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 conceptual model of object orientation among educators is low, and the study also shows that novices are not given any introduction to 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 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 knowing you.

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. I can rest in your company, thank You for caring!

My 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, you 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.

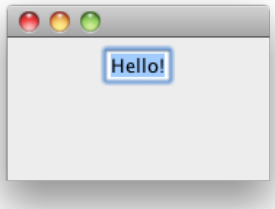


Figure 1.1: Execution of the `Greeting` class

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.

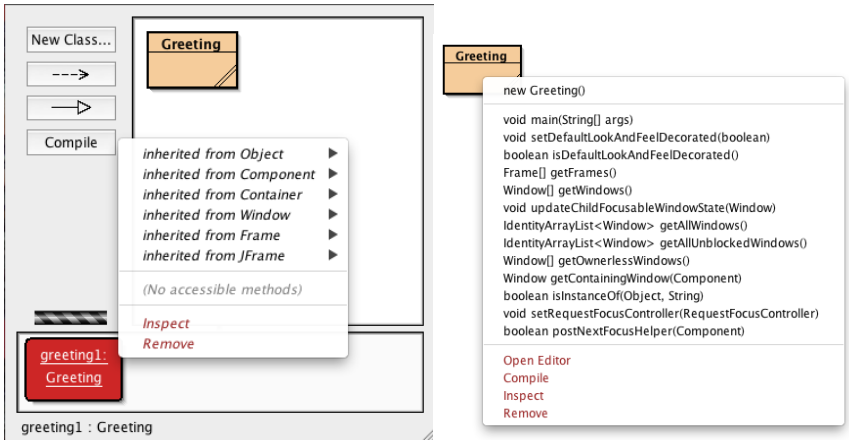


Figure 1.2: Available methods in `Greeting` class (screenshots from BlueJ).

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.

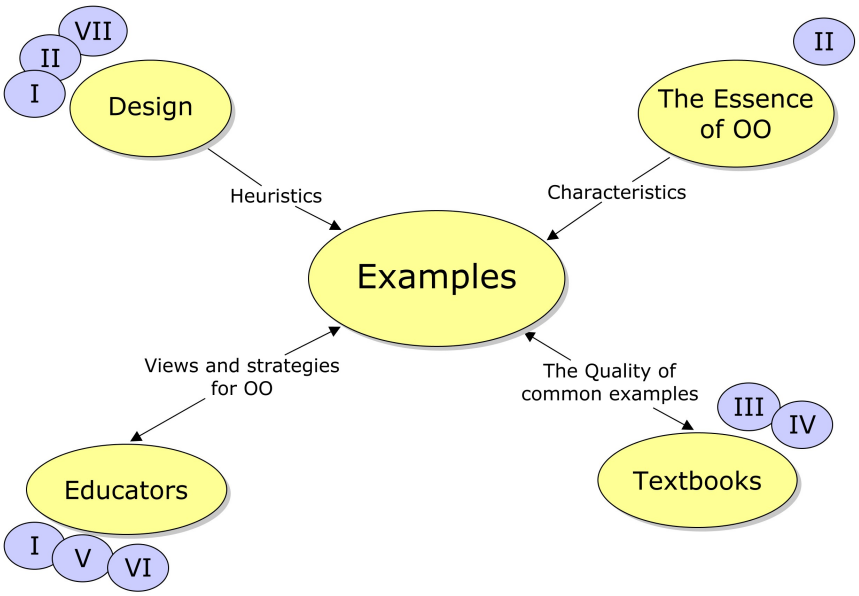


Figure 1.3: Different aspects of examples, and related papers in Roman numerals.

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.

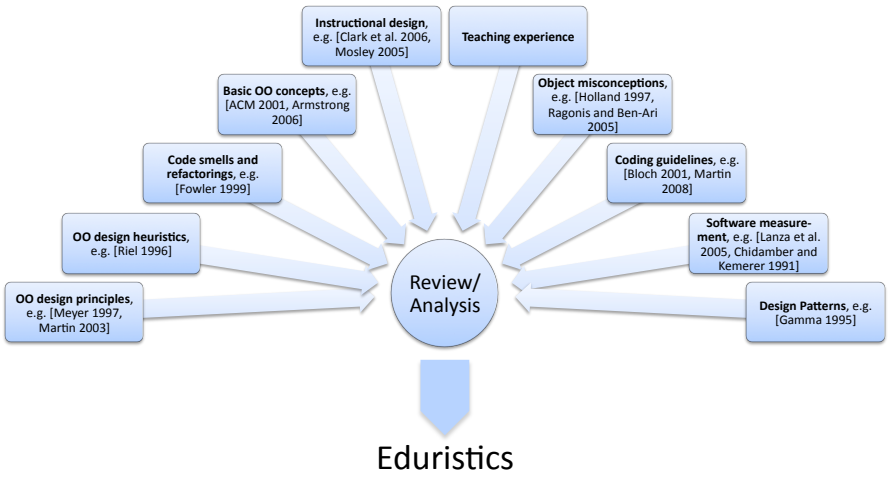


Figure 5.1: Investigating the characteristics of object orientation

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.

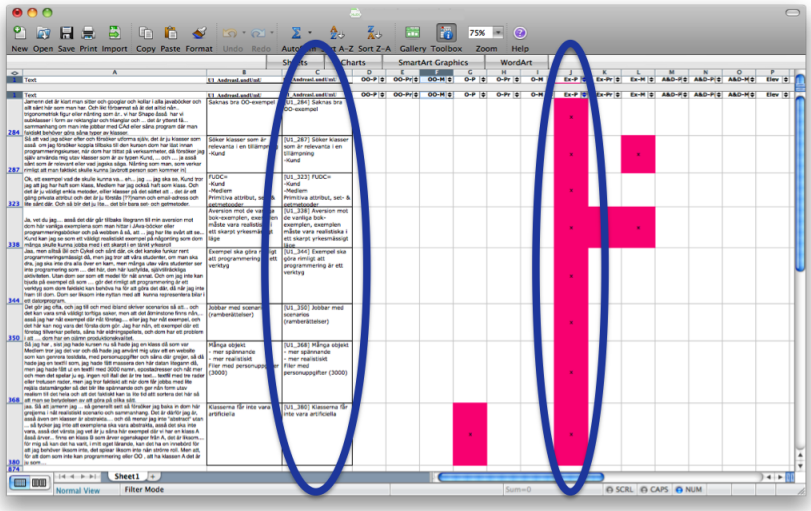


Figure 6.4: Using filter to collect statements for a certain aspect.

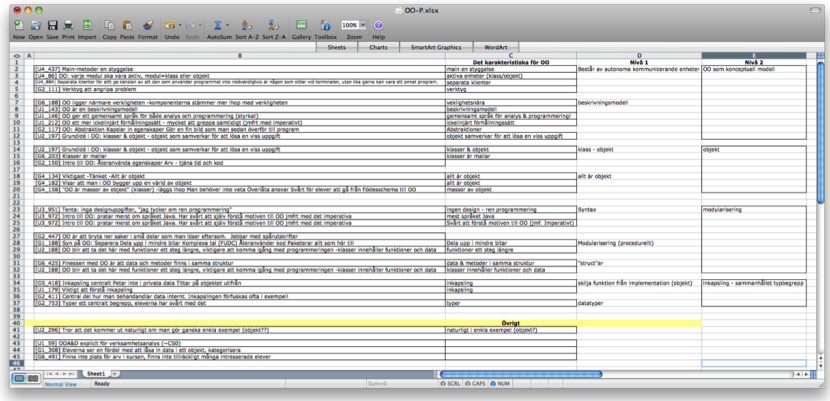


Figure 6.5: Looking for themes.

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.

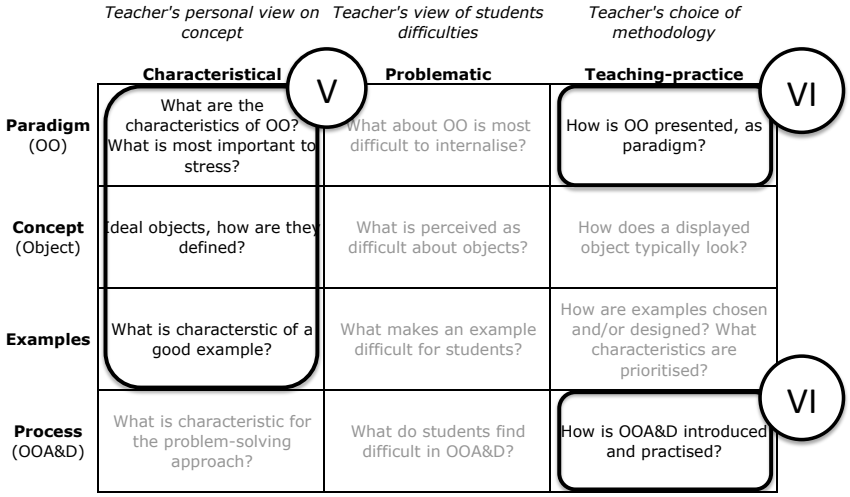


Figure 6.6: Aspects investigated.

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.

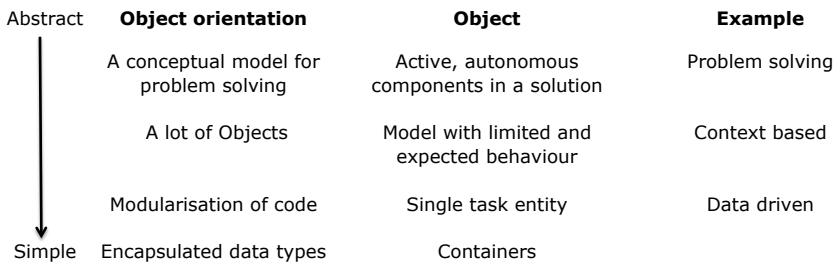


Figure 6.7: Categories of views on *Object orientation*, *Objects*, and *Examples*.

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.

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 used it for all three aspects. Most of the interviewees did not comment on the practice of a conceptual approach, not even when discretely prompted.

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 forces a more conceptual approach, than courses that are purely imperative (Schulte and Bennedsen, 2006).

One interpretation of the 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 rules 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.

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 approach 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 the 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, the 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, 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

What is trustworthiness in qualitative research? This question 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.

Paper I

Transitioning to OOP/Java – A Never Ending Story

Transitioning to OOP/Java — A Never Ending Story

Jürgen Börstler, Marie Nordström, Lena Kallin Westin, Jan-Erik Moström, and Johan Eliasson

Department of Computing Science, Umeå University, Sweden
{jubo,marie,kallin,jem,johane}@cs.umu.se

Abstract. Changing the introductory programming course from a traditional imperative model to an object-oriented model is not simply a matter of changing compilers and syntax. It requires a profound change in course materials and teaching approach to be successful. We have been working with this transition for almost ten years and have realized that teaching object-oriented programming is not as simple or "natural" as some proponents claim. In fact, it has proven difficult to convey to the students the advantages and methodologies associated with object-oriented programming. To help ourselves and others in a transition like this we have developed a number of "course design principles" as well as teaching methods and examples that have proven to have positive influence on student learning outcome.

1 Introduction

The object-oriented paradigm has become the most common programming paradigm for introductory programming courses[1][de Raadt et al., 2004; Stephenson and West, 1998; Chen et al., 2005]. The transition to this paradigm has proven to be more difficult than expected. Traditionally, programming concepts have been systematically introduced one after one, each building nicely on the concepts already learned. Abstract and advanced concepts (e.g., modules and abstract data types) were deferred to later courses. In the object-oriented paradigm, on the other hand, the basic concepts are tightly interrelated and cannot easily be taught and learned in isolation [Roberts et al., 2006], as illustrated in Figure 1. Furthermore, the basic object-oriented concepts are on a higher level of abstraction[2]. Together, this results in a higher threshold and steeper learning curve for the learner.

It is generally accepted that transitioning to the object-oriented paradigm is not just a programming language issue. Object-oriented development requires a new way of thinking [Bacvanski and Börstler, 1997]. This is particularly important in education. A syntax-driven approach can take the students' attention

[1] Even in upper secondary school (high school).
[2] Whether this is an advantage or disadvantage for teaching or learning is unclear.

J. Bennedsen et al. (Eds.): Teaching of Programming, LNCS 4821, pp. 80–97, 2008.

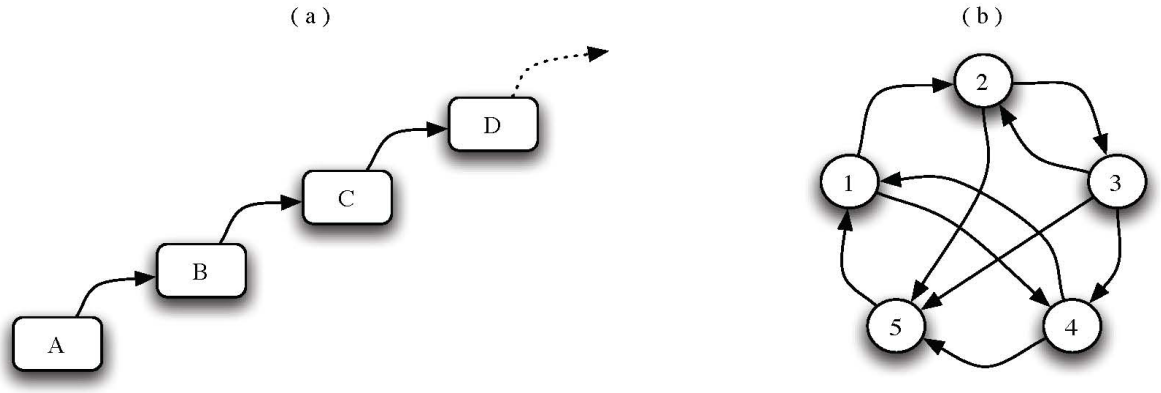


Fig. 1. Dependencies between basic concepts to be introduced in imperative - (a) and object-first approaches (b), respectively

away from the underlying concepts and principles (see also *Model-Driven Programming* by Bennedsen and Caspersen and *CS1: Getting Started* by Caspersen and Christensen). Studies show that there is a mismatch between the programming language used and the paradigm that is actually taught. In Australia for example, about 82% of the introductory programming instructors used an object-oriented language, but only about 37% taught their courses by using an object-oriented approach [de Raadt et al., 2004]. Approaches for teaching introductory programming courses are still heavily discussed [Bruce, 2005].

In this chapter, we describe our experience transitioning from a traditional approach using (Turbo) Pascal to a true objects-first approach in Java. The advantages of using object-orientation for teaching are many. It provides powerful mechanisms for the structuring and organisation of models (in particular designs and code) and decreases the conceptual distance between problem and solution models. This makes it much easier to communicate models and keep them consistent [West, 2004].

However, these advantages come at a price. The basic object-oriented concepts are highly interrelated and cannot easily be taught or learned in isolation (as illustrated in figure 1(b)). There also is no commonly accepted pedagogical approach to overcome this problem [Bruce, 2005]. It is furthermore very difficult, if not impossible, to develop "simple" examples. A proper and meaningful object-oriented example requires quite some overhead [Westfall, 2001]. Many textbook examples are, therefore, unnecessarily complex, not meaningful, or not even "truly" object-oriented [ACM-Forum, 2002; Hu, 2005]. During our "journey from Pascal to Java" we stumbled across these and other problems and made a few detours before realising that the object-oriented approach not only requires a new way of thinking, but also, a new way of teaching.

There are more factors contributing to a student's success or failure than the course material and how a course is taught. We have observed that our students as a group are less motivated and not as well prepared[3] compared to a few

[3] This problem has been observed by most math, science and engineering programs.

years ago. Attendance rates at exams, lectures and other scheduled events have decreased. Reading assignments are neglected to a high degree and mandatory assignments are submitted late. We introduced Supplemental Instruction (SI) [Arendale, 1997] to increase student activity and thereby improve course outcome. SI is a non-mandatory part of the course. In some course offerings, only half of the students participated in the SI programme while in other the majority of students participated. In all course offerings, students participating in SI have a higher attendance rate at exams and also get higher grades on average than the group of students not attending SI [Nordström and Kallin Westin, 2006].

The remainder of this chapter is organised as follows. First, we briefly explain the motives behind the principles we used for designing our new course, and how these were actually implemented. In section 4, we evaluate how well this new course worked with respect to our original principles. Section 5 summarises the lessons we have learned since we made the transition to the object-oriented paradigm in 1998. In sections 6 and 7, we discuss external factors affecting student performance and related work, respectively. The chapter concludes with a summary of our experience.

2 Principles for Course Design

Prior to our transition, we introduced object-oriented concepts in our data types and algorithms[4] course, following the introductory programming course. However, this was not sufficient to enable students to effectively use the object-oriented paradigm. Most students perceived object-orientation as a simple extension to imperative programming. They did not realise that object-oriented programs are conceptually different from strictly imperative ones and that using object-oriented syntax does not automatically lead to object-oriented programs.

When switching to the object-oriented paradigm in our introductory programming course in 1998, we only made minor changes to our traditional course design. Initially, our students did very well on this course, but we soon realised that their ability to develop code true to the object-oriented paradigm was not satisfactory. After several course offerings with unsatisfactory learning outcomes, we decided to develop a "truly" objects-early approach. When proficiency in a certain paradigm is the major learning goal of a course, it seemed sensible to start with that paradigm as early as possible [Bruce, 2005; Bergin, 2000b].

To support the design of such a course we developed a list of principles, to guide course development. These principles were either based on our teaching experience [Bacvanski and Börstler, 1997; Börstler et al., 2002; Börstler and Sharp, 2003; Kallin Westin and Nordström, 2003, 2004] or the collected advice and experience from the literature in computer science education (see e.g., [Bruce, 2005; Westfall, 2001; ACM-Forum, 2002; Guzdial, 1995; Holland et al., 1997; Kölling and Rosenberg, 2001; Kölling, 2003; Turk, 1997; and *Using BlueJ to Introduce Programming* by Kölling]).

[4] More or less similar to a CS2 course.

2.1 High-Level Goals

No magic (P1). We must provide a correct and consistent frame of reference, so that the students always can make sense of new material. The students must be able to associate the new material with something familiar or wellknown. The succession of learning units and topics must be carefully worked out. The frame of reference must be refined or extended accordingly. The current frame of reference should always be sufficient to understand new material and validly explain what is going on [Zull, 2002]. Everything requiring a comment like "don't worry now, you'll understand later," must be revised or delayed. Language specific complexities should be hidden until students are sufficiently mature to understand the underlying language design issues.

Students will always try to make sense of new material. If we cannot provide them with a correct and consistent frame of reference, they might construct invalid explanations by themselves. This can easily lead to persistent misconceptions about object technology and programming [Holland et al., 1997; Börstler, 2005; Clancey, 2004; Ragonis and Ben-Ari, 2005].

Objects from the very beginning (P2). Everything should build on the notion of objects, since they are at the very heart of object-orientation. Therefore, objects should be introduced in the very first lecture. The earlier we start with the most important concept, the more often we can reinforce it and the more time we give students to fully understand it.

General concepts favoured over language specific realisations (P3). Learning units should be based on the teaching and learning of general object-oriented concepts. Although the mastery of a particular programming language is an important learning goal, it is secondary to the understanding of the underlying concepts. Focusing on concepts does not necessarily mean to move strictly from concept to syntax for each new topic. It is, however, important to stress fundamental principles and techniques and not the elements of a particular language. This can, for example, be achieved by means of moving from concrete to abstract as proposed in *CS1: Getting Started* by Caspersen and Christensen.

No exceptions to general rules (P4). By general rules, we not only mean the definitions that constitute the object-oriented paradigm[5], but also design and programming guidelines and all the other pieces of advice we provide to our students. 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. Concepts must never be introduced or be reinforced by using flawed examples (see also P6).

2.2 Tools

OOA&D early (P5). It is necessary to provide students with simple tools to approach a problem systematically and to evaluate alternative solutions before

[5] Like for example "objects are instances of classes with state, behaviour and identity," or "in object-oriented programs, problems are solved by objects sending messages to each other."

starting to code. Early OOA&D conveys to the students that responsibilities are distributed amongst the objects that solve a problem [Börstler, 2005; Andrianoff and Levine, 2002.]

Exemplary examples (P6). 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) [Holland et al., 1997; Nordström, 2007]. Consequently, examples should be non-trivial and involve multiple classes. All examples should be made available for experimentation, e.g., by making the source code available for download from the course web page.

Easy-to-use tools (P7). Students should be provided with tools that support object-oriented thinking. The tools must be easy to learn and easy to use. Tool usage must add as little cognitive load as possible to the students' tasks. Usability should be favoured over any "bells and whistles."

2.3 Pragmatics

Hands-on (P8). Programming is a skill that must be "trained." Topics should be reinforced by means of practical exercises. Lectures should be followed by supervised in-lab sessions. For each session, step-by-step instructions and exemplary examples should be provided.

Less "from scratch" development (P9). "Reading before modifying before coding." All software development takes place in context (see also *Using BlueJ to Introduce Programming* by Kölling). Reuse is an important aspect of the object-oriented paradigm and should be emphasised early. To be able to read, to understand, and to modify existing code is, therefore, as important as developing understandable code.

Alternative forms of examination (P10). Assessment should support learning. It is very important to evaluate actual programming skills as well as conceptual understanding. Furthermore, assessment should not be separated from teaching. For example, peer evaluation or peer marking can call the students' attention to alternative ways of solving certain problems. It is important to realise that there rarely is a single, correct solution to a problem.

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

To summarise, our main goal was to follow an object-oriented approach in a true and consequent way. In addition, we wanted to provide our students with easy-to-use tools supporting "object thinking" and the systematic development of proper object-oriented code (see also *Model-Driven Progamming* by Bennedsen and Caspersen).

3 Implementation

The first course, based on these principles, was offered in spring 2001. After a case study in summer 2001 [Börstler et al., 2002], we have refined our teaching

approach and successively implemented it in all our introductory programming courses[6]. Some of the principles are very difficult to implement, or even in conflict with each other. In particular the principles *No magic (P1)* and *Exemplary examples (P6)* still cause a lot of work.

From an organisational point of view, we made four major changes to our original course as follows:

1. We introduced BlueJ [BlueJ, 2007], a programming environment particularly designed for the teaching and learning of object-oriented programming to novices (P7) [Kölling and Rosenberg, 2001; Kölling et al., 2003]
2. We introduced CRC-cards, a simple informal tool for collaborative object-oriented modelling (P5, P7) [Beck and Cunningham, 1989]. The strength of the CRC-card approach lies in its associated scenario role-play activities [Börstler, 2005; Andrianoff and Levine, 2002]. During the role-plays the students explore hypothetical, but concrete situations of system usage (scenarios). They enact the objects in the model, much like actors following a script when playing the characters in play. This supports "object thinking" and helps the students to develop a mental model of the workings of an object-oriented program (P1-P4) [Börstler and Schulte, 2005].
3. To accommodate for more practical training (P8), we substituted our traditional lecture room exercises by guided, hands-on exercises in computer-labs.
4. The traditional pen and paper exam was split into a shorter one, half way through the course, and a computer-based exam was used at the end to test actual programming skills (P10).

In addition to these changes, we started to offer Supplemental Instruction (SI) [Kallin Westin and Nordström, 2003] to improve students' study skills and to make them more active participants in the course. SI is targeted towards historically difficult classes to help students master content while developing and integrating strategies for learning and studying [Arendale, 1997]. This is done through sessions guided by a model-student, the SI leader.

A major difference between SI and other forms of collaborative learning is the role of the SI leader. Rather than forming study cluster groups and then releasing them in an unsupervised environment, the SI leader is present to keep the group on task with the content material and to model appropriate learning strategies that the other students can adopt and use in the present course, as well as other ones in future academic work.

