 2010-09-30.

Skolverket (2010b). The swedish national agency for education: Syllabuses. `http://www3.skolverket.se/ki03/front.aspx?sprak=EN` Last visited: 2010-09-30.

Spohrer, J. C. and Soloway, E. (1986). Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632.

Stroustrup, B. (1995). Why c++ is not just an object-oriented programming language. In *OOPSLA '95: Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*, pages 1–13, New York, NY, USA. ACM.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science*, 12(2):257–285.

Sweller, J. and Cooper, G. (1985). The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2:59–89.

Sweller, J., van Merrienboer, J., and Paas, F. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, 10(3):251–296.

Tennyson, R. D. and Cocchiarella, M. J. (Spring 1986). An empirically based instructional design theory for teaching concepts. *Review of Educational Research*, 56(1):40–71.

Thompson, E. (2008). *How do they understand? Practitioner perceptions of an object-oriented program*. PhD thesis, Massey University, Palmerston North, New Zealand.

Trafton, J. G. and Reiser, B. J. (1993). The contributions of studying examples and solving problems to skill acquisition y. In *The proceedings of the 15th annual conference of the Cognitive Science society*, pages 1017–1022.

Transcriva. Transcriva homepage. `http://www.bartastechnologies.com/products/transcriva/`.

Valentine, D. W. (2004). Cs educational research: a meta-analysis of sigcse technical symposium proceedings. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 255–259, New York, NY, USA. ACM.

VanLehn, K. (1996). Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539.

Vygotsky, L. S. (1978). *Mind in society: the development of higher psychological processes*. Cambridge, MA: Harvard University Press.

West, D. (2004). *Object Thinking*. Microsoft Press.

Westfall, R. (2001). 'hello, world' considered harmful. *Communications of the ACM*, 44(10):129–130.

Whitson, J. A. and Galinsky, A. D. (2008). Lacking Control Increases Illusory Pattern Perception. *Science*, 322(5898):115–117.

Whittemore, R., Chase, S. K., and Mandle, C. L. (2001). Validity in Qualitative Research. *Qualitative Health Research*, 11(4):522–537.

Wick, M. R., Stevenson, D. E., and Phillips, A. T. (2004). Seven design rules for teaching students sound encapsulation and abstraction of object properties and member data. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, Norfolk, Virginia, USA. ACM.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S., and Corritore, C. L. (1999). A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, 11(3):255–282.

Appendix A

Programming within the Educational System in Sweden

The Swedish National Agency for Education (*Skolverket*) (Skolverket, 2010a) is the central administrative authority for the Swedish public school system for children, young people and adults, as well as for preschool activities and child care for school children. Government and Parliament specify goals and guidelines for preschool and school. The general organisation is shown in Figure A.1.

Schooling starts with a nine-year compulsory type of school. It is composed of 9 school years and each school year consists of a fall and spring semester. Compulsory school is mandatory and is open to all children aged 7-16.

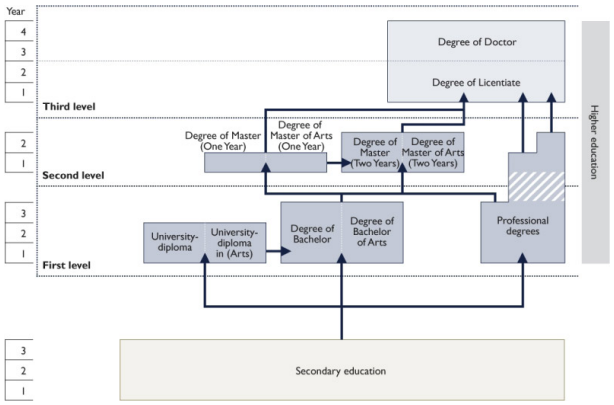

Figure A.1: The swedish educational system.

The National Agency steers, supports, follows up and evaluates the work of municipalities and schools with the purpose of improving quality and the result of activities to ensure that all pupils have access to equal education.

Secondary Education

All young people in Sweden who have finished compulsory school are entitled to three years of schooling at upper secondary school. Based on interest the students make a choice among a number of programs with different focus, yielding different eligibility for moving on to the university level. The government sets out the programme goals of each national programme at upper secondary school. The programme goals describe the purpose and objective of the course. The National Agency for Education adopts syllabi, and the syllabi set out the goals of the teaching of each individual subject and course. Because of this it is well known what the syllabi and requirements for programming courses in upper secondary school are (Skolverket, 2010b).

All courses concerning computers and programming are organized in a subject called Computer technology. In FigureA.2 the structure and relationships among the programming courses in upper secondary school is shown.

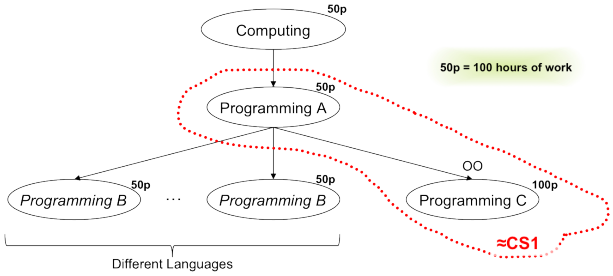

Figure A.2: Computing courses in upper secondary school.

Computing is a course common to most of the programs. It provides knowledge of PCs and skills in using software. *Programming A* provides a basic theoretical and practical knowledge of programming. *Programming B* is aiming at theoretical and practical knowledge in a structured programming language and skills in designing algorithms. *Programming C* should provide theoretical and practical knowledge in an object oriented programming language, as well as a knowledge of analysis and design methods. It also provides knowledge of graphical user interfaces. According to the syllabi, *Programming A* and *Programming C* together roughly contains the amount of stuff and time allocated to a university-level CS1 course. See (Skolverket, 2010b) for more details.

Higher Education

In Sweden, the Government has the overriding responsibility for higher education and research. It enacts the legislation and establishes the targets, guidelines and funding for the sector. At the university level the Swedish educational system is now adjusted to the Bologna system with 3-year bachelor degrees (*Kandidatexamen*) and 2-year Master degrees (Figure A.1). In addition to this, the Master of Science in Engineering (*Civilingenjörsexamen*) is a 5-year Masters degree. These degrees are given for a number of different majors, including Computer Science. In general, the computing curricula of these programs contains traditional CS1 and CS2 courses, for both CS majors and minors.