Since the "roll out" of our teaching approach, we have made several changes to our introductory programming courses (see Figure 2 for an overview). For example, we have postponed graphics and event handling to a newly developed advanced programming course. We have also slightly adapted our course for non-CS majors. However, we are still faithful to all our principles.

[6] We offer introductory courses in object-oriented programming for three different technical degree programs.

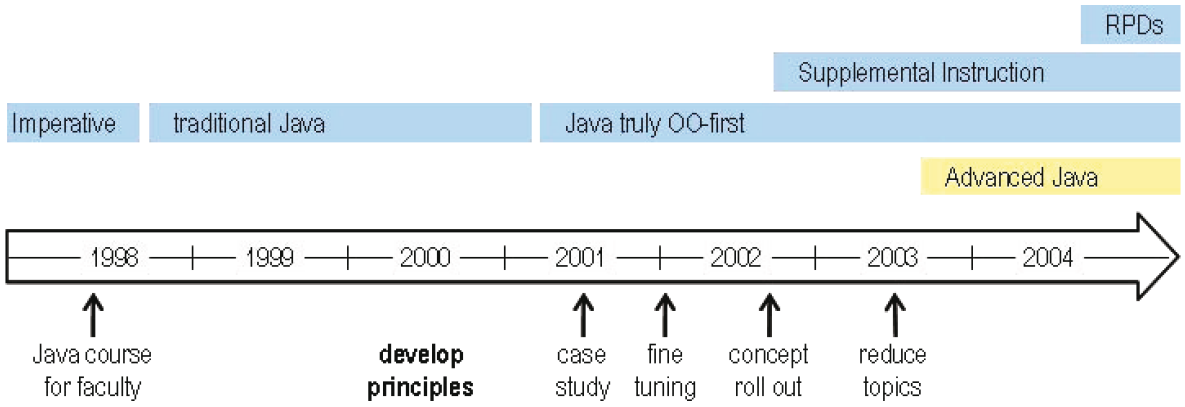


Fig. 2. Major steps in the evolution of the introductory programming course

4 Evaluation

In this section, we restrict our discussion to an evaluation of our principles as defined in section 2. Overall, we conclude that changing to a "truly" object-oriented approach according to our principles worked well. However, there are many factors not directly related to the teaching and learning of object-oriented programming itself that affect course design and outcome. Many important factors are difficult to control like prerequisites and attitudes of the students entering our programs, for example. This will be discussed in section 6.

4.1 High-Level Goals

In common for all high-level goals (P1-P4) is the urge to be "truly faithful" to the object-oriented approach. This means to avoid concepts or examples that seem to question or even contradict the idea of object-orientation, such as objects without meaningful state or behaviour, excessive use of static methods and public attributes, Singletons[7], etc. [Westfall, 2001; Hu, 2005]. To be "truly faithful" also means to strive for meaningful objects in realistic contexts.

No magic (P1). Our ambition has always been to use examples and contexts not only simple enough for the students to understand, but that also emphasises the object-oriented paradigm. BlueJ is an excellent tool for this since it allows teachers and students to concentrate on the object-oriented aspects instead of dealing with editors, configuration files, compilers, etc. BlueJ achieves this by visually representing classes and objects and manipulating them directly using its graphical user interface (see figure 3). Unfortunately, this approach has some limitations that can generate misconceptions that can be harmful and great care has to be taken to avoid them. A short example will illustrate the problem.

Since the very beginning we have used an example with geometrical shapes supplied with the BlueJ environment [Barnes and Kölling, 2003; and *Using BlueJ to Introduce Programming* by Kölling]. In BlueJ, objects are represented by red

[7] A Singleton is a class with one single instance only.

blocks in the object bench (see for example `c: Circle` in figure 3). Actually, to be more precise, these red blocks represent object references and not the objects themselves. This is a small, but important difference as explained below.

One can send messages to objects in the object bench by right clicking them. This will display a menu with the methods defined for this object. When selecting `makeVisible()`, a graphical surface is created ("automagically") and a representation of `c` is drawn on it (see Window `BlueJ Shapes Demo` in Figure 3). Whenever the state of an object is changed, its representation is changed or animated accordingly. This gives immediate feedback and helps students to understand the difference between classes and objects. On the other hand, it blurs the difference between the objects themselves and their references. Furthermore, the details of the graphics are quite involved and too complicated to understand ("magic") for a novice.

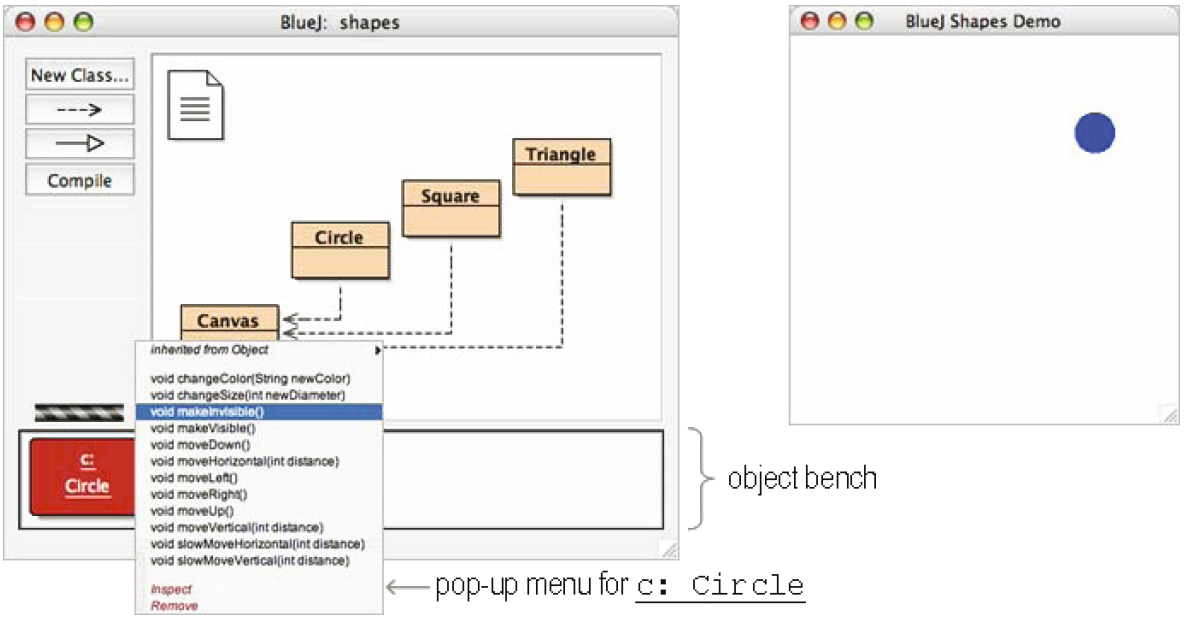


Fig. 3. Screenshot of the Shapes example

Another problem with the Shapes example is the cognitive difficulty to differ between the visual representation of an object (the circle in window `BlueJ Shapes Demo`) and the object itself, which actually cannot be seen. If, for instance, the reference to the object (the red block in the object bench) is removed, nothing happens in the drawing. This is puzzling for the inexperienced because the coloured dots on the canvas are mistaken for the object itself! Misconceptions like this are very hard to deal with. Other examples of difficulties are discussed in [Ragonis and Ben-Ari, 2005].

This example shows how difficult it is to create assignments early in the course, without (unintentionally) introducing magic or material not taught yet.

Objects from the very beginning (P2). To make the students immediately acquainted with the idea of objects, we use a kind of interactive exercise the first

lecture [Andrianoff and Levine, 2002; Bergin, 2000a]. Without previous explanation, the students are asked to discuss in general terms, something that needs to be modelled like, for example, a ticket machine or an employee. During the discussion the lecturer collects specific and general characteristics and behaviours on the whiteboard. At the end of the lecture, these things are pointed out as "properties" belonging to a single object or a class, respectively.

Nevertheless, it is difficult to convey to the students that they are working with an isolated "component" in a larger program, instead of a whole program. Many students, particularly those with previous programming experience, find it quite frustrating that there is no "program" to execute like they are used to. They seem to have difficulties focusing on the properties and responsibilities of objects without controlling its context at the same time [Guzdial, 1995].

General concepts favoured over language specific realisations (P3). The learned programming and problem solving should be transferable to other programming languages. It is important, therefore, to focus on general concepts. We try to highlight general concepts, knowledge and skills and to avoid language specific details and idiosyncrasies.

This has resulted in using elementary UML-notation throughout the course, instead of some kind of simplified temporary notation. However, we do not explicitly introduce UML. We just use its most intuitive parts consistently.

Furthermore, semi-formal syntax-diagrams are used. This makes it much easier to talk about the syntax, semantics and pragmatics of programming languages. Information hiding is also stressed as a general (design) concept and the usefulness of Boolean variables to formulate easy to read expressions.

On the other hand, topics like anonymous objects and classes are not discussed. These concepts require a thorough understanding of object-orientation and are saved for later courses. We try to avoid language idiosyncrasies as long as possible in particular shortcuts like ++, +=, ?:, etc. and forced returns out of for-loops and methods. They just make code harder to read and, therefore, their use is actually discouraged.

No exceptions to general rules (P4). It is important to be consistent with the frame of reference we provide to our students (see P1 in section 2.1). Students will hopefully adopt what the teachers present to them eventually. It is important, therefore, not to misguide them (see also P6). We must never present any material, explanation or example that we might reject as an answer or solution from a student.

Unfortunately, Java courseware in particular is littered with examples that contradict the "rules" or "styles" we want our students to adopt. The concept of objects, for example, should not be exemplified by using Java strings. In Java, `String` objects cannot be modified and do not posses all the characteristics we require from proper objects [Thimbleby, 1999]. Since `Math` has only `static` methods and there are no objects of this class type, its use should be postponed until the students have a firm understanding of the concepts class and object. The `main` method is an atypical method since there is no object it belongs to.

Thus, the method is never invoked explicitly and its parameters seem to be supplied by magic forces (see also P1).

4.2 Tools

OOA&D early (P5). The purpose of this principle is twofold: showing the students a systematic way to develop a solution for a given problem, and providing them with a tool to reason about object-oriented solutions without the need of actual code.

By using CRC-cards [Beck and Cunningham, 1989], we can do both. The object-as-person metaphor helps students with "object thinking" and to develop a conceptual model of the inner workings of an object-oriented program [West, 2004; Börstler and Schulte, 2005]. Another advantage of this approach is that it does not require any prerequisite knowledge.

A problem noted in [Bellin and Simone, 1997] and described in detail in [Börstler, 2005] is that CRC-cards are used as surrogates for classes (in the modelling activities) as well as for objects (in the role-play activities). This conflicts with the *No exception* principle (P4) and can easily confuse novices.

To address these problems, we enact a live CRC session in front of the class to give the students a feeling for the dynamics of such a session. In addition to that, we have developed so-called Role-Play Diagrams (RPDs) to support and document the role-play activities [Börstler, 2004]. RPDs combine elements from UML object and collaboration diagrams [OMG, 2003]. However, the notation is informaland much less extensive. In RPDs, we use specific object cards to denote objects and thereby, avoid the double role of the CRC-cards. Although the enhanced "method" is more complicated than the original one, the students have fewer problems using it. The RPDs also provide an excellent documentation of the role-play. To give the students some practical experience in CRC-card role-playing, we schedule two CRC exercises where the students develop designs for small problems. One of these designs is later implemented as an assignment.

Exemplary examples (P6). As discussed in *No magic* (P1), it has turned out to be difficult to find or to develop suitable problems and examples for the initial introduction of objects. The range of concepts and syntactical elements known to the students is still very limited. Examples should also be small and to the point, so that students do not lose sight of the concept exemplified. This limits the degree of freedom for defining "realistic" objects. For example, what would constitute a reasonable context illustrating the concepts of loops? What kind of object would have such behaviour? Immediately the example grows to justify the use of a simple construct and tends to conceal the small component it was intended to show.

Another problematic example is the usage of Singleton classes, like the popular Pig-Latin translator [Nordström, 2007]. One might ask whether it is reasonable to have a class `PigLatinTranslator`? How many objects of this class would anyone need? Singleton classes do probably not qualify as good examples. The main idea behind classes is instantiating as many objects as necessary. Singletons

are special cases, i.e. an exception to the general rules (c.f. P4). Their treatment should, therefore, be delayed to more advanced courses.

Many examples use print statements to present some result. This is not a representative way to illustrate objects and classes. Usually, results are returned and used by other objects. In an object-oriented program, objects communicate to fulfil a task. Objects that use printing to present results are rarely useful in other contexts. Students are not able to reuse such examples as prototypes or templates to solve more general problems.

Exploiting the "naturalness" of the object-oriented approach can also be difficult. Object-oriented models of real-life objects might have behaviours and responsibilities their real life counterparts never would or could have. Therefore, it is very important to make a distinction between the model and the entity being modelled. A typical example of this could be the model of an employee in an economy system for a company. The model of the employee could have the responsibility to know its salary, the number of remaining days of vacation and so on. This is conflicting to how things are in real life. No company would rely on their employees to be the only source of information for the payment of salaries. So, how could it be possible for the inexperienced designer to foresee this responsibility in the model?

Easy-to-use tools (P7). Some of the advantages of BlueJ turned out to be disadvantages for the students (initially). The interaction with entities is done by right-clicking a class or an object. The problem for the student is to understand the equivalence of right-clicking and generating the same action in code. Another problem is to realise the difference between classes and objects [Ragonis and Ben-Ari, 2005]. However, as the students continue to practise their skills using BlueJ they realise the strengths of this simple, but powerful, interaction with objects.

The ability to write code must not depend on the tools we provide to our students. Students must not be "locked" into BlueJ for example. This is also highlighted by the BlueJ developers (see also *Using BlueJ to Introduce Programming* by Kölling). They should develop and run at least one complete application outside BlueJ. Although experienced students tend to dislike BlueJ, we think they should be encouraged to at least try it. They might very well get some new insights into the object-oriented paradigm.

4.3 Pragmatics

Hands-on (P8). The initial idea of guided in-lab sessions directly following the lectures did not work as expected. The students complained about lack of time to think about the new material before using it. Most students actually had difficulties applying the ideas presented. They merely consumed the presentation at the lecture.

In later years, we have thus rescheduled the lab sessions. We still have the same number of hands-on sessions, but they are no longer scheduled on the same day as the corresponding lectures. We also developed very detailed guides

to make sure students succeed with initial tasks and so they can gain some confidence in working with the environment. Too detailed guidelines or fill-in-the blanks exercises, however, can be counter-effective. Students might be enabled to perform successfully without actually understanding their answers and activities. Students and teachers as well might get a faulty feeling of mastery of the subject.

Less "from scratch" development (P9). The practise of reading and manipulating existing code before actually writing own code turned out to be a major problem for our students. Inexperienced students acquired a passive practice and had difficulties writing complete programs on their own. It is important then to train some programming from scratch. Experienced students, on the other hand, often want to have full control over their programs and might reject "foreign" code [Guzdial, 1995]. However, code reuse is a crucial practice that requires code reading and understanding and needs to be trained as well.

Alternative forms of examination (P10). The content of the course is initially focused on object-oriented concepts, while the second half is heavier on actual problem solving and programming. To reinforce the need to work with and to understand basic concepts early on, we divided the examination into two parts. Halfway through the course a written (theoretical), closed-book test is given and at the very end, a practical problem solving and programming test is given. The results of the two tests are added and graded as one. In addition to this, the students have mandatory assignments and some of them orally assessed. The idea of splitting the examination into two tests with rather different focus was appreciated by the students. Furthermore, the exam results better reflect student skills than a single pen-and-paper test.

Emphasise the limitations of computers (P11). This principle had its origin in the numerical tradition of our department. We make students aware of problems and limitations in data representation and how these can lead to erroneous computations. We emphasise this by discussing examples leading to unexpected results in logically correct programs.

5 Lessons Learned

In this section, we briefly summarise the practices that worked particularly well for us. We have grouped them together into recommendations to make them easily accessible to the reader.

Teach "object thinking" and modelling explicitly.

- Start the first lecture with a modelling or role-play activity (no syntax involved). Students can be asked then to describe (model) an employee or a ticket-machine to illustrate the basic object properties (state, behaviour and identity).
- Introduce CRC-cards and scenario role-plays. This provides students with a framework to think in terms of (active) objects and their responsibilities. Furthermore, it teaches them basic modelling/ problem solving skills.

- Introduce role-play diagrams so that students easily can track and document scenario role-plays. This also helps to prevent some problems inherent in the original CRC approach [Börstler, 2005; Börstler and Schulte, 2005].

Schedule guided and supervised lab activities. Programming is a skill that needs to be trained extensively. Students should visit the labs as frequently as possible and receive immediate help when getting "stuck."

- Reduce the number of traditional lectures and introduce supervised lab sessions instead. Guide students through practical exercises in the labs. We provide for example step-by-step guides, including reflective questions, which the students have to work through. Lecturers and teaching assistants should always be present to discuss and resolve problems.
- As much as possible, move supervising time from office rooms to the computer labs to force students to visit the labs to ask questions.

Use and utilise a suitable programming environment. The environment must be easy to use and to support the object-oriented paradigm. However, it is also important to show how programs are developed and executed outside such an environment. We have used BlueJ [BlueJ, 2007] successfully since 2001.

Examine the "right" things. It is important to assess actual and individual programming skills in addition to conceptual and syntactical knowledge. This can be done, for example, by practical computer based tests (problem-solving and programming) and individual oral demonstrations of mandatory assignments.

There is no course design that fits all target groups. Different groups of students need different types or flavours of courses. It is important to be sensitive to changes in the field as well as the context and the environment of a course [Forte and Guzdial, 2005; Jenkins and Davy, 2000; and *Using On-line Tutorials in Introductory IT Courses* by Thomsen]. Our principles have been a useful guideline to us when adopting the course to different student groups. The principles make sure that the core of the course is the same and taught in roughly the same way, regardless of lecturer and student group.

Do not lull students and teachers into false security. Fill-in-the-blanks guides and exercises can give a faulty feeling of students' subject mastery. Too much help or undemanding tasks can lead to mechanical answering. If no reflection or second thought is necessary, then students can successfully complete such exercises without learning anything. Also, teacher expectations about what the students really have learnt might be too high.

Good examples are crucial, but very hard to develop. Truly object-oriented examples are very difficult to find or to develop. Educators should resist constructing examples "on-the-fly" (for example to exemplify a specific feature), since they rarely will follow principles P1, P4 and P6.

- Programming in a true object-oriented style often leads to overly (unnecessary) complex examples, due to the additional layers of abstraction imposed by the paradigm. This can be frustrating to students since they cannot understand why the different abstraction layers are necessary (e.g., "Why should I do it like that, it's easier and faster to read the information directly from the database"). It is a challenge for the teacher to explain that optimization is secondary to a good object-oriented design. Our main goal is to devise a good solution that fulfils certain quality criteria and not to simply make it work somehow. Students are not mature enough to differ between optimizations and proper design.
- Although often claimed, there is no 1:1 relationship between real-world objects and their corresponding software abstractions. A physical book for example is removed from the library, when it is checked out. A book object in a (software) model, however, stays in the library and is only marked as "on loan." Furthermore, in a "real world" library, we would never make the borrowers responsible for keeping track of their unpaid overdue fines. In a (software) model however, this might be a good design choice since trust is no issue there.

Keep students active. Data collected during the SI-projects shows without a doubt that student attendance and activity correlate with course results [Arendale, 1997; Kallin Westin and Nordström, 2003, 2004.] Mandatory in-lab exercises and a two-stage examination keep the students alert and active from the start. SI gives the students opportunities to work with the course material in a structured way and helps them to recognise the strength of collaboration. After introducing SI, the attendance rate on the examination rose from 80 percent to above 95 percent (see Section 6, in particular Figure 4).

6 Discussion

When analysing student performance over the years (see Figures 4 - 6) our case for a "truly-objects-first" approach does not look convincing. However, there are also external factors affecting student performance. These factors have lead to considerable changes in the student population in recent years.

Assessment consists of mandatory assignments, a pen-and-paper test and a computer-based test (see P10 in Section 4.3). In Figure 4, the passing rate after the first opportunity to finish the course is shown as a solid line. Java was introduced in 1998 and our "truly" object-first approach was introduced in 2002 (see Figure 2). The dashed line in Figure 4 shows the attendance rate on the exam (i.e. the proportion of students submitting at least one mandatory assignment or attempting at least one test). SI was introduced in 2002 to raise attendance rates and it seems to have an effect[8]. Participation in SI correlates with overall student

[8] In 2001, the seemingly high attendance rate was due to an examination system where handing in assignments (not necessary correct ones) gave credit points on the final exam. The numbers for 2001 in figures 4 - 6 must be seen, therefore, as statistical outliers.

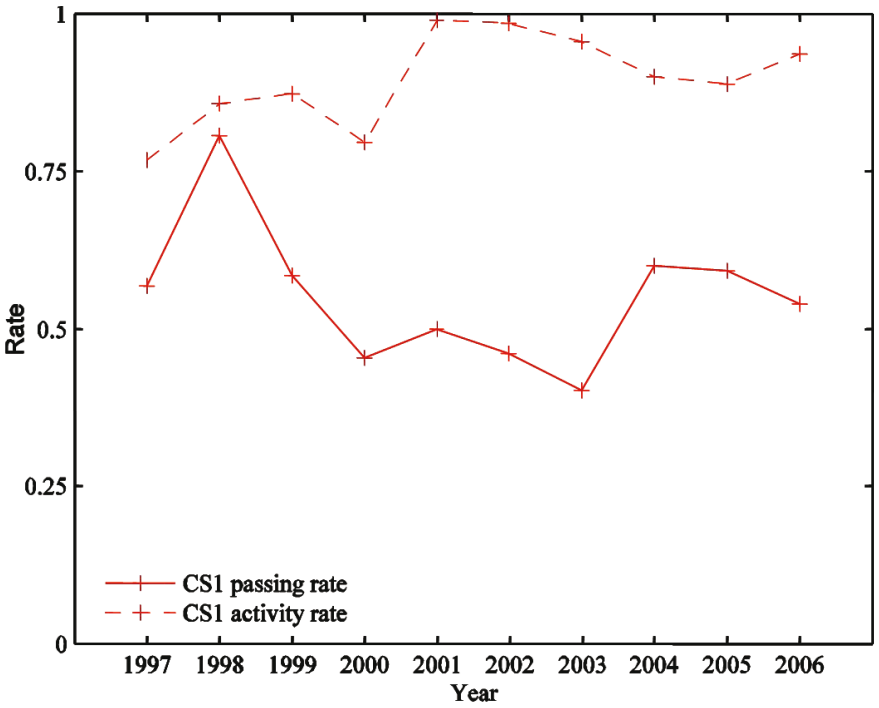


Fig. 4. Performance data for CS majors on our introductory programming course. The solid line represents the passing rate after the first opportunity to finish the course. The dashed line represents the proportion of students submitting at least one mandatory assignment or attempting at least one test.

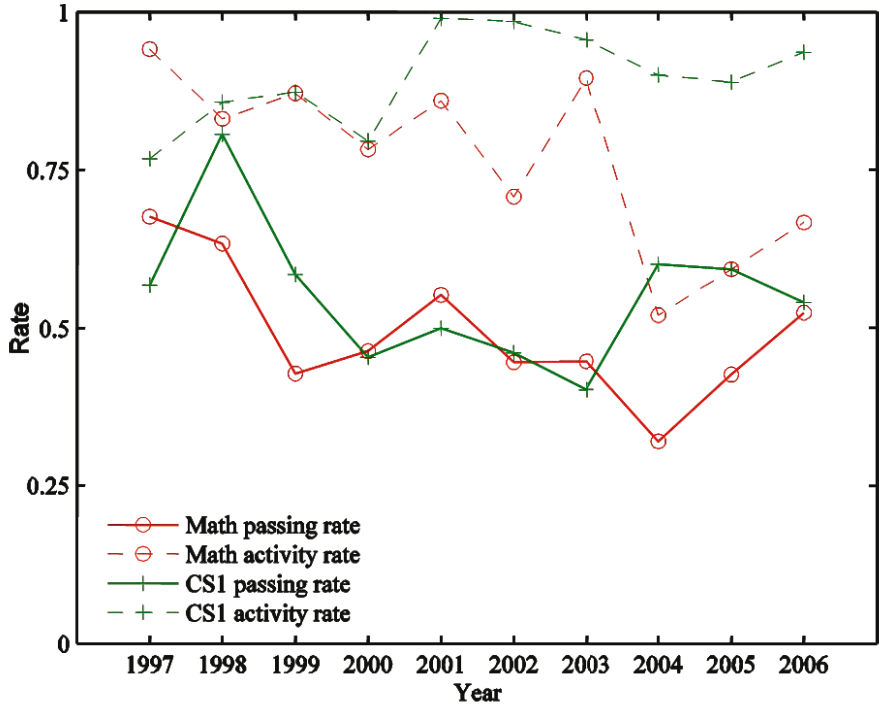


Fig. 5. Performance data for CS majors on the discrete mathematics course, compared to our introductory programming course (cf. figure 4)

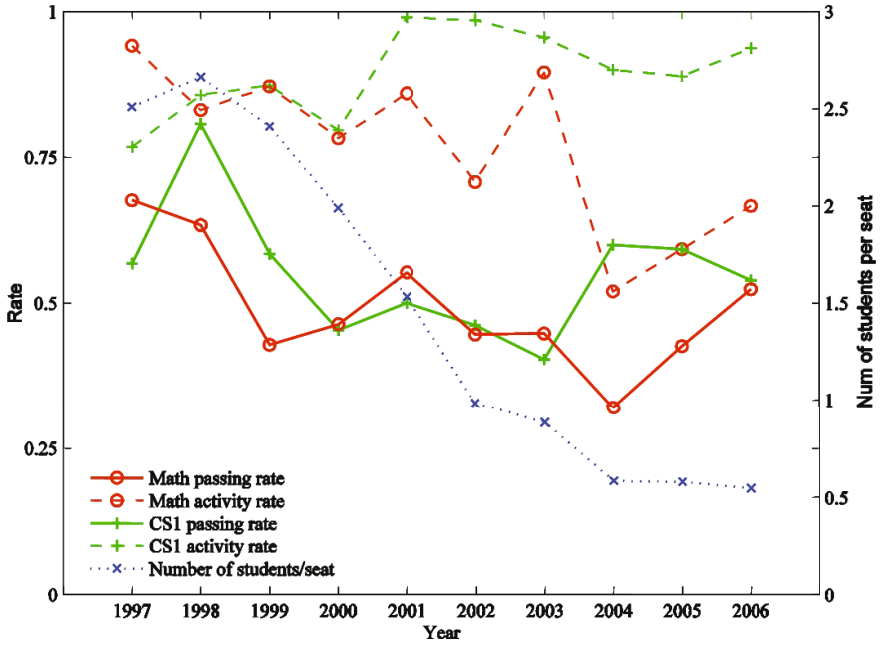


Fig. 6. The number of students per seat on our programme is shown as a dotted line added to Figure 5

performance. Unfortunately, the weakest students seem not to be motivated to participate in SI. An investigation of the students with severe problems in keeping up with the pace of the curricula of the programme showed that a vast majority had not attended SI at all, or only tried it a few times.

Another factor is that knowledge and skills in mathematics have been decreasing in general [Högskoleverket, 1999; Helenius and Tengstrand, 2005]. It is believed that mathematical ability is strongly connected to performance in introductory programming [Denning, 2004]. Skills, like (array) indexing and creating series of numbers seem to be more of a problem nowadays. Students also have a weak understanding of functions, in particular, their parameters and return (computed) values. This lack of understanding might result in the assumption that the only possible way to get something out of a method is by printing a string to the screen. We strongly believe this contributes to the lower passing rates, especially for mandatory assignments.

Our CS majors take a course in discrete mathematics in the same term as their introductory programming course. In Figure 5, we can see that the students also have problems in the discrete mathematics course. Attendance rates are even lower in discrete mathematics where students only have a single traditional exam at the end of the course. However, in the introductory programming course, a student needs to attend only one of the three exam parts to be counted as "active". This might falsely indicate high attendance rates.

The dotted line in Figure 6 represents the number of students per seat. In Sweden, each programme has a fixed number of student seats available. The applications to IT-related programmes have severely suffered from the turbulent situation within the IT business. Practically, this means that all students applying are admitted as long as they fulfil the basic prerequisites. Historically, we

have had 2 to 3 applications for each seat available, resulting in a higher grade average for the students admitted.

A further factor is a shift in motivation among novices. In a study we performed in 1994, the main reason for students applying for our programme was an interest in the subject itself (or in mathematics). In a later study, the motivation had shifted to "want high salary," "want to be a civil engineer," and other reasons not connected to the subject or the programme itself. Thus, students' interest in computing science is far from obvious [Kallin Westin and Nordström, 2001, 2003; Eliasson et al., 2006a,b]. Similar trends are reported internationally [Forte and Guzdial, 2005; Jenkins and Davy, 2000].

7 Related Work

There have been several attempts to explain why students are having difficulties in their first Java course. Three common explanations are the following:

- The students can program, they are just having problems with the design part [McCracken et al., 2001].
- We are not teaching object-orientation the correct way, we need to teach the subject in a pure, object-oriented way [Bergin, 2000a; Kölling and Rosenberg, 2001.]
- Java has so many special cases, like `public static void main` and string handling so that it becomes difficult for the students to remember and to understand all the special cases [Bruce et al., 2005].

In a multi-national cross-university study, [McCracken et al., 2001] investigated how well students actually could program. They proposed a list of five steps that students should be able to follow successfully after passing CS1.

- Abstract the problem from its description.
- Generate sub-problems.
- Transform sub-problems into sub-solutions.
- Recompose the sub-solutions into a working program.
- Evaluate and iterate.

The results from this study were disappointing. The students' programming skills were at a much lower level than expected. Somewhat surprising, the most difficult part seemed to be the first step (e.g., to abstract the problem).

[Lister, 2004] followed up on these results and investigated students' ability to read, to understand and to modify existing code. Here, the results were disappointing also. A surprisingly large proportion of the students had difficulties completing even the most basic tasks. It seems that students not only have problems with the abstraction step, they also have problems with the more basic task of reading and understanding code.

[Lister, 2004] also investigated the annotations students made while solving the problems. In general, it turns out that students who carefully trace executions are more likely to provide correct answers than those who do not. However,

there are considerable differences between universities [McCartney et al., 2005]. Students from some universities used annotations (traces) to a very high degree while others did not.

Considering the scope of this book it is interesting to note that the two universities with the least annotations are in Sweden and Denmark. Despite the low annotation level, these students performed on average compared to the students from other universities. Whether this is a coincidence or due to differences in object-oriented programming education needs to be further investigated.

8 Summary and Conclusions

The transition to object-orientation is not easy. It is not sufficient to simply change the language of instruction in an otherwise traditional introductory programming course. The strong relationships between basic, objected-oriented concepts constitute a high threshold to the learning and teaching of programming. Considerable changes to the course design are necessary to convey to the students the real power of the object-oriented approach.

We have presented and evaluated a set of eleven principles for course design that have helped us to stay on track in our efforts continuously to improve our introductory programming course. We have seen several factors that influence the results of an introductory programming course apart from the course itself. Attendance rates drop on all parts of the course and many students seem to think that knowledge can be acquired passively.

It is our firm believe that it is necessary to be faithful to the object-oriented approach. Tools that help students to "think in objects" are very important for successfully teaching basic object-oriented concepts. We must provide our students with a consistent frame of reference. This frame of reference will change with the knowledge and skills the students acquire. However, its core (i.e., the basic rules) should not be constantly contradicted by our own exercises and examples.

Paper II

Heuristics for Designing Object-Oriented Examples for Novices

Heuristics for Designing Object-Oriented Examples for Novices

MARIE NORDSTRÖM and JÜRGEN BÖRSTLER
Umeå University

Research shows that examples play an important role for cognitive skill acquisition, and students as well as teachers rank examples as important resources for learning to program. Students use examples as templates for their work. Examples must therefore be consistent with the principles and rules of the topics we are teaching.

Despite many generally accepted object oriented principles, guidelines and rules, textbook examples are not always consistent with those characteristics. How can we convey the idea of object orientation, using examples showing "'anti"'-object oriented properties?

Based on key concepts and design principles, we present a number of heuristics for the design of object oriented examples for novices. We argue that examples adhering to these heuristics are of higher object oriented quality than examples that contradict them.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Principles, Guidelines, Examples

Additional Key Words and Phrases: Example programs, object-orientation, design, quality

1. INTRODUCTION

Examples play an important role in learning, both teachers and learners consider them to be the main learning tool [Lahtinen et al. 2005]. Research also shows that the majority of learning takes place in situations where students engage in problem solving tasks [Carbone et al. 2001]. In a recent survey of pedagogical aspects of programming, Caspersen concludes that examples are crucial:

> *Studies of students in a variety of instructional situations have shown that students prefer learning from examples rather than learning from other forms of instruction .... Students learn more from studying examples than from solving the same problems themselves ....* [Caspersen 2007, p. 27]

To be useful, examples must help a learner to draw conclusions and to make inferences and generalisations from the presented information [Chi et al. 1989; Pirolli and Anderson 1985]. Since examples do not distinguish incidental from essen-

Author's address: Marie Nordström, Department of Computing Science, Umeå University, SE-90187 Umeå, Sweden; email: marie@cs.umu.se.

tial, or even intended, properties, we argue that examples developed according to established practices and experience are a necessity for the example to promote (accurate) generalisation. Novices should be able to use examples to recognize patterns and distinguish an example's accidental surface properties from those that are structurally or conceptually important. By continuously exposing students to well-designed examples, important properties are reinforced. Students will eventually gain enough experience to recognize general patterns that help them telling apart "good" and "bad" designs.

Though largely debated, object orientation is commonly used for introducing problem solving and programming to novices. The strength of object orientation lies in the handling of complexity in the design of large-scale system, with high demands on maintenance, flexibility and reusability. The educational situation, however, is rather different and does not fit these strengths well. Introductory examples are often small. Furthermore, the design space for examples is restrained since many concepts and syntactical elements might not have been introduced yet.

In the following, we will use the term *small-scale* for the specific situation of introducing object orientation to novices. A small-scale example is a program, example, or exercise intended for novices in order to present or illustrate a certain concept or feature of object oriented problem solving and programming. Small-scale significantly limits the design space of examples. The size of examples, the repertoire of concepts and syntactical components, and the need to present concepts in isolation are limiting conditions. Furthermore, we have to support object-thinking, and we have to be careful with the context or problem domain we choose. Nevertheless, we argue that one has to be truthful to the paradigm chosen for introductory programming, otherwise novices might not recognize any patterns.

Based on commonly agreed upon object oriented principles, guidelines and rules, we propose a number of heuristics for the design of example programs. We argue that examples developed according to established practices and experience will lead to suitable role-models. The proposed heuristics address object oriented quality in example programs independent of any particular choice of instructional design or sequence of concept-introduction.

2. RELATED WORK

Various aspects of teaching object orientation to novices have been addressed by many excellent professionals and educators, but the quality of examples has not been discussed in a systematic way. Specific common example programs, like "`HelloWorld`", have been critically discussed for a long time [Westfall 2001] and there have been ongoing debates on the object-orientedness of this and similar examples [CACM 2002; Dodani 2003; CACM Forum 2005]. However, all of these discussions have focused on technicalities, rather than conceptual object oriented qualities of the examples. A recent study of the quality of example programs in common introductory programming textbooks shows that there is much room for improvements [Börstler et al. 2009; Börstler et al. 2010].

McConnell and Burhans [2002] examined how the coverage of basic concepts in programming textbooks has changed over time. They conclude that "there has been a trend of decreasing coverage for basic programming and subprogram concepts as

other more current material has been added". In general, it seems as we are focusing more on syntactical issues than on problem solving. This view is also supported by De Raadt et al. [2005], who examined 49 textbooks used in Australia and New Zealand according their compliance with the ACM/IEEE curriculum recommendations.

It has been argued that object orientation is a "natural" way for problem solving, but several studies question this claim [Guzdial 2008]. In particular, it seems that novices have more problems understanding a delegated control style than a centralised one [Du Bois et al. 2006], which is critical for understanding object oriented programs. Such difficulties in understanding program execution have also been studied by Ragonis and Ben-Ari [Ragonis and Ben-Ari 2005b]. They conclude that "[i]t is futile to expect that a teaching approach (like objects-first) or a pedagogical tool (like BlueJ) will be able to solve all problems that students have when learning a subject".

A common result in many studies is that novices have a hard time understanding the difference between class and object/instance (see for example [Eckerdal and Thuné 2005; Ragonis and Ben-Ari 2005a; Sanders et al. 2008]). Holland et al. [1997] discuss a number of misconceptions concerning the concept of an object and suggest examples and exercises to avoid them. Fleury [2000] shows examples of erroneous student-constructed rules that could be avoided by more carefully defined examples.

A coarse categorization of "harmful examples" is provided by Malan and Halland [2004]: *Examples that are too abstract*, *Examples that are too complex*, *Concepts applied inconsistently*, and *Examples undermining the concept introduced*. However, they do not discuss instructional guidelines to address these problems.

3. ESSENTIAL CONCEPTS AND PRINCIPLES OF OBJECT ORIENTATION

In [Nordström 2009], we have reviewed the literature to identify a small set of commonly accepted basic object oriented concepts (e.g. [ACM 2008; Armstrong 2006; Henderson-Sellers and Edwards 1994; Kramer 2007]). We have furthermore reviewed design principles, guidelines, rules, and metrics for object oriented design (e.g. [Bloch 2001; Fowler et al. 1999; Gamma et al. 1995; Gibbon 1997; Grotehen 2001; Martin 2003; Riel 1996]). These sources, together with the literature on student problems and misconceptions (see Section 2), as well as our own and other educators' teaching experiences have formed the input for defining a small set of heuristics for defining small-scale example programs.

Figure 1 gives an overview over the types of sources we have considered in this work.

Within the small-scale context it is often difficult to follow all "good advise". In a perfect world, we should for example always put the `main`-method into a separate class (to emphasize proper encapsulation) and feature multiple instances of at least one class (to emphasize the difference between classes and objects). In an educational context, however, we prefer small and concise examples, very often for purely pragmatical reasons (to make them fit on a page for example). If we want it or not, our examples will be used as role-models by our students. If we do not give them example programs of high quality, we cannot expect high quality

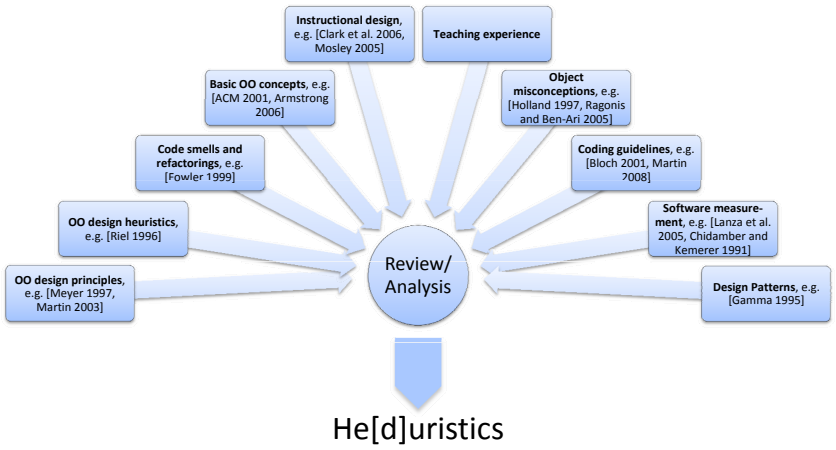


Fig. 1. Types of sources used as input for defining educational design heuristics.

code in return.

4. HE[D]URISTICS

The intention of our He[d]uristics is to support the design of exemplary examples. The He[d]uristics are targeted towards more general design characteristics. Specific detailed guidelines, like keeping all attributes private, are not stated explicitly. However, they are implicitly included in the more general guidelines. This made it possible to define a small, but powerful set of "rules" that can be easily handled:

(1) Model Reasonable Abstractions
(2) Model Reasonable Behaviour
(3) Emphasize Client View
(4) Favour Composition over Inheritance
(5) Use Exemplary Objects Only
(6) Make Inheritance Reflect Structural Relationships

We want to stress that the He[d]uristics are independent of a particular pedagogy (objects first/late, order of concepts, ...), language, or environment.

4.1 Model Reasonable Abstractions

Abstractions are at the heart of object orientation. An abstraction focuses on the essential properties of objects from the perspective of a particular viewer or "user" and suppresses accidental, internal details [Booch 1994]. A good abstraction makes the objects of interest easier to handle, mentally, since we do not need to constantly think of all the details that might complicate their handling.

An abstraction should also be plausible, both from a software perspective as well as from an educational perspective. In particular, it must be plausible seen through the eyes of a novice. Concept formation is driven by cognitive economy and inference [Rosch 1999]. A good classification provides a maximum amount of

information about (the properties of) a particular instance with the least cognitive effort. Concept formation is also influenced by the knowledge of and experience with the things that are categorized. A novice's concept formation will therefore be very different from an experienced software developer or domain expert. Which properties are perceived as meaningful or not can therefore be very different [Rosch 1999].

In an educational context, we often make simplifications to decrease a problem's size and complexity. These simplifications should, however, never lead to non-intuitive or artificial classes and objects. It must be possible to imagine a client[1] using the objects we are modelling and the objects must model some meaningful entity in the problem domain.

To promote the understanding of objects, it is important to emphasize the basic characteristics of objects: identity, state and behaviour. Among other things, this implies that classes modelling mere data containers are not exemplary. To develop real software systems, one would, of course, need such non-exemplary classes. Gil and Maman [2005], for example, showed that a significant portion of the classes in common Java software are "degenerate". However, we argue that novices should internalize the basics rules, before turning to the exceptions.

For the small-scale context, the *Single Responsibility Principle* (SRP) [Martin 2003] implies few attributes and few methods. Furthermore, the educational conditions of a small-scale example will preferably result in few lines of code. Keeping the abstraction focused with few collaborators means less passing of parameters. Encapsulation and information hiding should also be emphasized.

Another important implication of *Model Reasonable Abstractions* is that context is critical to the abstraction. It is close at hand to use objects from real life as examples. But, with every day life examples it is important to explicitly discuss the differences between the model and the modelled. It is difficult for a novice to accept that a model has behaviour and responsibilities that its real-life counterpart never would have [Börstler 2005].

What differs good from bad often depends on small details. For example, for illustrating smaller syntactical components it is not uncommon to use a single application class and place the example in the `main`-method, see Listing 1. This example is not contributing to a novice's understanding of object orientation. If we want to promote the ideas that (1) an object oriented program is a system of communicating objects and that (2) each object represents an individual real or abstract entity with a well-defined role in the problem domain, such examples might do more harm than good. There are no obvious objects in this example. Furthermore, `main` does neither represent any behaviour of an object, nor is it explicitly called. To avoid potential confusion, we should avoid to turn a single `main` into the entire program.

Listing 1. Application illustrating a for-loop.

```
public class Ex
{
```

[1]We use the term *client* to refer to classes/objects that make use of the resources provided by the class/object under development.

```
  public static void main(string[] args)
  {
    int i = 0;
    for (int j=0; j<10; j++)
    {
      i = i+j;
    }
    System.out.println( "Sum = " + i);
  }
} //class Ex
```

To make abstractions reasonable, they should be taken from a domain that is easy to explain and/or familiar to the novices. A typical domain are simple games. The following class a familiar or easy to explain domain. A class modelling a playing card might be a suitable candidate (see Listing 2) [Wick et al. 2004]. But what are the essential properties of playing cards this abstraction is focussing on? A card has a rank and a suit, but these are fixed and cannot be set randomly from the outset. Furthermore, exposing the "accidental" realization of essential properties actually works against the main goal of abstraction. Mentally handling `Card` objects becomes actually more complex instead of easier.

Listing 2. Non-reasonable abstraction for playing card objects (see [Wick et al. 2004].)

```
public class Card
{
  private int rank;  // 2 .. 14
  private char suit; // 'D', 'H', 'S', 'C'

  public int getRank()
  {
    return rank;
  }

  public void setRank(int r)
  {
    rank = r;
  }

  public char getSuit()
  {
    return suit;
  }

  public void setSuit(char s)
  {
    suit = s;
  }
  ...
} //class Card
```

4.2 Model Reasonable Behaviour

A real problem in defining examples for novices is that educators have only a limited set of concepts and syntactical elements to play with. Not everything can be introduced in the first lecture. Examples must be simple enough to not overwhelm a novice with new concepts or syntax, but still feature meaningful object behaviour. Discussing what a client might expect in terms of consistency and logic will most likely extend an example, but will also empower novices in terms of analysis and design thinking. What is reasonable is highly dependent on the context of an example. Without explicit context (like a "cover story" or client classes), the actual meaning of the concept of behaviour is difficult to understand.

When reviewing the playing card example from Listing 2, we should ask ourselves which behaviour would be appropriate for a class `Card`. Changing rank or suit would not be reasonable. The cards in a deck never change suit or value. It might be much more reasonable to have some comparison behaviour between `Card`-objects. Depending on the context for the cards, one suggestion could be Listing 3.

Listing 3. Improved `Card` class (see [Wick et al. 2004]).

```
public class Card
{
  private int rank;  // 2 .. 14
  private char suit; // 'D', 'H', 'S', 'C'

  public Card(int r, char s)
  {
    // Validating rank (r) and suit (s) to construct
    // valid cards only!
    ...
    this.rank = r;
    this.suit = s;
  }

  public boolean isDiamond()
  {
    return suit == 'D';
  }

  public boolean isHigherThan(Card c)
  {
    return this.rank > c.rank;
  }
  ...
} // class Card
```

Reasonable behaviour also means emphasizing the difference between a model and the modelled. A software object does not necessarily have exactly the same behaviour and characteristics as its real world counterpart. In a library system it would, for example, be reasonable for a borrower object to be responsible for keeping track of its outstanding fees. In real life, however, this would be a bad idea. It is important to convey that we do not model the real world, we model systems that solve problems that originate from the real world. This is a subtle, but quite significant difference. Therefore, we have to make an effort to aid novices

in separating the model from the modelled.

Using code snippets without context leaves it up to the learners to imagine the means and ends of an object's behaviour, where this particular code snippet makes sense. We argue that code snippets do not support "object-thinking", but direct focus from the underlying concepts to syntactical details. They actually work against the ideas of abstraction and behaviour, which are central to object orientation.

The `for`-loop in Listing 1, for example, does not aid the understanding of why, when, and how objects would or should have this kind of behaviour. The problem seems highly artificial and it is hard to imagine where summing up 0–10 could be used "for real".

Like code snippets, printing for tracing is also often used in example programs for novices. We argue that printing for tracing not only is a bad habit, but contradicts the very idea of abstraction and communicating objects. Novices are misled to believe that results can only be returned by printing. In teaching practice, it is tempting to use printing to, for example, show the values of (instance) variables as the execution progresses. However, seeing "printing" in too many example programs, novices might conclude that it actually is necessary to do the printing to "get things done".

4.3 Emphasize Client View

Martin [2003] emphasizes that a model must be validated in connection to its clients. This means there must be some context where such clients can "live". From an educational point of view a meaningful context plays an important role for discussing and contrasting the strength and weaknesses of different solutions. Taking a clients' view when discussing the design of a class, promotes the idea of objects as autonomous, collaborating entities. With a meaningful context a particular design gets a purpose. An educator can explain why a certain model is the way it is. Again this helps focusing on concepts instead of syntactical details.

It is crucial to discuss the responsibilities and services of an object as independently as possible from their internal representation and implementation in terms of methods and attributes. We argue that this promotes "object thinking" [West 2004] and makes object oriented problem solving easier for novices. Meyers [2004] gives the following practical advice for defining the protocol/interface of a class: "anticipate what clients might want to do and what clients might do incorrectly". On the other hand, each object should only have a single well-defined responsibility and only contain necessary "features" (see *Single Responsibility Principle* and *Interface Segregation Principle* [Martin 2003]). For examples without context these important issues cannot be discussed in a meaningful way.

The methods provided by a class must make immediate sense to a novice or a good "cover story" must be provided that motivates object behaviour.

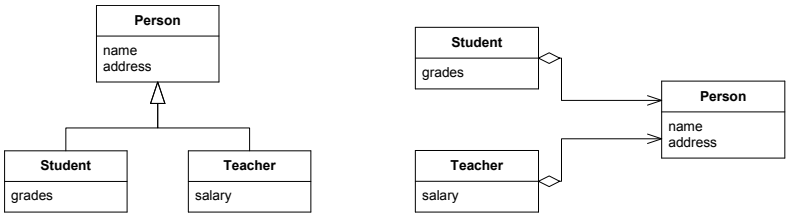
4.4 Favour Composition over Inheritance

Inheritance is the concept that distinguishes object orientation from other paradigms. It is therefore given significant coverage in introductory object oriented programming. Introducing inheritance early typically leads to very simple examples, since novices only have mastered a very limited repertoire of concepts and syntactical constructs. Early examples are therefore usually too simple to show the real power

of inheritance. Furthermore, inheritance is often used to exemplify code reuse, which generally is considered a problematic motivation for inheritance [Armstrong and Mitchell 1994] and may lead to typing problems [Liskov and Wing 1994], for example when implementing a stack by inheriting from a vector or a queue.

Since inheritance is so powerful, it might easily be overused [Armstrong and Mitchell 1994; Johnson and Foote 1988]. We argue that inheritance also might be overeducated. There is so much focus on inheritance in introductory programming that example programs often use inheritance where other solutions might actually be more suited. The usage of inheritance should be carefully considered and discussed. Being careful with inheritance is important to guide novices toward a proper use of inheritance. Novices must not be led to believe that inheritance has to be used as the only relationship between classes.

A common pitfall is to model roles as classes [Fowler 1997; Reenskaug et al. 1996; Skrien 2009; Steimann 2000] as in the common person/student/teacher example, see Figure 2. `Teacher` and `Student` seem good examples of subclasses of `Person`, since a teacher or a student "is-a" person. However, this model is neither flexible nor extensible. Even in very simple contexts, like a course administration system, students might act as teachers or teachers might take some courses. In model 2(a), we would get the `Person`-attributes of persons with double roles twice. This will lead to problems as soon as some of these attributes have to be changed. On might argue that this could be a good example to motivate multiple inheritance. However, this would not solve the problem, since roles will likely change dynamically during the lifetime of a person.



(a) `Teacher` and `Student` modelled as subclasses of `Person`.

(b) `Teacher` and `Student` modelled as roles of `Person`.

Fig. 2. Inheritance versus composition (and delegation) [Skrien 2009].

The design patterns by Gamma et al. [1995] explicitly build on the principle of favouring composition over inheritance. Skrien [2009] shows many examples of how composition can lead to better designs.

4.5 Use Exemplary Objects Only

Learners use examples as role-models or templates for their own work [Lahtinen et al. 2005]. All example properties, even incidental ones, will therefore affect what students learn from the examples. The literature discusses many examples of student-constructed rules or misconceptions that could be avoided by more careful example design [Holland et al. 1997; Ragonis and Ben-Ari 2005b; Fleury 2000; Malan and Halland 2004].

To emphasize the notion of communicating objects, examples should actually feature communicating objects, i.e. at least one explicit instance that sends a message to another explicit instance. To emphasize the differences between objects and classes, examples should feature at least two instances of at least one class. Otherwise students might infer that we need a new class for each new object. However, many textbooks use one-of-a-kind examples heavily, like the `RobberLanguageCryptographer` in Listing 4[2].

Listing 4. Example of a class modelling a one-of-a-kind object.

```
public class RobberLanguageCryptographer
{
  public boolean isConsonant (char c) { ... }

  public String encrypt(String s)
  {
    StringBuffer result = new StringBuffer();
    for (int i = 0; i<s.length(); i++)
    {
      char c = s.charAt(i);
      result.append(c);
      if (isConsonant(c))
      {
          result.append('o');
          result.append(c);
      }
    }
    return result.toString();
  }

  public String decrypt(String s) { ... }
}
```

In this example, multiple instances make no sense, unless it is explicitly shown that different instances can produce different results given the same input. One reason could be that the encoding-algorithm can vary among the the objects, depending on some information submitted to the constructor. The constructor of this example is missing, could be empty, and there are no attributes. This is troublesome for novices, and it is not a good role-model for the definition of objects. This class defines no state, only behaviour, sometimes even static, which is non-exemplary for objects [Booch 1994]. In this case it is no more than two methods (with a small helper, `isConsonant`) that easily could be the responsibility of some other object. A small change can make this example more suitable as a role-model for object orientation. One reason for more than one object of this kind is to let the object take responsibility for knowing its substitution-character, then it will be plausible to have many different cryptographers.

It is also important to be explicit, i.e. using explicit objects whenever possible. Following the *Law of Demeter* is one way to make a design more explicit. Calling methods of explicit objects instead of calling the nameless object resulting from a

[2]This example is actually taken from the exam solutions provided for an introductory programming at a Swedish university.

method call, makes the behaviour of objects less obscure. Anonymous classes is way of making the example shorter which often is desired, but contradictory to the needs of a novice. Since tracing is an important part of learning to program, avoiding anonymous objects and classes, will decrease the cognitive load for a novice. Avoiding anonymity and using explicit objects mean that the use of static elements becomes an issue. Static attributes and static methods can confuse novice of the concepts class, object and behaviour and should preferably be avoided, or deferred.

Another example of non-exemplary objects is when illustrating the instantiation and use of objects within the class itself. To save space (examples should preferably be short in terms of lines-of code!), a `main` method is added to the class, an object is instantiated in main and the class' own methods can be called, often to demonstrate how to use objects of the class. In this case, an unnecessary strain is put on a novice. It is artificial to instantiate an object inside a static method of the class. How can something that does not exist create itself? Still one might argue for using the main-method. One reason for insecurity among novices is the lack of control. A common question is -How is this run? or -Where is the program? Dealing with objects, there is no simple answer to these questions. One of the difficulties with object oriented is the delocalised nature of activities. The flow of control is not obvious, and working with complete applications is one way of gaining a sense of control for the novice programmer. "This is a complete program - and I wrote it myself" is a comforting feeling that should not be underestimated. This can be achieved by adding a class with the single purpose of acting as a client in need of these objects. Through the isolation of `main`, the boundaries of the abstractions/objects stay clear.

4.6 Make Inheritance Reflect Structural Relationships

Inheritance is often over-emphasized and misused when introducing hierarchical structures early. To show the strength and usefulness of inheritance it is necessary to design examples carefully. Behaviour must guide the design of hierarchies and specialisation must be clear and restricted. The *Liskov Substitution Principle* (LSP) promotes polymorphism, but restricts the relationship between the base class and the derived class. This must be taken into account when designing examples. What can be expected of an object of the base class must always be true for objects of the derived class.

A common small-scale example for inheritance is the design of the geometrical shapes rectangles and squares as shown in Listing 5.

Listing 5. A `Rectangle` class.

```
public class Rectangle
{
  private double height, width;

  public Rectangle(double h, double w)
  {
    setHeight(h);
    setWidth(w);
  }

  public void setHeight(double h)
```

```
  {
    height=h;
  }

  public void setWidth(double w)
  {
    width=w;
  }
  ...
} //class Rectangle
```

From a mathematical point of view it might be possible to say that a square is a specialised form of a rectangle (a rectangle with height = length). This could be regarded a reasonable specialisation hierarchy, demanding only a small adjustment in Square to make sure that its height and width are the same, see Listing 6.

Listing 6. `Square` derived from `Rectangle`.

```
public class Square extends Rectangle
{
  public Square(double s)
  {
    super(s, s);
  }

  public void setHeight(double h)
  {
    super.setHeight(h); super.setWidth(h);
  }

  public void setWidth(double w)
  {
    super.setHeight(w); super.setWidth(w);
  }
} //class Square
```

It is reasonable to believe that the designer of the method `setWidth`, assumed that setting the width of a rectangle leaves the height unaltered. One could therefore argue that `Square` violates the *Liskov Substitution Principle* which demands that an object of a subclass can replace an object of its superclasses in any context.

Adhering to the design heuristic "only derive a class from an abstract class" [Riel 1996] would prevent some of the most common problems concerning both exemplifying and understanding inheritance. In our opinion, examples of inheritance, should demonstrate that the base class is an unfinished description shared by structurally related things. Even though geometrically a square might be regarded as a rectangle, it is not true when talking about objects. The behaviour of the derived class, `Square`, is not consistent with the expected behaviour of a `Rectangle` object. From a structural point of view one could instead argue that `Rectangle` should be a subclass of `Square` as shown in Figure 3. The inherited method `changeSize` would be redefined to change `width` and `height` by the same factor.

An extensive discussion of this particular example is given in [Skrien 2009].

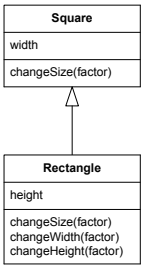


Fig. 3. `Rectangle` inheriting from `Square`.

Table I. Correspondence between accepted OO principles (rows) and He[d]uristics (columns).

	H1: Reasonable Abstractions	H2: Reasonable Behaviour	H3: Emphasize ClientView	H4: Favour Composition	H5: Exemplary Objects	H6: Structural Relationships
Single Responsibility Principle	X	X	X		X	
Open − Closed Principle						X
Liskov Substitution Principle	X					X
Dependency Inversion Principle		X				X
Interface Segregation Principle	X	X	X			
Law of Demeter		X			X	
Favour Object Composition Over Class Inheritance				X	X	

5. DISCUSSION

In [Nordström 2009], we have evaluated how well our He[d]uristics cover basic object oriented concepts and commonly accepted object oriented design principles. The results of this evaluation, summarized in Table I and Table II, show that all important concepts and principles are covered well. This indicates that the He[d]uristics actually address all properties of object orientation that are generally considered relevant for novices. However, this does not show whether an example designed according to these heuristics actually holds high object oriented quality or not.

In [Börstler et al. 2008], we present a checklist for evaluating the quality of examples according to their technical, object oriented, and didactical quality. This checklist was later refined and used in a large-scale study reviewing 38 example programs from 13 common introductory programming textbooks [Börstler et al. 2009; Börstler et al. 2010]. Regarding object oriented quality, the checklist covered the following quality factors:

—*Reasonable Abstractions*: Abstractions are plausible from an OO modelling perspective as well as from a novice perspective.

—*Reasonable State and Behaviour*: State and behaviour make sense in the presented software world context.

Table II. Correspondence between basic OO concepts (rows) and He[d]uristics (columns).

	H1: Reasonable Abstractions	H2: Reasonable Behaviour	H3: Emphasize ClientView	H4: Favour Composition	H5: Exemplary Objects	H6: Structural Relationships
Responsibility	X	X		X	X	
Abstraction	X		X		X	X
Encapsulation	X	X		X	X	X
Information Hiding			X		X	
Inheritance	X			X		X
Polymorphism				X		X
Protocol/ Interface	X	X	X			X
Communication	X	X	X	X	X	X
Class	X	X	X			X
Object	X	X	X	X	X	X

Table III. Correspondence between OO quality factors (rows) and He[d]uristics (columns).

	H1: Reasonable Abstractions	H2: Reasonable Behaviour	H3: Emphasize ClientView	H4: Favour Composition	H5: Exemplary Objects	H6: Structural Relationships
Reasonable Abstractions	X				X	X
Reasonable State and Behaviour		X	X		X	X
Reasonable Class Relationships				X		X
Exemplary OO	X	X			X	X
Promotes "Object Thinking"	X	X	X	X	X	

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

—*Exemplary OO code*: The example is free of "code smells".

—*Promotes "Object Thinking"*: The example supports the notion of an OO program as a collection of collaborating objects.

The results of this study showed that the checklist captured the strengths and weaknesses of examples well. There was also very high reviewer agreement, indicating that the checklist is a reliable evaluation instrument. Correspondencies between the heuristics presented in the present paper and the checklist are summarized in Table III. The table shows that the heuristics cover object oriented qualities well.

In the following, we will discuss our He[d]uristics with respect to the checklist results. Examples that scored low according to the checklist should indicate violation of at least some He[d]uristics and examples that scored high should not violate any He[d]uristic. We will furthermore discuss in which way the He[d]uristics can help

to improve an example.

An important type of example is the first user defined class (FUDC) novices are confronted with. These examples are the initial frame of reference for novices' perception of object orientation, and are therefore crucial. FUDC-examples must therefore be exemplary and follow high standards. In [Börstler et al. 2010], we examined 9 FUDC-examples that shifted significantly in their rating on object oriented quality factors.

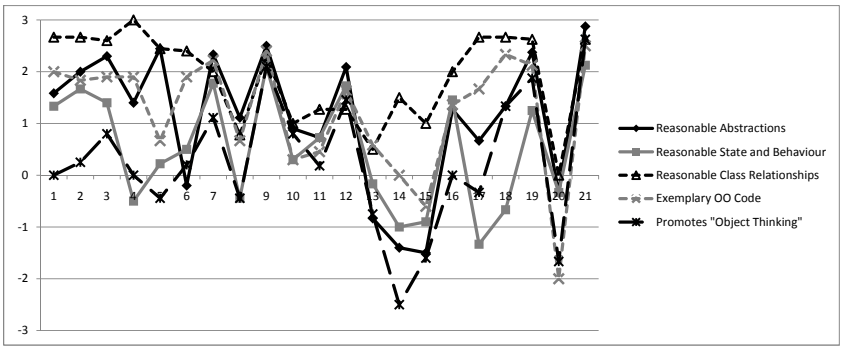


Fig. 4. Average object oriented quality scores for the 21 examples in [Börstler et al. 2010] (range: +3(extremely poor) .. -3(excellent)).

To illustrate the influence He[d]uristics might have on the design of examples we examine two example from the study discussed above: one that scored low and one that scored high.

The lowest-scoring FUDC-example models a `GradeBook`. In this example different components of the final class are introduced gradually and with each new version a class `GradeBookTest` is shown that illustrates the instantiation and use of `GradeBook` objects. Initially the class contains only a single method (`displayMessage`) that prints a fixed string as a welcome message. For the second version a parameter is added to the method to vary the welcome message. Next, an instance variable for the course name and methods to get and set the course name are added. At this stage the parameter in `displayMessage` is removed again. Finally a constructor is added, see Listing 7 for the final (full) version of `GradeBook`.

Listing 7. First User Defined Class: `GradeBook`

```
// GradeBook class with a constructor to initialize the course name.

public class GradeBook
{
  private String courseName; // course name

  // constructor initializes courseName
  public GradeBook( String name )
  {
    courseName = name;        // initialize courseName
  }
```

```
  // method to set the course name
  public void setCourseName( String name )
  {
    courseName = name;       // store the course name
  }

  // method to retrieve the course name
  public String getCourseName()
  {
    return courseName;
  }

  // display a welcome message to the GradeBook user
  public void displayMessage()
  {
    // this statement calls getCourseName to get the
    // name of the course this GradeBook represents
    System.out.printf( "Welcome to the grade book for\n%s!\n",
      getCourseName() );
  }
} //class GradeBook
```

Using our He[d]uristics, we would evaluate the object oriented qualities of this example in the following way:

(1) *Model Reasonable Abstractions*: What entity in the problem domain is this class modelling? How can it be argued that this is a well chosen abstraction representing a component in the problem domain? It would be reasonable for a grade book to keep track of student records, but this is not mentioned explicitly, and not supported in the details of the implementation.

(2) *Model Reasonable Behaviour*: This class does not have any behaviour at all, and it is not indicated in the accompanying text what objects of this kind would be needed for. It states: *Class* `GradeBook` *will be used to display a message on the screen welcoming the instructor to the grade-book application.*

(3) *Emphasize Client View*: No discussion of whom the client might be is supplied by the example. The object does not have state and behaviour, not even in its final version.

(4) *Favour Composition over Inheritance*: There are no relationships introduced in this problem, but if discussed what a `GradeBook`-object should be responsible for (student records), it would have been possible to indicate this as a composition of objects.

(5) *Use Exemplary Objects Only*: The example does not support the idea of many objects. It starts with just a print method, and does not support the idea of objects having state and behaviour.

(6) *Make Inheritance Reflect Structural Relationships*: Not applicable, since no inheritance is used or needed in this example.

In comparison, we also want to take a look at a high-scoring example, modelling a class `Die`, see Listing 8.

Listing 8. First User Defined Class: `Die`

```
public class Die
{
  private final int MAX = 6;  // maximum face value
  private int faceValue;      // current value showing on the die

  public Die()
  {
    faceValue = 1;
  }

  public int roll()
  {
    faceValue = (int)(Math.random() * MAX) + 1;
    return faceValue;
  }

  public int getFaceValue()
  {
    return faceValue;
  }

  public String toString()
  {
    String result = Integer.toString(faceValue);
    return result;
  }
} //class Die
```

From the perspective of our He[d]uristics, we would argue as follows:

(1) *Model Reasonable Abstractions*: The abstraction is crisp and easy to understand. It is also reasonable from a software perspective. It is easy to find possible applications where such a class could make sense.

(2) *Model Reasonable Behaviour*: One would expect that a die can be rolled and rolling should potentially change the die's face value. That corresponds closely to a die's "behaviour" in reality. What seems less reasonable is that all dice initially have a face value of 1. Also less reasonable is `setFaceValue`-method, unless its existence is motivated by the context the example is set in. Furthermore, the `Die` interface is not minimal; the methods are not orthogonal. The `roll`-method returns a new face value, although there is a method for accessing the face value.

(3) *Emphasize Client View*: In its original context, the `Die`-example starts with a test class exemplifying the instantiation and use of two `Die`-objects. However, the usage scenario is not very convincing, since, for example, the face value of a die is set manually.

(4) *Favour Composition over Inheritance*: Not applicable. There are no class relationships in this problem, nor does it seem sensible to extend the example.

(5) *Use Exemplary Objects Only*: It is easy to imagine applications with many `Die`-objects. `Die`-objects have state, behaviour and a clear, well-defined (single) responsibility. There are no superfluous printing methods and the `toString`-

method just returns a die's face value, leaving decisions about textual representations up to the client.

(6) *Make Inheritance Reflect Structural Relationships*: Not applicable, since no inheritance is used or needed in this example.

The issues raised above can be easily fixed, since the underlying abstraction is reasonable. Small adjustments would turn the `Die`-class into an exemplary FUDC example, see Listing 9.

Listing 9. First User Defined Class: `Die` – slighly adjusted

```
public class Die
{
  private final int FACES = 6;  // a standard die has 6 faces
  private int faceValue;        // current value showing on the die

  public Die()
  {
    roll();                     // sets a random initial face value
  }

  public void roll()
  {
    faceValue = (int)(Math.random() * FACES) + 1;
  }

  public int getFaceValue()
  {
    return faceValue;
  }

  public String toString()
  {
    return Integer.toString(faceValue);
  }
} //class Die
```

In this example it could be argued that the constant `FACES` should be declared `static`, since it is supposed to be the same for all dice, and that it is better from a design point of view to have the constant shared by all objects. It could also be argued that it would be slightly more general to allow for the client to choose the number of faces for a particular die.

Both of these objections are justifiable, but both have consequences. Declaring the constant `FACES` static would require the novice to know the concept of static elements of classes. Allowing for variable number of faces, means that the example must have two constructors, the input value must be validated in some way, and one constructor should call the other using `this`. All contributing in making the example more complex.

So, it is really important to adjust an example to the actual situation where it is being used. The knowledge of the learner could require compromises with software quality demands, and the challenge is to find a balance to uphold the basic principles of object orientation, and at the same time acknowledge the educational conditions.

6. CONCLUSIONS

In this paper, we have described a number of heuristics for the design of object oriented examples for novices. The foundation for these heuristics are concepts and principles constituting object orientation. We have also shown how these heuristics can help in designing or improving examples.

In the He[d]uristics, the word reasonable is rather imprecise. What is reasonable is context-dependent, and it is therefore vital to design examples carefully and to discuss the differences in solutions depending on the contexts. Furthermore, reasonable also indicates that the responsibility lies heavily on the designer of examples to scrutinize what object orientation really means. As we all have experienced, this is not a trivial task. There are many limiting conditions to take into account in teaching, and in small-scale examples we always have to compromise. It is important to take into consideration that examples always fulfill several purposes. One is the very immediate one; exemplifying a certain concept or construction, syntactical or semantical, but there is also a more generic exemplification of the paradigm itself and what could be called object thinking.

However, surprisingly often, being particular about details can make all the difference when it comes to upholding object oriented quality.

REFERENCES

ACM. 2008. Computing curricula update 2008. `http://www.acm.org/education/curricula/ComputerScienceCurriculumUpdate2008.pdf`. Last visited: 2008-12-15.

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

Armstrong, J. and Mitchell, R. 1994. Uses and abuses of inheritance. *Software Engineering Journal 9,* 1, 19–26.

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

Booch, G. 1994. *Object-Oriented Analysis and Design with Applications, 2nd edition.* Addison-Wesley.

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.* ACM, 356–364.

Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Kallin Westin, L., Jan-ErikMoström, and Caspersen, M. E. 2008. Evaluating oo example programs for cs1. In *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education.* ACM, New York, NY, USA, 47–52.

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., 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.*

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

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

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.* ACM, New York, NY, USA, 93–96.

Caspersen, M. E. 2007. Educating novices in the skills of programming. Ph.D. thesis, University of Aarhus, Denmark.

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.

De Raadt, M., Watson, R., and Toleman, M. 2005. Textbooks: Under inspection. Tech. rep., University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia.

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 *SE 2006 International Multi-Conference on Software Engineering*, P. Kokol, Ed. IASTED, 346–355.

Eckerdal, A. and Thuné, M. 2005. Novice java programmers' conceptions of "object" and "class", and variation theory. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*. ACM, New York, NY, USA, 89–93.

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

Fowler, M. 1997. Dealing with roles. In *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP)*.

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.

Gibbon, C. 1997. Heuristics for object-oriented design. Ph.D. thesis, University of Nottingham.

Gil, J. Y. and Maman, I. 2005. Micro patterns in Java code. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*. San Diego, CA, USA, 97–116.

Grotehen, T. 2001. Objectbase design: A heuristic approach. Ph.D. thesis, University of Zurich, Switzerland.

Guzdial, M. 2008. Paving the way for computational thinking. *Commun. ACM 51,* 8, 25–27.

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*. 131–134.

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

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*. 14–18.

Liskov, B. H. and Wing, J. M. 1994. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16,* 6, 1811–1841.

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*. 83–87.

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*. T4G–1–T4G–6.

Meyers, S. 2004. The most important design guideline? *IEEE Softw. 21,* 4, 14–16.

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

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.

Ragonis, N. and Ben-Ari, M. 2005a. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education 15,* 3, 203–221.

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

Reenskaug, T., Wold, P., and Lehne, O. A. 1996. *Working With Objects: The OOram Software Engineering Method*. Manning/Prentice Hall.

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

Rosch, E. 1999. Principles of categorization. In *Concepts: Core Readings*, E. Margolis and S. Laurence, Eds. MIT Press, 189–206.

Sanders, K., Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Thomas, L., and Zander, C. 2008. Student understanding of object-oriented programming as expressed in concept maps. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer science education*. ACM, New York, NY, USA, 332–336.

Skrien, D. 2009. *Object-Oriented Design Using Java*. McGraw Hill.

Steimann, F. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering 35,* 1, 83–106.

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

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

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*. ACM, Norfolk, Virginia, USA.

Received October 2010; ...

Paper III

Evaluating OO Example Programs for CS1

Evaluating OO Example Programs for CS1

Jürgen Börstler
Dept. of Computing Science
University of Umeå, Sweden
jubo@cs.umu.se

Henrik B. Christensen
Dept. of Computer Science
University of Aarhus, Denmark
hbc@daimi.au.dk

Jens Bennedsen
IT University West
Aarhus, Denmark
jbb@it-vest.dk

Marie Nordström
Dept. of Computing Science
University of Umeå, Sweden
marie@cs.umu.se

Lena Kallin Westin
Dept. of Computing Science
University of Umeå, Sweden
kallin@cs.umu.se

Jan Erik Moström
Dept. of Computing Science
University of Umeå, Sweden
jem@cs.umu.se

Michael E. Caspersen
Dept. of Computer Science
University of Aarhus, Denmark
mec@daimi.au.dk

ABSTRACT

Example programs play an important role in learning to program. They work as templates, guidelines, and inspiration for learners when developing their own programs. It is therefore important to provide learners with high quality examples. In this paper, we discuss properties of example programs that might affect the teaching and learning of object-oriented programming. Furthermore, we present an evaluation instrument for example programs and report on initial experiences of its application to a selection of examples from popular introductory programming textbooks.

Categories and Subject Descriptors

K3.2 [**Computers & Education**]: Computer and Information Science Education—*computer science education*

General Terms

Experimentation, Human Factors, Measurement

Keywords

CS1, example programs, object-orientation, quality

1. INTRODUCTION

Examples are important tools for teaching and learning. Both students and teachers cite example programs as the most helpful materials for learning to program [10]. Also research in cognitive science confirms that "examples appear to play a central role in the early phases of cognitive skill acquisition" [18, p 515]. Moreover, research in cognitive load theory has shown that carefully worked-out examples (so called worked examples) play an important role in order to increase learning outcome [5].

In mathematics education exemplification is a well researched topic [11] and "the choice of examples that learners are exposed to plays a crucial role in developing their ability to generalize" [21, p 131]. Examples must therefore always be consistent with all learning goals and follow the principles, guidelines, and rules we want to convey. Otherwise, students will have a difficult time recognizing patterns and telling an example's non-essential properties (noise) from those that are structurally or conceptually important.

Learner's will learn from examples, but we cannot guarantee that they abstract the properties and rules we want them to learn. They might not see the generality and just mimic irrelevant example properties and features [13]. The rules they construct might be erroneous, since there are many ways to interpret and generalize example features [7, 9, 18]. It is therefore also important to present examples in a way that conveys their "message", but at the same time be aware of what learners might actually see in an example [13].

Carefully developed and presented examples, can help preventing misconceptions [4, 8].

In this paper, we discuss quality properties of example programs and formulate criteria which are used to develop an evaluation instrument. We then present the results of using this instrument on a selection of program examples from popular textbooks. Finally, we discuss the issues concerning the design and usage of the evaluation instrument and outline how our work can be taken further.

2. RELATED WORK

Although examples are perceived as one of the most important tools for the teaching and learning of programming, there is very little research in this area. Most often example issues are only discussed in the narrow context of a single simple and concrete example, like the recurring "Hello World"-type discussions [6, 19], or they are regarded as a language issue [2, 15]. Only few authors have taken a broader view by investigating features of example programs and their (potential) effects on learning.

ITiCSE'08, June 30–July 2, 2008, Madrid, Spain.

Wu et al. studied programming examples in 16 high school computer textbooks and concluded that most of them "lacked detailed explanation of some of the problem-solving steps, especially problem analysis and testing/debugging" [20, p 225]. Almost half of the examples fell into either the math-problem (27%) or syntax-problem (21%) category.

Holland et al. [9] provide guidelines for designing example programs to prevent object-oriented misconceptions, which are successfully used by Sanders and Thomas [16] for assessing student programs.

Malan and Halland [12] describe four common pitfalls that should be avoided when developing example programs. They argue that examples that are too abstract or too concrete, that do not apply the taught concepts consistently, or that undermine the concept they are introducing, might hinder learning.

Furthermore, there are many studies of software development in general showing that adherence to common software design principles, guidelines, and rules [3], as well as certain coding, commenting, naming guidelines, and rules [14, 17] support program understanding.

There is also a large body of research on worked examples providing general guidelines regarding the form and presentation of examples [5].

However, to our knowledge, neither of the above principles, guidelines, and rules have been used to evaluate example programs from programming textbooks.

3. RESEARCH APPROACH

This project is carried out by two research groups from two countries. During an initial two-day workshop, a large number of example programs from different textbooks were discussed to identify common strengths and weaknesses. The goal was to define a set of criteria to effectively discriminate between different levels of "quality", based on accepted principles, guidelines, and rules from the literature (see Section 2) and our own teaching experience. The outcome of this workshop was an initial evaluation instrument and a test set of textbook examples.

The instrument was tested on two examples by four reviewers, which lead to several revisions of the instrument. After testing further examples, the instrument was finally refined to the one described in Section 4. The instrument was then used by six reviewers (two female, four male; age 37–48) to evaluate five example programs. All reviewers are experienced computer science lecturers in object-oriented programming, most of them at the introductory level. A summary of the evaluation results are presented in Section 5. Finally, validity and reliability of detailed results were discussed between reviewers and researchers (the groups overlapped considerably) in small groups and by e-mail. A summary of these discussions is presented in Section 6.

Table 1: Categorization of example programs.

	First example	First user-defined class	Several classes	Inheritance
E1	—	—	X	—
E2	X	—	—	partly
E3	—	X	—	—
E4	—	X	partly	—
E5	—	—	X	X

To cover a wide range of aspects, we chose representative examples of different levels of quality and complexity covering the following aspects; the very first example of a textbook, the first exemplification of developing/writing a (user-defined) class, the first application involving at least two interacting classes and a non-trivial (but still simple) example of using inheritance. Table 1 summarizes the features of our five examples E1–E5.

4. EVALUATION INSTRUMENT

Inspiration for the evaluation instrument was drawn from the checklist-based evaluation by the Benchmarks for Science Literacy project [1] by defining a set of specific, well-defined criteria that can be evaluated on a uniform scale. All criteria should be based on accepted programming principles, guidelines, and rules; educational research; and the groups' collective teaching experience. The resulting set of 11 criteria was grouped into three independent categories of quality; *technical quality* (three items), *object-oriented quality* (two items) and *didactic quality* (six items).

Technical quality (T1–T3). The criteria in this category focus on technical aspects of example programs that are independent of the programming paradigm. Examples should be syntactically and semantically correct, written in a consistent style, and follow accepted programming principles, guidelines, and rules (see Table 2).

Table 2: Checklist items for technical quality.

T1	**Problem versus implementation.** The code is appropriate for the purpose/problem (note that the solution need not be OO, if the purpose/problem does not suggest it).
T2	**Content.** The code is bug-free and follows general coding guidelines and rules. All semantic information is explicit. E.g., if preconditions and/or invariants are used, they must be stated explicitly; dependencies to other classes must be stated explicitly; objects are constructed in valid states; the code is flexible and without duplication.
T3	**Style.** The code is easy to read and written in a consistent style. E.g., well-defined intuitive identifiers; useful (strategic) comments only; consistent naming and indentation.

Object-oriented quality (O1–O2). The criteria in this category address technical aspects that are specific for the object-oriented paradigm, i.e., how well an example can be considered a role model of an object-oriented program. In contrast to technical quality, the principles, guidelines, and rules covered here are specific for the object-oriented paradigm (see Table 3).

Table 3: Checklist items for object-oriented quality.

O1	**Modeling.** The example emphasizes OO modeling. E.g., emphasizes the notion of OO programs as collections of communicating objects (i.e., objects sending messages to each other); models suitable units of abstraction/decomposition with well-defined responsibilities on all levels (package, class, method).
O2	**Style.** The code adheres to accepted OO design principles. E.g., applies proper encapsulation and information hiding; adheres to the Law of Demeter (no inappropriate intimacy); avoids subclassing for parameterization; etc.

Didactical quality (D1–D6). The criteria in this category deal with instructional design, i.e., comprehensibility and alignment with learning goals for introductory (object-oriented) programming (see Table 4).

In order to evaluate an example the expected previous knowledge of a student and supporting explanations must be taken into account. In textbooks the placement of an example naturally defines the expected previous knowledge of

Table 4: Checklist items for didactic quality.

D1	**Sense of purpose.** Students can relate to the example's domain and computer programming seems a relevant approach to solve the problem. In contrast to, e.g., flat washers which are only relevant to engineers, if the concept or word is at all known to students outside the domain (or English-speaking countries).
D2	**Process.** An appropriate programming process is followed/described. I.e., the problem is stated explicitly, analyzed, a solution is designed, implemented and tested.
D3	**Breadth.** The example is focused on a small coherent set of new concepts/issues/topics. It is not overloaded with new "stuff" or things introduced "by the way". Students' attention must not be distracted by irrelevant details or auxiliary concepts/ideas; they must be able to get the point of the example and not miss "the forest for the trees". In contrast to, e.g., explaining JavaDoc in detail when the actual topic is introducing classes.
D4	**Detail.** The example is at a suitable level of abstraction for a student at the expected level and likely understandable by such a student (avoid irrelevant detail). In contrast to, e.g., when an example sets out to describe the concept of state of objects, but winds up detailing memory layout in the JVM).
D5	**Visuals.** The explanation is clear and supported by meaningful visuals. E.g., uses visuals to explain the differences between variables of primitive (built-in) types and object types. In contrast to, e.g., showing a generic UML diagram as an after-thought without relating to the actual example.
D6	**Prevent misconceptions.** The example illustrates (reinforces) fundamental OO concepts/issues. Precautions are taken to prevent students from overgeneralizing or drawing inappropriate conclusions. E.g., multiple instances of at least one class (to highlight the difference between classes and objects); not just "dumb" data-objects (with only setters and getters); show both primitive attributes and class-based attributes; methods with non-trivial behavior; dynamic object creation; etc.

a student. In the context of this work an example is considered as a complete application or applet plus all supporting explanations related to this particular program.

To summarize, one could say that T1–T3 and O1–O2 assess the actual code of an example program and D1–D6 assess how it is presented to and conceived by a student. The categories complement each other; an example of high technical and object-oriented quality will not be very effective, if it cannot be understood by the average student. However, such an example might still be a very valuable teaching resource, in case the educator using it finds better ways to explain it.

All ratings in the resulting checklist are on a Likert-type scale from 1 (strongly disagree) to 5 (strongly agree). Since all checklist items are formulated positively, 5 is always best. Beyond the criteria described in Tables 2–4, the actual checklist also contains additional fields for commenting each rating and a field for overall comments that might not fit any of the available criteria. An example of a filled-in checklist can be found at `http://www.cs.umu.se/research/education/checklist_iticse08.pdf`.

5. RESULTS

The results presented here are based on the evaluation that was made in order to answer two questions:

- Can the instrument distinguish between "good" and "bad" examples?
- Do reviewers interpret the items of the instrument in the same way?

Figure 1 summarizes the results of six reviewers' evaluation of the five examples, E1–E5 (see also Table 1). As can be seen, only one example (E1) is consistently rated very high across all three quality categories. Low average ratings have almost always a relatively high standard deviation (i.e., disagreement between reviewers).

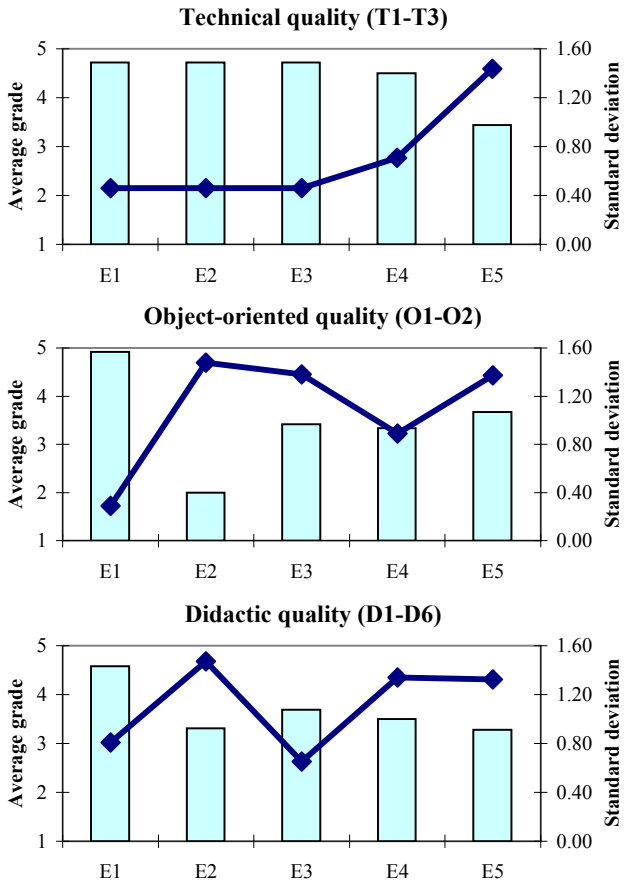


Figure 1: Average grade (bars) and standard deviation (line) for evaluation of five examples. Results are shown by item category (technical, object-oriented, and didactic quality).

Besides the overall high rating of E1, there are several other noteworthy observations. The overall technical quality of the reviewed example programs is very high, except for E5 which did not correctly implement its stated requirements. The section on object-oriented quality has the largest variation. It should, however, be noted that E2 is a "Hello World"-type example which cannot be expected to achieve high ratings in this category. Given that we used examples from quite popular textbooks, the overall ratings for didactic quality and the ratings of E3–E5 on object-oriented quality were surprisingly low.

Figure 2 shows the overall distribution of ratings for each of the six reviewers, R1–R6. It can be noted that the reviewers utilize the rating scale differently. Reviewer R5, for example, used the best grade (5) only half as much as the average (21.8% compared to 43% for all reviewers together). Reviewer R6, on the other hand, did not use a single 1. However, except for reviewer R5, the distributions of ratings are quite similar (in total the usage of rating 1 was only 7.3%).

It seems that teaching experience somewhat influences the grading. One reviewer, R5, has exclusively taught advanced

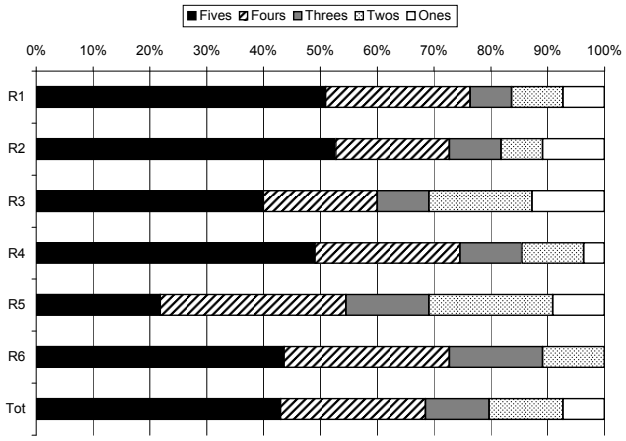


Figure 2: Distribution of ratings between reviewers.

programming courses for the last couple of years, and grades given by R5 tend to be slightly lower on average.

To summarize, it is evident that the instrument distinguishes between examples. Furthermore, the example with the overall highest ratings, E1, is also considered to be a "good" example by the reviewers. However, when looking at Figure 2, it is evident that there are differences in ratings among the reviewers.

6. DISCUSSION

The purpose of the presented instrument has been to replace intuitive "I know it, when I see it" assessments of example programs with a more objective and reliable measurement. So what conclusions can we draw from the results? Are the criteria meaningful? How well do assessment results using the instrument reflect an experienced teacher's overall "gut feeling" of example quality?

This section summarizes the discussions among researchers and reviewers regarding validity and reliability of the evaluation instrument.

6.1 Quality Categories

Overall, we find that the instrument ranked examples as we might have done based on "gut feeling" alone. The instrument was quite useful to point out particular strengths and weaknesses. The reviewers found the three categories natural and covering all important issues of examples.

Technical quality (T1–T3). Assuming that textbook authors have developed and tested their examples carefully, one would generally expect the technical grades to be very high. This is also reflected in consistent and high ratings. The only exception is E5, that contains a defect and the resulting program does not fulfill its stated requirements. This issue was well captured by criteria T2. The standard deviation is generally low, indicating objectivity of the criteria.

Object-oriented quality (O1–O2). In general, the object-oriented characteristics of examples seem to be captured well. E1 received the highest rating and was also agreed to be an exemplary example. E2 is the first example in a textbook (see Table 1) and not really focusing on object-oriented techniques, which is reflected by its low rating in this category. In three of the examples, the standard deviation is high. One reason for this seems to be a lack of common agreement on the importance of explicit object interaction. Another reason seems to be the dependency between O1 and O2 (see Section 6.2).

Didactic quality (D1–D6). When comparing the average grades in this category, one example is rated high; the others are rated lower but at approximately the same level. This is also in concordance with the comments from the discussion after the evaluation. Three of the examples exhibit a high standard deviation indicating a high degree of disagreement among the reviewers. It seems that the reviewers do not share a common understanding of the meaning of the criteria and how they should be rated.

6.2 Criteria

Although all criteria were discussed thoroughly and the instrument was tested twice, we underestimated the semantical issues concerning the criteria. It was implicitly assumed that all reviewers shared the same interpretation of each single criterion. However, careful inspection of all evaluations revealed that the rating still was difficult in some cases.

O1 vs. O2. Since criteria are supposed to be independent, several reviewers gave high ratings for O2 and low ratings for O1 for the same example. They argued that accepted object-oriented principles and guidelines (like encapsulation and information hiding) could very well be followed even by examples that are not considered object-oriented and we should not "penalize" an example twice for the same reason. However, this is in conflict with the overall intention of the category, namely rating object-oriented quality. Lack of object-orientedness should result in low ratings. Therefore, it is necessary to agree on and carefully describe the intended use of O1 and O2.

A solution could be to replace O1 and O2 by a single item for overall object-orientedness and conditional items for a more detailed assessment of object-oriented characteristics. The detailed assessment would then only be carried for examples above a certain threshold value for overall object-orientedness.

D3 vs. D4. Breadth and detail are two views on extraneous or superfluous material. It can therefore be difficult to decide how to balance ratings on these and there might be cases where an example has been penalized twice. A solution could be to use only one criterion assessing the amount of extraneous or superfluous material.

D5. Some reviewers gave high ratings for visuals, although the example lacked visuals, arguing that the explanation was perfectly clear and understandable even without visuals. Discussing this aspect revealed that there might be a more general problem of rating criteria that are simply not addressed properly by an example, e.g., O1, O2, and D2.

D6. When the instrument was constructed it was debated whether D6 should be regarded as "object-oriented quality" rather than "didactic quality". Moving the ratings for D6 to this category resulted, however, only in minor changes of the results as compared to Figure 1.

Granularity and impact. It might be tempting to compute a single value for the rating of an example. However, granularity and impact of criteria can never be perfectly equal. Even within a category, criteria will be valued differently by different people. Moreover, categories with many criteria might be overemphasized.

We have tried to balance criteria within a category. However, an overall total (over all categories) seems not very meaningful.

6.3 Rating Instructions

Already when developing the instrument, a recurring topic of discussion was when to rate a criterion for an example as 1 and when to rate it as 5, i.e., to get a common understanding of the extremes for each criterion. During these discussions, examples were often used to illustrate these extremes. Despite these discussions, reviewers utilized the rating scale quite differently (see Figure 2). If this or a similar instrument is to be used in a community, we strongly recommend supplying a written instruction, containing prototypical examples, with the instrument.

7. SUMMARY AND CONCLUSIONS

In this paper, we have described the design and test of an instrument for evaluating object-oriented examples for educational purposes. The instrument was tested by six experienced educators on five examples, which we consider quite representative for a wide range of examples from introductory programming textbooks.

Our results show that such an instrument is a useful tool for indicating particular strengths and weaknesses of examples. Although only five examples from introductory programming textbooks were formally evaluated, the results indicate that there might be large variations regarding object-oriented and didactic quality of textbook examples. Since examples play an important role in learning to program, it would be valuable to formally evaluate textbook examples at a larger scale.

However, we consider the evaluation instrument presented here not reliable enough for evaluations on a larger scale; inter-rater agreement is too low. As discussed in Section 6, this problem can be reduced by revising the criteria to avoid misunderstandings and, most importantly, developing detailed rating instructions.

8. REFERENCES

[1] AAAS. Benchmarks for science literacy, a tool for curriculum reform, 1989. `http://www.project2061.org/publications/bsl/default.htm`, last visited 2007-12-07.

[2] L. Böszörményi. Why Java is not my favorite first-course language. *Software-Concepts & Tools*, 19(3):141–145, 1998.

[3] L. Briand, C. Bunse, and J. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Transactions on Software Engineering*, 27(6):513–530, 2001.

[4] M. Clancey. Misconceptions and attitudes that infere with learning to program. In S. Fincher and M. Petre, editors, *Computer Science Education Research*, pages 85–100. Taylor & Francis, Lisse, The Netherlands, 2004.

[5] R. Clark, F. Nguyen, and J. Sweller. *Efficiency in Learning, Evidence-Based Guidelines to Manage Cognitive Load*. Wiley & Sons, San Francisco, CA, USA, 2006.

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

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

[8] M. Guzdial. Centralized mindset: A student problem with object-oriented programming. In *Proceedings of the 26th Technical Symposium on Computer Science Education*, pages 182–185, 1995.

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

[10] E. Lahtinen, K. Ala-Mutka, and H. Järvinen. 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, 2005.

[11] Liz, Bills, T. Dreyfus, J. Mason, P. Tsamir, A. Watson, and O. Zaslavsky. Exemplification in mathematics education. In *Proceedings of the 30th Conference of the International Group for the Psychology of Mathematics Education, Vol. 1*, pages 126–154, 2006.

[12] K. Malan and K. Halland. 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, 2004.

[13] J. Mason and D. Pimm. Generic Examples: Seeing the General in the Particular. *Educational Studies in Mathematics*, 15(3):277–289, 1984.

[14] P. Oman and C. Cook. Typographic style is more than cosmetic. *Communications of the ACM*, 33(5):506–520, 1990.

[15] N. Ourosoff. Primitive types in Java considered harmful. *Communications of the ACM*, 45(8):105–106, 2002.

[16] K. Sanders and L. Thomas. Checklists for grading object-oriented cs1 programs: Concepts and misconceptions. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 166–170, 2007.

[17] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *Journal of Programming Languages*, 4(143):167, 1996.

[18] K. VanLehn. Cognitive skill acquisition. *Annual Review of Psychology*, 47:513–539, 1996.

[19] R. Westfall. 'Hello, World' considered harmful. *Communications of the ACM*, 44(10):129–130, 2001.

[20] C.-C. Wu, J. M.-C. Lin, and K.-Y. Lin. A content analysis of programming examples in high school computer textbooks in taiwan. *Journal of Computers in Mathematics and Science Teaching*, 18(3):225–244, 1999.

[21] R. Zazkis, P. Liljedahl, and E. J. Chernoff. The role of examples in forming and refuting generalizations. *ZDM Mathematics Education*, 40:131–141, 2008.

Paper IV

On the Quality of Examples in Introductory Java Textbooks

On the Quality of Examples in Introductory Java Textbooks

JÜRGEN BÖRSTLER and MARIE NORDSTRÖM
Umeå University
and
JAMES H PATERSON
Glasgow Caledonian University

Example programs play an important role in the teaching and learning of programming. Students as well as teachers rank examples as the most important resources for learning to program. Example programs work as role models and must therefore always be consistent with the principles and rules we are teaching.

However, it is difficult to find or develop examples that are fully faithful to all principles and guidelines of the object-oriented paradigm and also follow general pedagogical principles and practices. Unless students are able to engage with good examples, they will not be able to tell desirable from undesirable properties in their own and others' programs.

In this paper we report on a study in which experienced educators evaluated the quality of object-oriented example programs for novices from popular Java textbooks. The evaluation was accomplished using an on-line checklist that elicited responses on the technical, object-oriented, and didactic quality of examples.

In total 25 reviewers contributed 215 reviews to our data set, based on 38 example programs from 13 common introductory programming textbooks. Results show that the evaluation instrument is reliable in terms of inter-rater agreement. Overall, example quality was not as good as one might expect from common textbooks, in particular regarding certain object-oriented properties.

We conclude that educators should be careful when taking examples straight out of a textbook.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented Programming; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education*

General Terms: Principles, Guidelines, Examples

Additional Key Words and Phrases: Example programs, check list, courseware, textbooks, assessment

1. INTRODUCTION

Although example programs are perceived as one of the most important tools for the teaching and learning of programming [Lahtinen et al. 2005], there is very little research regarding their properties and usage. There is a large body of knowledge on

Author's address: Jürgen Börstler, Department of Computing Science, Umeå University, SE-90187 Umeå, Sweden; email: jubo@acm.org.

program comprehension (e.g., [Brooks 1983; Burkhardt et al. 2002]) and software quality and measurement (e.g., [Purao and Vaishnavi 2003]), but this is rarely applied in an educational setting [Börstler et al. 2007; Magel 1982].

In this paper, we describe a checklist-based approach for assessing object-oriented example programs for novices. The checklist covers technical, object-oriented and didactic qualities of example programs. The checklist has been used to evaluate common textbook examples. Our results show that the checklist is reliable and helps to indicate strengths and weaknesses of example programs.

Although checklist-based assessment is a common approach for quality assurance in industry [Brykczynski 1999], it has rarely been used in educational contexts. Sanders and Thomas [2007], for example, have reviewed typical novice misconceptions to develop checklists that educators can use when designing or grading student programs.

The work presented in the present paper is based on the author's earlier work on checklist-based evaluation of example programs [Börstler et al. 2009; Börstler et al. 2008]. In the present work we have extended our detailed analysis to further reviewers and examples. Furthermore, we also also provide analyses which are different from those in our previous work. We have identified a set of characteristics and examined the examples and reviews in detail with reference to these to determine which characteristics may be influential in defining "good" examples, and we relate our results to traditional software measures.

2. IMPORTANCE OF EXAMPLES

Examples play an important role in teaching and learning programming. Students and teachers alike cite examples as the most helpful resource for learning to program [Lahtinen et al. 2005]. With respect to LISP programming, for example, Anderson et al. [1984] showed that all novices needed example programs to complete their first recursive example program. When given a choice, students generally prefer examples over written instructions for solving problems [Reed and Bolstad 1991]. Even when the example conflicts with written problem solving instructions, students typically use the example information and disregard the written instructions [LeFevre and Dixon 1986].

Examples are powerful role models; novices use examples as templates for their own work.

> What novices often do, however, is employ a knowledge-lean style of analogical reasoning, analogical transfer *within* a domain, not across domains. For instance, and most important, students use worked-out examples provided in textbooks, by the teacher, or by their peers when solving new problems. [Reimann and Schult 1996, p. 123]

Examples must therefore be consistent with the principles and rules being taught and should not exhibit any undesirable properties or behaviour. In other words, all examples should follow exactly the same principles, guidelines, and rules we expect our students to eventually learn. If our examples do not do so consistently, students will have difficulty in recognizing patterns and distinguishing an example's surface properties from those that are structurally or conceptually important. In other words, it is important to present examples in a way that conveys their "message", but

at the same time be aware of what learners might actually see in an example [Mason and Pimm 1984].

Trafton and Reiser [1993] note that in complex problem spaces (like programming), "[l]earners may learn more by solving problems with the guidance of some examples than solving more problems without the guidance of examples". By continuously exposing students to "exemplary" examples, important properties are reinforced. Students will eventually gain enough experience to recognize general patterns which helps them to distinguish between good and bad designs.

> Simply learning to perform procedures, and learning in only a single context, does not promote flexible transfer. The transfer literature suggests that the most effective transfer may come from a balance of specific examples and general principles, not from either one alone. [Bransford et al. 2004, p.77]

With carefully developed examples, we can reduce misinterpretations, premature generalizations or otherwise unintended conclusions. This helps to prevent misconceptions, which might hinder students in their further learning [Clancy 2004; Guzdial 1995; Malan and Halland 2004].

3. RELATED WORK

Textbooks are an important component of teaching introductory programming. They are a major source for example programs and also function as a reference for how to solve specific problems. Although examples play an important role in the teaching and learning of programming, there are very few systematic evaluations of textbook examples.

De Raadt et al. [2005] compared 40 introductory programming textbooks by measuring their amount of content particularly relevant for the textbook's usefulness as a learning tool, such as the number of pages covering examples, exercises, bibliographies, appendices, language reference, index, glossary, and other chapter content. In the 22 texts covering object-oriented programming (in C++, Java, Eiffel, and Delphi), examples covered from 0% to 43% of the total page counts (about 17% on average). About examples, the authors conclude that:

> Examples should concisely illustrate a technique. They should include line numbers for reference, though should preferably be as self-contained as possible, not requiring the reader to keep referring back to the accompanying text discussion. Better examples will often include the author's comments maybe accompanied with some lines and arrows like the typical classroom blackboard example.

Wu et al. [1999] analysed the examples of 16 Taiwanese high school computer textbooks (using BASIC as a programming language), covering 967 examples. Each example was categorized according to its problem type and coverage of problem solving as a process. Their analysis revealed that only 2% of the examples displayed a thorough analysis of the problem statement. Testing and debugging was discussed for only 0.2% of the examples. Only 25% of the examples dealt with real-life problems, the remaining examples were categorized as Math (27%), Graphics

(25%), Syntax (21%) and Miscellaneous (2%). The authors conclude that the material should be made "potentially meaningful to the students in order for meaningful learning to occur". Furthermore they advise a more thorough coverage of the problem solving process.

A subsequent analysis of 32 Taiwanese high school computer textbooks found that most of these problem still persisted [Lin and Wu 2007]. In particular the authors also criticize "inadequate analogies" and "dry examples".

Malan and Halland [2004] discussed the potential harm "bad" example programs might do when learning object-oriented programming. They identified four main problem areas, based on their own experiences as teachers: examples which are either too abstract or too complex and examples which apply concepts inconsistently or even undermine the concept(s) being introduced.

An important aspect of examples is *misconceptions* and how to avoid them. Holland et al. [1997] outlined a number of student problems and how they might relate to properties in example programs, such as object/class conflation, objects as simple records, and reference vs. object. Along with each problem they provide a pedagogical suggestion for avoiding potential misconceptions by choosing suitable examples.

A similar problem is noted by Fleury [2000], who discussed how students constructed their own rules by misapplying correct rules. She described certain cases in which these *student-constructed rules* can systematically lead students to incorrect solutions.

In 2001, a discussion on '`HelloWorld`'-type examples was initiated in Communications of the ACM [Westfall 2001] which created a series of follow-ups on the object-orientedness of common introductory programming examples [CACM Forum 2002; Ourosoff 2002; Dodani 2003; CACM Forum 2005]. Surprisingly, the main discussion has been centered on how to adjust the '`HelloWorld`'-example to better fulfil the characteristics of object-orientation, not on whether this is a good example at all.

Hu [2005] discusses the related problem of data-less objects (of which '`HelloWorld`' is an instance). Using data-less objects is contradictory to the basic notion of objects when introducing them to novices. He calls them merely containers holding (static) methods and simulating a procedural way of programming.

4. METHOD

Data collection and initial analysis started out as an ITiCSE working group [Börstler et al. 2009] and was based on earlier work by some of the co-authors [Börstler et al. 2008]. In the present work we have extended the pool of analysed reviews by 57% and also provide different analyses from the previous work.

An electronic checklist (described in Section 4.4) was administered for evaluating object-oriented example programs from common CS1 textbooks.

In total, we selected 38 examples from 13 introductory programming textbooks (mainly Java). Examples were nominated by working group participants, who were given the guidance that examples were preferred from textbooks which were published in 2007 or later and in the second (or later) edition, the latter being taken as an indicator of common usage. The examples were grouped into two sets:

mandatory and optional. All working group participants were required to review mandatory examples (16) and to review optional examples (22) if time permitted. A number of additional reviewers were invited to review as many examples as they were able to do. In the present paper, we further discuss only those 21 examples that received ≥3 reviews (191 reviews in total) covering 11 of the 13 textbooks. For detailed information about the full set of examples and textbooks the interested reader is referred to [Börstler et al. 2009].

4.1 Textbooks

When selecting source for examples, we aimed at a broad and representative coverage with respect to popularity, coverage, presentation style and pedagogic approach.

In order to get comparable examples, we looked for the first example in a source that exemplified a certain topic or concept. Examples were furthermore required to be *complete and clearly identifiable*, where complete means that the full code is shown together with a discussion or explanation.

Following these requirements, we had to exclude a wide range of sources. For example we excluded all sources working mainly with code snippets (incomplete code) as well as sources introducing examples gradually through successive versions (no clearly identifiable example). We also had to exclude all online tutorials we looked at, since they either had no clearly identifiable first example or lacked a discussion or explanation. Our selection of sources is therefore not fully representative.

This left us with textbooks as the only sources for examples, We classified these into two main categories: those with a clear and early focus on object-orientation (category *OO*) and those with a more traditional imperative first approach (category *Trad*). A first classification was made based on the texts' titles, back cover texts, prefaces or web pages. For a second classification, we followed VanDrunen's approach [VanDrunen 2006] and looked at each text's detailed table of contents to determine when various key concepts were introduced. In addition to that, we also looked into each text to get a feeling for actual focus and depth of a section and the style of concept presentation.

The 11 textbooks, from which the examples in the present paper are taken, are listed in Table I. One can observe a clear discrepancy between the "self-declared" approach of a text and our classification. Although most texts claim to focus on object-orientation and follow some kind of object-early or -centric approach, a closer inspection reveals that the texts in the OO category are actually in a minority. More information about the textbooks can be found in [Börstler et al. 2009].

4.2 Example Programs

To get down to a manageable but still representative set, we focused on examples with comparable properties. We considered only *complete examples* in the sense that the full source code should be present together with an explanation. Each example should furthermore be the first one in a text exemplifying certain high level concepts or ideas. To reach a good balance of varying types of examples, we categorized all examples according to the concepts they illustrate. Table II gives an overview of the number of examples per category.

FUDC: First user defined class. These examples reflect the first occurrence of a

Table I. Summary of textbooks. Column `Ex. pp.` refers to the page numbers of the examples reviewed. Entries in column `Self declared as` are quotes from the texts' back cover texts, prefaces or web pages.

					Category	
Text	*Pages*	*Edition*	Ex. pp.	Self declared as	*Self*	*Our*
[Barnes and Kölling 2009]	516	4*th*	18–22, 56–71	Objects first	*OO*	*OO*
[Bravaco and Simonson 2010]	1210	1*st*	404–408	Fundamentals first	*Trad*	*Trad*
[Deitel and Deitel 2007]	1596	7*th*	86–103	Early classes and objects pedagogy	*OO*	*Trad*
[Farrell 2010]	870	5*th*	370–375	—	—	*Trad*
[Horstmann 2008]	1204	3*rd*	85–90, 236–241, 438–442	Objects gradual	?	*OO*
[Lewis and Loftus 2009]	832	6*th*	190–194, 241–243, 515–527	True object-orientation	*OO*	*Trad*
[Malik and Burton 2009]	1018	1*st*	185–192	Objects early but gently	*OO*	*Trad*
[Niño and Hosch 2008]	1040	3*rd*	85–92, 123–135	Objects first	*OO*	*OO*
[Riley 2006]	769	2*nd*	263–265, 386–395	Object centric	*OO*	*OO*
[Roberts 2008]	587	2*nd*	190–198, 332–338	Modern objects first approach; class hierarchies from the beginning	*OO*	*Trad*
[Wu 2008]	987	5*th*	156–162, 309–310	Objects first	*OO*	*Trad*

user defined class in a text. We consider these examples particularly important, since they "set the stage" for how students are expected to think about object-oriented class design.

OOD: Multiple user defined classes. Examples in this group exemplify some kind of design decision/strategy. 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.

CS: Control structures. The main purpose of the examples in this group is to exemplify the usage of control structure (selection and repetition). One could argue that object-orientedness does not matter in this category, since this is not the purpose of the example. However, these examples are interesting as they demonstrate the authors' approach to teaching language syntax within the context of OOP.

Note that for the OOD and CS categories examples generally reflected the first occurrence of a particular topic, for example loops, and not necessarily the first example within the category, to avoid skewing the selection towards particular topics within each category.

4.3 Reviewer Demographics

The reviews were performed by experienced educators from a diverse range of institutions in five countries (Denmark, Germany, Sweden, UK, and USA). On average reviewers have more than 10 years of experience with teaching object-orientation specifically. In addition to that, several reviewers also have considerable profes-

Table II. Number of Examples chosen per book category.

Example Category	*Text category* *OO*	*Trad*	$\sum$
First user defined class (FUDC)	4	5	9
Multiple user defined classes (OOD)	4	4	8
Control structures (CS)	2	2	4
$\sum$	10	11	21

Table III. Characterization of reviewers. `Years of Practical OO experience` refers to work experience such as "professional trainer", "researcher", "professional programmer" or "systems/software engineer". `Novice courses` are the number of OOP classes for novices taught in the last 10 years.

Reviewer	*Years of OO experience* *Practical*	*Teaching*	*Novice courses*	Comments
*R*1	0	5	$1-3$	Works in teacher education
*R*2	3	9	$7-9$	
*R*23	11	10	$7-9$	
*R*4	4	15	$4-6$	Dealt with OOD in PhD
*R*41	17	15	≥ 10	
*R*5	0	8	≥ 10	Textbook author (OOP/Java)
*R*6	5	20	$7-9$	Dealt with OO notation in PhD
*R*7	0	11	≥ 10	
*R*71	0	2	$1-3$	
*R*72	4	12	$7-9$	Background as professional programmer
*R*8	> 0	9	≥ 10	20+ years of professional programming

sional experience with object-orientation, for example as a researcher or a professional programmer. Most reviewers teach or have taught object-orientation to novices many times, some of them are doing so more than once a year. This work is focussed on the quality of examples for teaching object-oriented programming to novices and among other things reviewers were asked specifically to rate examples for object-oriented quality. A summary of the background data of the 11 reviewers contributing ≥ 4 reviews can be found in Table III.

4.4 The Checklist

Our checklist comprises 10 quality factors that we grouped into three quality categories; technical quality, object-oriented quality, and didactic quality.

Technical qualities (T1–T2) capture properties that are independent of a particular programming paradigm, such as correctness and readability:

—Correctness and Completeness (T1): The code is bug free and the example is sufficiently complete.

—Readability and Style (T2): The code is easy to read and follows a consistent formatting and style.

Object-oriented qualities (O1–O5) capture commonly accepted principles, guidelines and heuristics of object-oriented design and programming:

—Reasonable Abstractions (O1): Abstractions are plausible from an OO modelling perspective as well as from a novice perspective.

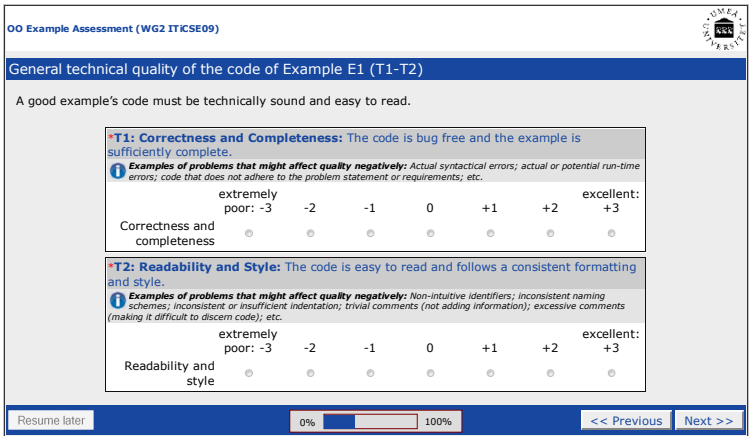


Fig. 1. Example survey screen (Technical quality).

—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 OO program as a collection of collaborating objects.

Didactic qualities (D1–D3) capture properties related to the understandability of the example, its discussion and development:

—Sense of Purpose (D1): Students can relate to the example's domain and computer programming seems a reasonable way to solve the problem.

—Process (D2): An appropriate programming process is followed/described.

—Well Balanced Cognitive Load (D3): Explanations and supporting materials promote comprehension; they are neither simplistic, nor do they impose extraneous cognitive load.

For each quality factor we also provided a list of typical problems, which were distilled from the literature on student problems or misconceptions and object-oriented design principles, guidelines and rules (see for example [Fowler 1999; Nordström 2009; Riel 1996]).

Each quality factor was assessed on a 7-point Likert-type scale from −3 (extremely poor) to +3 (excellent) using the electronic survey instrument LimeSurvey [LimeSurvey]. Figure 1 shows an example screen for the questions in the category Technical Quality. The interested reader is invited to test the instrument at `http://www.cs.umu.se/proj/limesurvey/`.

In addition to the 10 quality factors described above, we also asked reviewers for their overall impression of example quality before and after the actual assessment. In two final open questions, reviewers could provide additional comments regarding

Table IV. Number of reviews per reviewer.

Reviewer	*Reviews*	*in%*
*R*1	16	7.44
*R*2	38	17.67
*R*23	5	2.33
*R*4	22	10.23
*R*41	4	1.86
*R*5	18	8.37
*R*6	18	8.37
*R*7	21	9.77
*R*71	11	5.12
*R*72	10	4.65
*R*8	16	7.44
All other 14 *reviewers*	36	16.74
$\sum$	215	100

Table V. Number of reviews per example.

Reviews	*Examples*
12	$E1, E2, E13$
11	$E11, E12, E16$
10	$E3, E4, E6, E9, E10, E14, E15$
9	$E5, E7, E8$
8	$E21, E26$
3	$E19, E20, E25$
1–2	$E17, E18, E22$–$E24, E27$–$E38$

example quality and the questionnaire itself. The instrument is discussed in more detail in [Börstler et al. 2009].

5. RESULTS

In total, we received 215 valid reviews by 25 individuals performing between 1 and 38 reviews each. Of the 25 reviewers, 9 reviewers submitted $\geq$10 reviews and 11 reviewers submitted $\geq$4 reviews. Of the 38 examples, 21 received $\geq$3 reviews each (191 reviews in total). Of those 191 reviews, 47 (roughly 25%) were contributed by people involved in checklist design. Details of reviews per reviewer and reviews per example can be found in Table IV and Table V, respectively.

Table VI summarizes the main results for all examples with $\geq$8 reviews (examples in parentheses have 3 reviews). Considering our Likert-type scale from -3 (extremely poor) to $+3$ (excellent), we get total average scores in the range $[-30, 30]$. Since we only considered examples from popular textbooks, we would expect most of the scores in the upper positive range. However, as many as 10 out of the 21 examples in Table VI scored below 10 and received an overall final impression (I2) $\leq$0.

The average ratings for overall impression before and after the actual review (I1 and I2, respectively) further corroborate this impression. Only 8 examples received an overall final impression $\geq$1 and as many as 10 examples were rated $\leq$0. Interestingly, the overall impression seems to degrade during the review, in particular for the examples that already have a low overall first impression (I1) (see also Table VII). This indicates that the checklist might help to spot problems that might be easily overlooked.

Table VI. Summary of the main results for all examples with ≥ 3 reviews (191 reviews in total). Ranking is top down according to average total score for the 10 quality factors T1–D3 (column `Score` $\in [-30, 30]$). Columns `EC` and `BC` list example (as defined in Section 4.2) and book category (as defined in Table I), respectively. Columns `I1` and `I2` list the average score for the overall impression ($\in [-3, 3]$) of example quality before review start and after finishing the review, respectively.

	EC	*BC*	*Score*	*I1*	*I2*	*I2 − I1*
*E*26	*OOD*	*OO*	23.88	2.00	2.13	0.13
*E*9	*OOD*	*OO*	21.50	1.90	1.80	−0.10
*E*21	*FUDC*	*Trad*	19.63	1.75	1.75	0.00
*E*7	*OOD*	*OO*	18.00	1.11	0.78	−0.33
*E*3	*FUDC*	*OO*	17.00	1.00	1.10	0.10
(*E*20)	*FUDC*	*Trad*	16.67	1.00	1.33	0.33
*E*2	*FUDC*	*OO*	16.42	0.92	0.83	−0.09
*E*12	*OOD*	*Trad*	16.00	1.55	1.36	−0.19
*E*16	*CS*	*OO*	15.00	1.46	1.09	−0.37
*E*1	*FUDC*	*OO*	14.17	1.00	1.00	0.00
*E*5	*FUDC*	*Trad*	13.33	0.56	0.56	0.00
*E*6	*FUDC*	*Trad*	9.90	0.20	−0.20	−0.40
*E*4	*FUDC*	*Trad*	9.90	0.10	−0.60	−0.70
(*E*19)	*FUDC*	*Trad*	7.00	0.00	−0.67	−0.67
*E*10	*OOD*	*Trad*	6.80	0.40	0.00	−0.40
*E*8	*OOD*	*Trad*	6.22	0.11	−0.33	−0.44
*E*11	*OOD*	*OO*	4.55	0.27	−0.18	−0.45
*E*14	*CS*	*Trad*	0.70	0.20	−0.60	−0.80
*E*13	*CS*	*OO*	−1.08	−1.17	−1.17	0.00
(*E*25)	*OOD*	*Trad*	−2.33	−1.33	−1.67	−0.34
*E*15	*CS*	*Trad*	−2.60	−1.50	−1.80	−0.30

Table VII. Changes in overall impression of quality from initial impression `I1` (before starting the review) to final impression `I2` (after finishing the review).

Steps	*Count*	%
≤ -3	4	1.87
−2	12	5.58
−1	48	22.33
0	117	54.42
1	29	13.49
2	5	2.33
≥ 3	0	0.00
$\sum$	215	100
Total negative	64	29.77
Total unchanged	117	54.42
Total positive	34	15.81

The examples in Table VI can be grouped into 4 groups, where the differences in final overall impression (I2) are smaller between the items in a group than between groups (the groups are set apart by horizontal lines in the table).

Note that these groups did not change compared to our previous analysis[1] [Börstler et al. 2009], although we now have 191 data points compared to 122 before. This indicates that our evaluation instrument is quite reliable. This is also corroborated by the high inter-rater agreement (see Table VIII in Section 6).

[1]Except for E10, which moved from the second group (where it was last) to the third group.

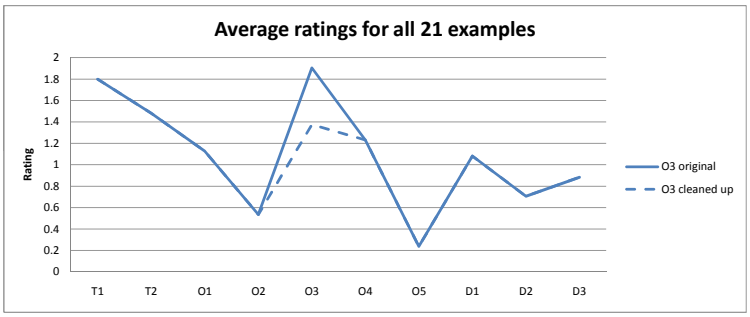


Fig. 2. Average ratings of all quality factors for the 191 reviews.

Figure 2 shows the average ratings for all quality factors. The peak for O3 (Reasonable Class Relationships) might be attributed to our rating instructions to consider this factor as excellent (i.e. +3) "when no relationships are present and the example doesn't call for any". When deleting all examples consisting of a single class only, O3 still has the third largest average (1.38) after the technical quality factors T1 (1.80) and T2 (1.48).

6. DISCUSSION

6.1 Agreement between raters

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 (see Table VIII). This strongly indicates that our evaluation instrument is reliable.

The only outliers are reviewers R1 and R23. For R1 the deviation could be a result of this reviewer's different background (see Table III). R1 is the only reviewer with a background in teacher education, whereas all others have a computer science background. Furthermore, R1 is less experienced than most other reviewers. For reviewers R23 and R41, the data includes only 5 and 4 data points, respectively. Their rho-values can therefore be at best be interpreted as indications.

6.2 Differences between Textbook Categories

In our previous work ([Börstler et al. 2009]), we noticed that the examples from textbooks categorized as OO on average scored somewhat higher than the ones from textbooks categorised as Trad. With the extended data in the present analysis this is still true, and the overall difference is still at the same level. Based on their average total score (see Table VI), OO-type book examples have an average rank of 8.2, examples from Trad-type books an average rank of 13. However, one should note that there are examples from both book categories among the examples in the top group as well as among the examples in the bottom group.

Since OO-type books have a clear and early focus on object-orientation (see our definition in Section 4.1) it is not surprising that their examples score higher on a scale that is partly based on object-oriented properties. For Trad-type texts it could be argued that object-oriented qualities are not equally important for all types of

Table VIII. Spearman rank correlation (`rho`) for reviewer's scores and the total average score for all reviews. R23 and R41 had too few reviews to compute meaningful p-values.

Reviewer	*Rho*	*P-value*
*R*1	0.298	0.263
*R*2	0.922	2.80*E*-8
*R*23	−0.290	–
*R*4	0.895	9.68*E*-8
*R*41	0.892	–
*R*5	0.633	0.005
*R*6	0.582	0.011
*R*7	0.899	4.04*E*-7
*R*71	0.904	1.31*E*-4
*R*72	0.876	< 0.01
*R*8	0.883	5.85*E*-6

examples. The three different types of examples (*FUDC: First User-Defined Class*, *OOD: Multiple User Defined Classes*, *CS: Control Structures*) are included in the books to illustrate significantly different concepts. FUDC examples may illustrate classes, objects, attributes, methods and encapsulation, while OOD examples may exemplify any of association, collaboration, message passing, inheritance and polymorphism. The programming concepts in CS examples, on the other hand, are not explicitly object-oriented. We will discuss this issue in more detail in the following sections.

6.3 Dependencies between Quality Factors

Our previous results ([Börstler et al. 2009]) indicated that the different quality factors seem to capture different aspects of quality. There were examples that consistently scored high or low on all quality factors and also examples without consistent scoring patterns.

Our current data shows only weak correlations between quality factors over all examples. The Spearman rank correlation for all quality factors lies in the range $[-0.28, 0.35]$, except for two slightly higher values of 0.48 (O1 vs. O2) and 0.76 (O4 vs. O5) (p-values in the range $[0.01, 0.025]$).

When looking at the quality factors O1–O5, in particular, we can see interesting differences between the example categories (Figure 3). For CS and OOD examples, example rankings are almost the same for all five quality factors. For the category CS, the only deviation is for O3 which is concerning class relationships. Those small examples of control structures rarely include any relationships and that is most likely to considered reasonable. Furthermore, all ratings are close to the average rating for O1–O5[2].

The variation for FUDC examples is much greater than the variation for CS and OOD examples. We can find a similar pattern of variation between ratings for the different example categories for technical quality (T1–T2) and didactic quality (D1–D3), but much less pronounced.

[2]The differing behaviour on O3 is an artefact of our rating instructions as explained at the end of Section 5.

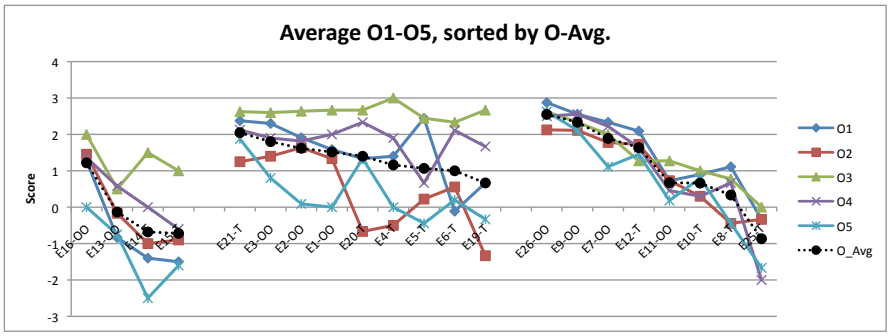


Fig. 3. Average ratings on O1–O5 for all examples, grouped by example category; CS, FUDC, and OOD (from left to right).

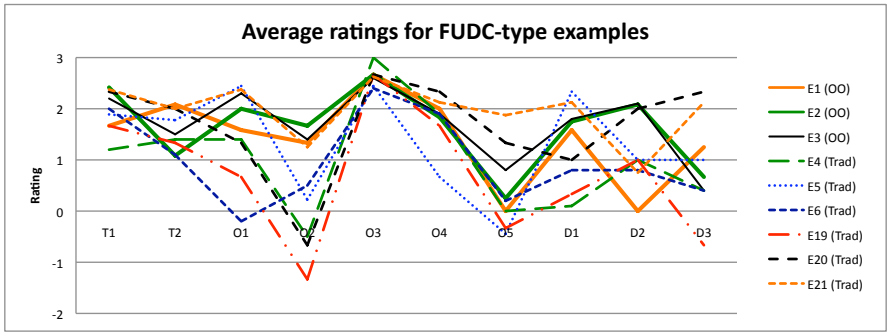


Fig. 4. Average ratings of all examples exemplifying the first user-defined class (FUDC).

We conclude from this analysis that good FUDC examples are more difficult to find or develop than the other types of examples we have looked at. This is hardly surprising, since the demands on a FUDC example seem more difficult to satisfy. It should be a role model for a "good" object-oriented class, i.e. score well on the characteristics captured by O1–O5. However, it should also be small and use as few concepts as possible, since most concepts have not been introduced yet.

Quality factors O1–O5 have been more closely examined by example category in Section 6.4–6.6.

6.4 Object-oriented quality of FUDC-examples

An example of a user-defined class should make it clear why it is necessary to create that class and why it needs to have particular state and behavior. Figure 4 shows the average ratings for O1–O5 for FUDC examples. Here we consider which quality factors indicate how successfully the examples achieve this and what characteristics of the examples may influence ratings for these quality factors. O1 (Reasonable Abstractions) and O2 (Reasonable State and Behaviour) relate closely to these requirements. O1 is likely to be influenced as much by the "cover story" (the text which introduces the context of the example) as by the code and is not considered in

Table IX. Book category , number of fields, default constructor (Def.), number of constructors (Cnstr.), number of methods (Meth.), way(s) in which class is exercised (Ex.) and number of objects instantiated (Obj.), average scores for quality factors O2 and O5, and average total score (TOD) for FUDC examples. Possible values for Ex. are IDE (interactive instantiation in IDE), TC (test class) and TM (test method).

	Book	*Fields*	*Def.*	*Cnstr.*	*Meth.*	*Ex./Obj.*	*O2*	*O5*	*TOD*
E2	*OO*	1	*y*	1	1	*IDE*/1	1.64	0.09	16.42
E3	*OO*	1	*n*	1	2	*IDE&TC*/1	1.40	0.80	17.00
E1	*OO*	3	*n*	1	2	*IDE*/2	1.33	0.00	14.17
E21	*Trad*	1	*y*	1	1	*TC*/2	1.25	1.88	19.63
E6	*Trad*	2	*y*	2	1	*TC*/2	0.56	0.20	9.90
E5	*Trad*	4	*n*	1	0	*TM*/1	0.22	−0.44	13.33
E4	*Trad*	1	*y*	2	0	*TC*/2	−0.50	0.00	9.90
E20	*Trad*	1	*y*	1	0	*TC*/2	−0.67	1.33	16.67
E19	*Trad*	1	*n*	1	0	*TC*/2	−1.33	−0.33	7.00

this discussion. O5 (Promotes "Object Thinking") presents an interesting problem for authors—how can single-class examples promote object thinking? The way in which instances of the class are created and exercised is important here. The data for O3 (Reasonable Class Relationships) has clearly been influenced by the instruction that examples with no relationships should be given score of 3, and this data is not considered here.

Neither is O4 (Exemplary OO code)—this quality factor does not appear to discriminate strongly between the examples and suggests that these are generally free from "code smells".

The following characteristics of each example were identified by examining the code listings:

—number of fields/attributes – in all examples the fields are appropriately implemented as private with accessors and mutators as required depending on whether the field is read-only or read-write

—provision of a default constructor

—total number of constructors, including the default constructor if provided

—number of methods – this number excludes accessors/mutators and `toString` implementations, and also excludes methods which play one of these roles (for example, a method which simply prints out a message containing the value of a field)

—the way in which the class is exercised – all the examples provide an example of creating and using instances, either in the form of a separate test class or by interactive instantiation in an IDE such as BlueJ or Dr Java. The only exception provides a test method which acts upon an instance of the UDC, but does not provide a full working example to show how the method could be implemented.

Table IX shows the above information for each of the examples together with: book category; average score for O2; average score for O5; average total score. The results are shown in order of average O2 score.

It is strikingly clear from Table IX that the books which we have categorized as OO attain higher scores for O2 than books in the Trad category. This is not the case for O5, however. Interestingly, the books in our OO category all choose to

exercise classes interactively in an IDE while the Trad books simply provide test classes.

The 'size' of the class in terms of the number of fields does not appear to influence "Reasonable state and behavior"—the examples with the highest and lowest O2 scores each have a single field. Constructors also have little influence—some reviewers have made negative comments where examples do not include a default constructor, but this is not particularly reflected in the scores. The most significant code feature for O2 appears to be the methods provided. Examples with only accessors/mutators and print operations score poorly, not surprisingly as this issue is clearly identified in the examples of problems given in the checklist. What is perhaps surprising is that four out of the nine examples presented a first user defined class with no meaningful behavior. This may promote misconceptions about the purpose of classes. It is interesting to note that the ranking by O2 score is quite different from the ranking by total score. E20 in particular has a good total score but a very poor O2 score. The comments for that example explicitly identify the limited behavior but excuse this—for example *"perhaps this example does not lend itself to behaviour which can be modeled"*.

The ranking by O5 score is quite different from the ranking by O2, and again class size and constructors do not have any clear influence. The only common factor which can be observed is that examples with only accessors/mutators and print operations score poorly. Scores for "Promoting object thinking" do not appear to be strongly influenced by the way in which classes are exercised. Test classes and interactive instantion in an IDE appear to be equally acceptable, although the lowest score is attained by the example which shows an incomplete test class. There is, though, a wide range in the scores for this quality factor. This suggests that factors other than the code itself are influential here.

It is difficult to draw firm conclusions on what makes an "exemplary" FUDC example. It does appear, perhaps not surprisingly, that examples which illustrate meaningful behavior display greater object-oriented quality than examples which include no or trivial behavior.

6.5 Object-oriented quality of OOD examples

It might be expected that examples in the OOD category should score highly for object-oriented quality. An object-oriented program is in essence a group of collaborating classes, and a well designed example should illustrate collaborations and the programming structures which allow these collaborations to occur. The examples reviewed were generally among the first multiple class examples in each book.

These examples have been examined further to ascertain any clear influence on scores for object-oriented quality of following factors:

—number of classes in the example

—type(s) of relationship represented in the example

The number of classes in these examples ranges from 2 to 6. Some examples exercise the classes using a client class, while others either do not provide a way to exercise the classes, or do so using the capability of the IDE to instantiate and interact with objects. Client classes are not included here in the number of classes listed here.

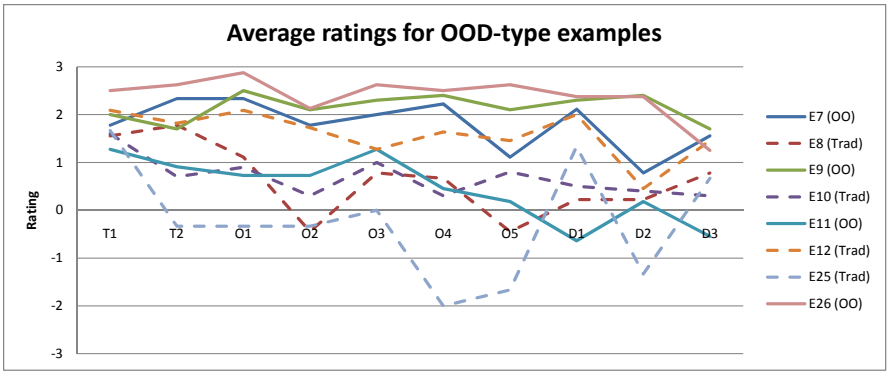


Fig. 5. Average ratings of all examples exemplifying object-oriented design involving several classes (OOD).

A class relationship is a binary association between classes in the example. Each example may include one or more binary associations. The types of relationships represented have been grouped into three categories which we define as:

IN: Inheritance. This relationship indicates specialization, and is realised by the implementation of one class as a subclass of another.

CO: Composition. This relationship indicates containment, and is realized by the presence of a field in one class having a field which holds a reference to one or more instances of another class. This includes relationships commonly described as composition or aggregation, or as a "has-a" relationship.

AS: Association. This indicates a transient relationship, where one class has a reference to one or more instances of another class as a method parameter, method return value or local variable. This includes relationships commonly described as association or dependency, or as a "uses-a" relationship.

Note that under these definitions, a composition and association relationships provides means for objects to collaborate, while inheritance represents a static structural relationship.

Table X shows the following information for each of the examples: number of classes; average score for O3 (Reasonable Class Relationships); average score for O5 (Promotes "Object Thinking"); average total score. Of all the quality factors, O3 and O5 are likely to have been most strongly influenced by the representation of class relationships. O3 is likely to have been considered in the way it was intended to be for these particular examples by reviewers as there clearly are relationships present. The results are shown in order of average O3 score. It is apparent that the examples which score well for object-oriented quality factors also score well overall—it is not only their representation of relationships which makes them good examples.

The number of classes does not appear to have any influence on the results. The highest and lowest rated examples both contain only two classes. It is clearly possible to create examples which provide a useful illustration of the relationship between a pair of classes. Examples with more classes can illustrate a more complex

Table X. Book category, Number of classes (Cl.), class relationship types represented (Rel.), average scores for quality factors related to relationships and average total score (TOD) for OOD examples. **In E10 two additional subclasses are discussed but the code for these is not listed.*

	Book	*Cl.*	*Rel.*	*O3*	*O5*	*TOD*
*E*26	*OO*	2	*CO*	2.63	2.63	23.88
*E*9	*OO*	2	*AS*	2.44	2.22	22.44
*E*7	*OO*	2	*IN*	1.88	1.25	18.38
*E*12	*Trad*	6	*IN, CO*	1.50	1.50	16.50
*E*11	*OO*	3	*IN*	1.30	0.20	4.90
*E*8	*Trad*	2	*IN*	1.00	−0.38	6.63
*E*10	*Trad*	3∗	*IN, CO*	1.00	0.67	6.33
*E*25	*OO*	2	*IN*	0.00	−1.67	−2.33

system of collaborations, but need to be designed carefully so that the details and significance of each collaboration are not lost.

The types of relationships represented does appear to be significant. The two best examples illustrate composition and association respectively, while the lowest rated example illustrates a simple two-class inheritance hierarchy. Examples of composition and association are able to exemplify collaboration between objects at runtime, which is helpful in promoting object-thinking. While inheritance is an important concept, it does not by itself reflect object collaboration. Two of the examples combine inheritance with composition to illustrate polymorphism. It might be expected that showing inheritance in a typical collaborative context would lead to better examples than demonstrating it in isolation, but this is not clearly reflected in the reviewers' scores. The best example of inheritance focuses clearly on the issues of additional state and behavior, in contrast to another example which only considers additional data members in a subclass. The lowest rated example also deals with state and behavior, but comments suggest that it is a rather confusing example. Some reviewers commented that examples of inheritance were based on scenarios which would be better modelled as roles and were therefore not exemplary illustrations. This issue would be reflected in the O5 score and may be significant for E8 in particular.

It is clearly possible to create exemplary examples of multiple collaborating classes, and such examples illustrate fundamental concepts in object-oriented programming. The results suggest that early examples in this category should adopt a clear focus on illustrating a specific type of relationship. Examples which develop more complex structures can then build on the simple examples or can be constructed from smaller clearly focused examples of collaborations.

6.6 Object-oriented quality of CS examples

Figure 6 shows that, with one exception, the control structure examples scored poorly on object-oriented quality. As was the case for FUDC examples, the data for O3 (Reasonable class relationships) has probably been artificially influenced by the instructions given in the checklist and is disregarded.

Control structures are fundamental programming concepts and are not a feature of any particular paradigm. The authors of these books follow quite different

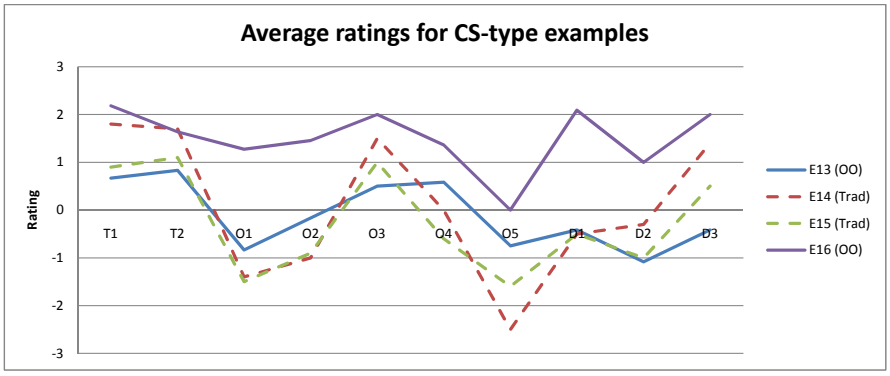


Fig. 6. Average ratings of all examples exemplifying control structures (CS).

approaches to introducing control structures:

—E14 (Trad book category) – shows a single class which consists entirely of a main method. This is an example of the data-less object described by Hu [2005]. It scores poorly, particularly for O5 (Promotes object-thinking).

—E15 (Trad) – a single class which contains a main method which instantiates an instance of the class and calls a "start" method. One other method is included which is used by the start method. This example also scores poorly—scores are very similar to E14 except for a slightly higher O5 score. The inclusion of the main method in the single class is noted by one reviewer as bad practice, and contrasts with the way in which main methods are used appropriately in test classes in many of the FUDC examples.

—E13 (OO) – presents an if-else structure within a method to handle a mouse button click. It scores poorly, but marginally better for object-oriented quality than E14 and E15. Reviewers comments note the dependence of the example on code presented elsewhere in the book and the complexity of this code.

—E16 (OO) – structured in a similar way to most FUDC examples. It shows a class which represents a clear abstraction and includes relevant state and behavior, and includes a separate test class. The methods provide motivations for the use of control structures. This example received the highest scores among the CS examples for object-oriented quality. The score for O5 is low, although it was noted in section 6.4 that O5 scores may depend strongly on factors other than the example code.

These results suggest that reviewers regard the use of control structures as a way of implementing behavior within meaningful class methods as exemplary in this category.

6.7 Relation to traditional software measures

We also investigated the relationship of total average scores with traditional measures of software quality. In Table XI we have summarized the results from comparing the total average score of our instrument with the measures described below. Most of these measures have been obtained using the measurement tool

Table XI. Spearman rank correlation (Rho) for common software measures and the total average score of our 21 examples.

Measure	*Rho*	*P-value*
Size	−0.080	0.729
Maintainability Index	0.494	0.023
Complexity	−0.138	0.550
Code density	0.557	0.009
Comment density	0.498	0.022

JHawk [JHawk]. Since most of our examples consisted of only a single class, we did not collect any object-oriented measures.

—*Size:* The total lines of code, counting code, comments and empty lines. Although size is a very simple measure it tends to correlate with many software attributes.
—*Maintainability Index (MI):* The Maintainability Index is a measure for predicting the relative effort for software maintainability [Welker et al. 1997]. Since maintainability requires easy to read and to understand code, MI intends to capture these properties.
—*Complexity:* McCabe's cyclomatic complexity measures the number of (statically) distinct paths through a method [McCabe 1976].
—*Code density:* The number of statements divided by Size (see above).
—*Comment density:* The number of comments divided by Size (see above).

Our checklist captures different aspects of quality from traditional measures of software quality. However, since MI, code density and comment density all capture different aspects of readability and understandability, we would expect some kind of relationship to our ratings. Looking at the scatterplots in Figure 7, we can see some indications for such relationships:

—Examples with dense code tend to have worse overall average ratings. The best rated examples tend to have a code density between 0.2 and 0.3. If we look at examples in the two top and two bottom "quartiles" (see Table VI), we can see that 10 of the 11 better examples have a code density below 0.3, whereas 7 of 10 of the worse examples have a code density above 0.3.
—The picture for comment density is somewhat similar. The better examples tend to have more comments, but this relationship is much less pronounced than for code density.
—Although all examples have MI-values above 85, indicating "high maintainability" [Welker et al. 1997], there is a clear difference between the two top and two bottom "quartiles". Nine of the 11 better examples have MI $\geq$ 130, whereas 7 of 10 of the worse examples are beyond this level.

The actual correlations are, however, only moderate. This indicates that our measure does indeed capture different aspects of quality from traditional software measures.

7. CONCLUSIONS

In this paper we analysed 191 reviews of 21 object-oriented example programs from 11 introductory object-oriented programming texts. The reviews were done

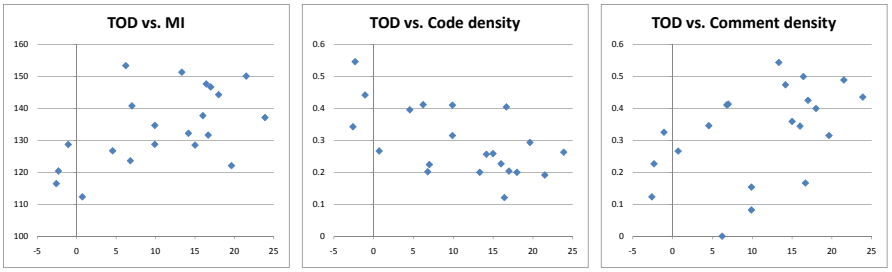


Fig. 7. Scatterplots for the 21 examples' average total scores (TOD) against Maintainability Index (MI), Code density, and Comment density, respectively.

by 11 reviewers from Denmark, Germany, Sweden, the UK, and the USA. The review instrument comprised 10 quality factors, grouped into three quality categories: technical quality (T1–T2), object-oriented quality (O1–O5), and didactic quality (D1–D3).

Our results show that the examples in the analysed sample varied markedly in quality. The object-oriented quality of many examples is not as high as one would expect to find in an introductory programming text. In particular, many examples received low ratings for "object thinking" (O5) and Reasonable state and behaviour (O2). Since examples are the most important tools for learning, these results are alarming. High quality examples are a prerequisite for successfully learning a new skill.

We looked at three different categories of examples (FUDC, OOD, and CS[3]) and two different categories of texts (OO-type and Trad-type[4]).

Our analysis revealed different rating patterns for FUDC examples compared to CS and OOD examples indicating that it is specifically difficult to develop FUDC examples with consistent high ratings for all object-oriented quality factors. Of the 4 CS examples we analysed, 3 are among the bottom 4 examples. This was not surprising, since such examples are said to be impossible to do in an "object-oriented way". However, one of these examples was rated above average, indicating that it actually is possible.

Although examples from OO-type texts receive somewhat higher ratings on average than examples from Trad-type texts, taking an example from an OO-type text is no guarantee for high ratings in object oriented-quality. Examples with high ratings in object-oriented quality can be found in Trad-type textbooks as well as examples with low ratings in OO-type texts.

Our review instrument is highly reliable and measures aspects of quality that are not captured by common size or complexity measures. It can be a useful tool for identifying problems in example programs that might otherwise go unnoticed.

[3]The first example of a user defined class in the text (FUDC), the first example of an object-oriented design featuring multiple classes (OOD), and the first example of control structures (CS).

[4]Texts with a clear and early focus on object-orientation (OO-type) and others (Trad-type).

REFERENCES

Anderson, J., Farrell, R., and Sauers, R. 1984. Learning to program in LISP. *Cognitive Science 8,* 2, 87–129.

Barnes, D. J. and Kölling, M. 2009. *Objects First with Java*, 4th ed. Prentice Hall.

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

Börstler, J., Christensen, H. B., Bennedsen, J., Nordström, M., Westin, L. K., Moström, J. E., and Caspersen, M. E. 2008. Evaluating oo example programs for cs1. In *ITiCSE'08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education*. 47–52.

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,* 4, 126–143.

Bransford, J. D., Brown, A. L., and Cocking, R. R. 2004. *How People Learn, Expanded Edition*. National Academy Press, Washington, D.C., USA.

Bravaco, R. and Simonson, S. 2010. *Java Programming – From the Ground Up*, 1st ed. McGraw-Hill.

Brooks, R. 1983. Towards a theory of the comprehension of computer programs. *Intl. Journal of Man-Machine Studies 18,* 6, 543–554.

Brykczynski, B. 1999. A survey of software inspection checklists. *ACM SIGSOFT Software Engineering Notes 24,* 1, 82–89.

Burkhardt, J., Détienne, F., and Wiedenbeck, S. 2002. Object-oriented program comprehension: Effect of expertise, task and phase. *Empirical Software Engineering 7,* 2, 115–156.

CACM Forum. 2002. 'Hello, World' gets mixed greetings. *Communications of the ACM 45,* 2, 11–15.

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

Clancy, M. 2004. Misconceptions and attitudes that infere with learning to program. In *Computer Science Education Research*, S. Fincher and M. Petre, Eds. Taylor & Francis, Lisse, The Netherlands, 85–100.

de Raadt, M., Watson, R., and Toleman, M. 2005. Textbooks: Under inspection. Tech. rep., University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia.

Deitel, H. M. and Deitel, P. J. 2007. *Java – How to Program*, 7th ed. Prentice Hall.

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

Farrell, J. 2010. *Java Programming*, 5th ed. Thomson.

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

Fowler, M. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc.

Guzdial, M. 1995. Centralized mindset: A student problem with object-oriented programming. In *Proceedings of the 26th Technical Symposium on Computer Science Education*. 182–185.

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

Horstmann, C. S. 2008. *Big Java*, 3rd ed. Wiley.

Hu, C. 2005. Dataless objects considered harmful. *Communications of the ACM 48,* 2, 99–101.

JHawk. Product homepage. `http://www.virtualmachinery.com/jhawkprod.htm`, last visited 2009-11-03.

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*. 14–18.

LeFevre, J. and Dixon, P. 1986. Do written instructions need examples? *Cognition and Instruction 3,* 1, 1–30.

Lewis, J. and Loftus, W. 2009. *Java – Software Solutions*, 6th ed. Addison-Wesley.

LimeSurvey. Project homepage. `http://www.limesurvey.org/`, last visited 2009-10-20.

Lin, J. M.-C. and Wu, C.-C. 2007. Suggestions for content selection and presentation in high school computer textbooks. *Computers & Education 48,* 3, 508–521.

Magel, K. 1982. A Theory of Small Program Complexity. *ACM SIGPLAN Notices 17,* 3, 37–45.

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*. 83–87.

Malik, D. and Burton, R. P. 2009. *Java Programming – Guided Learning with Early Objects*, 1st ed. Course Technology.

Mason, J. and Pimm, D. 1984. Generic Examples: Seeing the General in the Particular. *Educational Studies in Mathematics 15,* 3, 277–289.

McCabe, T. 1976. A complexity measure. *IEEE Transactions on Software Engineering 2,* 4, 308–320.

Niño, J. and Hosch, F. A. 2008. *Introduction to Programming and Object Oriented Design Using Java*, 3rd ed. Wiley.

Nordström, M. 2009. He[d]uristics—heuristics for designing object oriented examples for novices. Ph.D. thesis, Umeå University, Umeå, Sweden.

Ourosoff, N. 2002. Primitive types in Java considered harmful. *Communications of the ACM 45,* 8, 105–106.

Purao, S. and Vaishnavi, V. 2003. Product metrics for object-oriented systems. *ACM Computing Surveys 35,* 2, 191–221.

Reed, S. and Bolstad, C. 1991. Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition 17,* 4, 753–766.

Reimann, P. and Schult, T. J. 1996. Turning examples into cases: Acquiring knowledge structures for analogical problem solving. *Educational Psychologist 31,* 2, 123–132.

Riel, A. J. 1996. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA.

Riley, D. D. 2006. *The Object of Java*, 2nd ed. Addison-Wesley.

Roberts, E. 2008. *Java – An Introduction to Computer Science*, 2nd ed. Addison-Wesley.

Sanders, K. and Thomas, L. 2007. Checklists for grading object-oriented cs1 programs: Concepts and misconceptions. In *Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*. 166–170.

Trafton, J. G. and Reiser, B. J. 1993. Studying examples and solving problems: Contributions to skill acquisition. Tech. rep., Naval HCI Research Lab, Washington, DC, USA.

VanDrunen, T. 2006. Java interfaces in CS 1 textbooks. In *OOPSLA Conference Companion – 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 875–880.

Welker, K. D., Oman, P. W., and Atkinson, G. G. 1997. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice 9,* 3, 127–159.

Westfall, R. 2001. 'Hello, World' considered harmful. *Communications of the ACM 44,* 10, 129–130.

Wu, C.-C., Lin, J. M.-C., and Lin, K.-Y. 1999. A content analysis of programming examples in high school computer textbooks in taiwan. *Journal of Computers in Mathematics and Science Teaching 18,* 3, 225–244.

Wu, C. T. 2008. *A Comprehensive Introduction to Object-Oriented Programming with Java*, International ed. McGraw-Hill.

Received February 2010; accepted June 2010.

Paper V

Educators' views on object orientation

Computer Science Education
Vol. 00, No. 00, ?? 2010, 1–16

RESEARCH ARTICLE

Educators' views on Object orientation

Marie Nordström*

Department of Computing Science, Umeå University, S-091 87 Umeå, Sweden

(*v1.0 released november 2010*)

Much of the research on the teaching of object orientation has been focused on the students and their learning. Less is known of how the educators themselves think about different issues of the paradigm. The personal view of the educator is an important aspect that will affect how object orientation is taught. To investigate this, a qualitative study on educators' views on object orientation has been conducted and categories of views concerning object orientation, objects, and examples for object orientation defined. In all, ten educators have been interviewed, six teaching in upper secondary school and four teaching at university-level. The results indicate that educators have a simple conceptual model of object orientation, which is likely to affect the presentation of the paradigm.

Keywords: Educators, Novices, Object orientation

1 Introduction

An important aspect of teaching introductory object oriented programming to novices, is to convey the general ideas of the paradigm. Since the presentation of the paradigm is likely to be based upon the personal views of the educator, it is important to know how educators think about the paradigm. What do we know about the way educators themselves think about object orientation? What conceptual models are the basis for their teaching? There are, to our knowledge, no studies on this perspective of teaching object orientation to novices. This lack of previous work in teachers' views on object orientation makes this study exploratory in nature. If we want to discuss the teaching and learning of object orientation, we need to listen to the teachers to try to identify the basis for their approach to teaching object orientation to novices.

Discerning the views of educators teaching object orientation to novices seems critically important due to the implications for student retention, the quality of higher education as well as the quality of the professional training of teachers for upper secondary schools.

To be able to listen to educators talking about the teaching of object orientation in their own words, we decided to use a qualitative approach with semi-structured interviews.

There are no prerequisites in programming for entering a CS-majors program at university-level in Sweden. Still it is often the case that students taking the introductory programming course, have previous formal training from upper secondary school. This makes it interesting to investigate the views of both upper secondary school teachers, as well as lecturers at the university-level. Due to the swedish educational system the learning outcomes of upper secondary school courses is well known, see Appendix A for details.

The research question investigated in this paper is *How can educators' views on OO be characterised*?

*Email: marie@cs.umu.se

ISSN: 0899-3408 print/ISSN 1744-5175 online

DOI: 10.1080/0899340YYxxxxxxxx
http://www.informaworld.com

2 Related Work

It has been argued that object orientation is a natural way for problem solving. However, several studies question this claim,see a survey of studies in (Guzdial, 2008). For example, when asked to describe a given (algorithmic) situation, e.g., situations and processes that occur in a Pacman game, non-programmers did not indicate any use of categories of entities, inheritance or polymorphism. It has also been shown that novices have more problems understanding a delegated control style than a centralised one (Du Bois et al., 2006). Dale (2005) presents the result of a survey asking educators about the most difficult thing to teach in CS1. In the category object-oriented constructs, polymorphism and inheritance are major topics mentioned. Seemingly minor topics reported was: Instance methods, instance variables and static variables. Even such basic ideas as user defined classes and objects were considered difficult to teach by some of the respondents. The respondents frequently mentioned a struggle to find a balance between object orientation and more general programming constructs. Object oriented analysis and design was considered hard to exemplify because of the, by necessity, small examples.

Most of the focus on educators has been to investigate what they think of their students' difficulties and different to teach object orientation (Clancey, 2004; Holland et al., 1997; Kaczmarczyk et al., 2010; Eckerdal & Thuné, 2005). Thompson (2008) has been exploring educators' perceptions of object orientated programs, as a result formulated a number of critical aspects and then evaluated to what extent common text books address these critical aspects. Educators, at different educational level, have been asked about what topics should be taught in introductory programing courses (Schulte & Bennedsen, 2006). A thorough analysis of the educators' ranking of topic-relevance in relation to how they ranked the teaching-level of topics according to Bloom's taxonomy, supports the conclusion that focus is mainly on coding (syntactical issues). This matches research findings that students do not gain a general conceptual understanding of programming during CS1 (Guzdial, 1995; Milne & Rowe, 2002; Lahtinen et al., 2005). Teachers' experiences of their students' successes and failures, show that there is a disempowering view among some educators that their teaching has little, if any, impact on the learning outcome of the students (Pears et al., 2007). This is contradicted by Winslow (1996), who argues that educators must supply models to the students. Models of control, data structures and data representation, program design, and problem domain are all important. Models are crucial to building understanding, and if the instructor omits them, the students will make up their own models of dubious quality.

3 Method

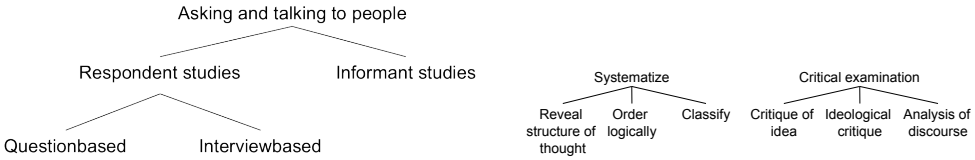
The research question was thematically operationalised according to four themes; the paradigm itself, the concept of an object, examples, and object orientation analysis and design. Each theme was viewed from three different aspects, the educator's personal view, the educator's view of student difficulties and the educator's choice of methodology to meet those issues, illustrated by the matrix in Figure 1. 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.

The goal of all qualitative research is to understand a phenomenon. According to the definition by Esaiasson et al. (2007) there are two types of studies when talking to people; respondent studies and informant studies, see Figure 2(a).

In this classification informants are witnesses and it is the different stories of a certain event that aids the researcher in putting the picture together. In these cases there is no need to ask the same questions of all the informants. In a respondent study it is the individuals themselves and their thoughts that are the object of study. In this case it is the aim of the research to find common patterns and themes in the stories told by the respondents and it is therefore important to ask or discuss the same questions.

	[C] Characteristical	[P] Problematic	[M] 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 1. Interview guide.



(a) Types of studies (b) Reseach Questions for Qualitative Studies

Figure 2. Types of studies and research questions in qualitative research.

Qualitative research can answer different kinds of questions. Figure 2(b) shows an overview of the aims of content analysis. 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.

The purpose of the study presented in this paper, was to explore and investigate ways of thinking about object orientation. Shank & Villella (2004) use the metaphor of a lantern to describe qualitative research. This kind of research sheds light on areas previously obscured.

3.1 *Sample*

In all, 10 interviews were conducted, 6 with educators from upper secondary school (students at the age 16-19) and 4 with lecturers at the university level. The group of interviewees consists of nine men and one woman, all, except one, with many years experience of teaching and experience with object orientation. The group of interviewees has been recruited through mail-contact based on either web-based searches or recommendations from university colleagues, some through a net-work for programming educators in upper secondary school. Many potential schools did not teach object orientation at all. Contact was made with responsible director of studies or similar, and if the school did teach object orientation, the request was either forwarded directly by them or a name was given to me. The final set of respondents is therefore a convenience sample. Nonetheless, the population shows a diverse background, see the demographics in Figure 3.

The columns in Figure 3 describe:

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

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 3. Demographics of Interviewees.

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 [fig:IPdata] shows the respondents' experience, in years, in teaching programming (Exp).

It was not easy to find any women teaching object orientation, so we are grateful to have one woman among the respondents.

Sample size for qualitative studies is often discussed, and Sandelowski (1995) concludes that the quality of information obtained per unit is the most critical measure. Sample size is difficult to determine and one recommendation is to proceed until analytical saturation is received. Another recommendation for this particular kind of study is to include about six to ten participants (Sandelowski, 1995), based on work by Morse (2000).

All of the upper secondary school educators (id R1-R6) are trained teachers in maths 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 (e.g. R3). This variety in teachers background in upper secondary school is probably due to the fact that Computer Science is not recognised as a subject within the teacher education system. The lack of trained CS-teachers makes it common for schools to assign science teachers, without formal CS training, to teach programming courses. Teachers in upper secondary schools are often autodidacts and on many schools they are also the only teacher teaching this subject. In is not uncommon in computer science departments for university educators to teach before and during their PhD-studies. An interesting note is that one of the university lecturers in this study earned a PhD in Chemistry before switching to CS.

3.2 *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 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 shown in this paper have been translated by the author.

In an effort to limit the effect the interviewer might have on the interviewee, the vocabulary was kept at a non-formal level, not to influence or intimidate the interviewee unnecessarily.

3.3 *Analysis*

The analysis has been done using qualitative content analysis (Hsieh & Shannon, 2005; Forman & Damschroder, 2007). Content analysis is a widely used qualitative research technique, particularly in health studies (Graneheim & Lundman, 2004; Hsieh & Shannon, 2005; Forman & Damschroder, 2007; Elo & 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 and, hence, adhere to the naturalistic paradigm. 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 & 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 of the study, is the manifest view of object orientation. This is investigated through the different aspects presented in Figure 1.

Once the transcripts were done and proof read, each was transferred from a word-document to a spreadsheet-document. All statements in the transcripts have been given an identification code [id_row]where id is the respondents identification, seen in Figure 3, and row is the row number of that particular transcript. Reading through the text, interesting statements were condensed/concentrated, in a separate column. Next, 13 columns were added, the first twelve for marking any of the 12 aspects in Figure 1, and the last column to mark other interesting comments in the text. Then the interviews were processed over again and each concentrate was labeled as belonging to one or more of the 13 aspects. 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 observe any patterns or themes among them. According to Forman & Damschroder (2007) the coding allows the data to be rearranged in analytically meaningful categories. The selected concentrates were organised into thematic categories, sometimes in several passes, to achieve a suitable level of abstraction.

During the analysis both the audio files and the transcripts have been processed many times, and, if necessary, corrections of the transcripts have been made throughout the work.

4 Results

Based on the process described in Section 3.3, three aspects from Figure 1 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*, se Figure 4.

The resulting categories are shown i Figure 5.

	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 4. Aspects analysed.

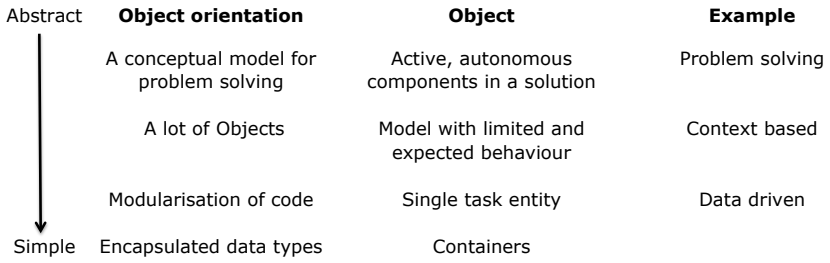


Figure 5. Categories.

4.1 *Object orientation*

Asking the educators for their personal view on object orientation was more complicated than expected. Some of the respondents tended to give examples rather than to formulate some theoretical or conceptual point of view. Often their personal view had to be collected from statements discussing more practical issues of their teaching.

Four categories emerged from the analysis. Object orientation is characterised as: *A conceptual model for problem solving*, *A lot of Objects*, *Modularisation of code*, or *Encapsulated data types*.

A conceptual model for problem solving

On the most abstract level of description R7 phrases it like this:

> **I :** Thinking more generally about OO, what is your personal view, how do you think of OO, what is kind of the central...
>
> **R7_143:** [...] I like... or what I find appealing about it is on one hand that it is a way of viewing not only computer programs but also activities and... phenomenon that you would like to describe. So you have a, foc.. some kind of spectacles or raster that you could apply when regarding something.

R7_146: Then using the same raster when making a program that in a way is related to this activity, so that this is a support to address two different types... of understanding of activities and the implementation and the design of computer programs. I find it a real strength to have a common language [for these two aspects of work, authors note]

Collaboration is rarely mentioned, but is often implicitly present. R8 is one of the few stating it explicitly:

R8_197: [...] the main idea is that I want to present classes and objects and try to make small examples where objects collaborate to solve a certain task.

However the way to do it is a problem to him/her:

R8_197: [...] Eh .. and ...I do not think that I succeed so well with this in this course, because it is a very small part of the course, eh...

A lot of Objects

The idea that object orientation is characterised by a many objects, is expressed explicitly by many of the interviewees. The concept of an object is however not clear from this expression. See Section4.2 for further discussion on this topic. R4 uses the metaphor of a smrgsbord to explain object orientation:

R4_158: I think that...,you have to understand that object orientation is lots of objects that you use, or lots of classes that you put on a smrgsbord and make a program out of it. And they work together.

The nature of the collaboration is not entirely clear, R4 continues:

R4_158: [...] you kind of pick, here is this object and I pick this method from this object and use it in my program, then I pick this object from here and use it in my program And then they work together. [...] Object orientation is that you have a method her that you call that does a lot of things for you, you don't really need to know what it really does, but all you have to know is what comes out of it.

Several of the respondents emphasises the need to be familiar with the API of Java (or similar libraries connected to other languages). R4 continues and elaborates on this:

R4_172: If you master that [using classes from a library] it does not matter if it is Java, Pearl or whatever. Eh, you, you kind of see that, here are these classes, which ones do I need to make a program that does this, well then there is this method in this class and there is this method in this class and then I pick them and put them here.

R3_75: if it feels familiar [when trying out a new language] and you know how a language works with API's and object orientation with classes and then it is not difficult to switch to another... instead of switching between lots of.. on one hand it is good to have tested a lot of languages, but then I think it becomes.... then they only get to try then they never get the possibility to get deeper into it I think, that is my personal view. It becomes to jumpy...

Modularisation of code

The use of object orientation as a way to structure code is often mentioned by the interviewees. Asked for a personal opinion on on the characteristics of object orientation R1 replies:

R1_188: Ehmmm... Oh, that was a difficult question. Yes well, or an extensive question.... Well really it is, in a way, this thing about.... to separate... no, to split the problem in... in smaller pieces ...that you should... you can view this in different ways. One way that you create your own variable types, that is... I usually have an exercise in the beginning, which I don't have now, but used to have where they should make one for complex numbers. A, A class for complex numbers. Ehm, which you then can use, that is one view of object orientation that you reuse code and that

you.... package... all that belongs to... it. And then when you have the final package then it is done, like it is.... well....

Among others, respondent R8 has to deal with the problem of a very inhomogeneous group of students. Some of the students are not taking any more programming classes, and because of them R8 builds on the procedural way of modularising a problem:

R8_188: [...] Okay, let's take this thing with functions one step further, and then we create classes, that contain function and data then.

R8 does not consider this suitable for the CS-majors, that take the same class. When further probed for the essence of object orientation R8 continues to state that he/she wants to show collaborating objects that solve a certain task (Section 4.1 quote **R8_197**). This educator would like to use different views depending on the audience, and is in fully aware that the CS-majors suffer from being forced into the same class a a group of students taking only this particular course.

Encapsulated data types

Several of the respondents have difficulties discussing object orientation from a conceptual point of view. Encapsulation is mentioned when asked about central concepts.

I: I am just going to recapitulate [...] You mentioned that what you think is central in object orientation is encapsulation. This thing about not fiddling with private data and... that, that one takes an outside view of the object.

R5_418: Yes. And that, eh.... that they [students] have great trouble in understanding the need for this.

To try to identify R2's view, he/she was asked if there was anything he/she considered contradictory to object orientation, the answer was:

R2_411: [... (thinking) ...] well, but if you look at this thing of encapsulation, which I find, that is a rather central part, how you treat data internally with an interface, when you sort of depart from that you have, you ignore that thing with encapsulation and just move on. Eh...

Probed for if this could mean, for instance, public attributes, R2 continued:

R2_417: [...] for instance, I read in one of the books that if something is set to public you don't have any problems, and that is kind of, that is true, but then, then you loose something as well... I think, because it, it is a huge point in protecting them so that they kind of... only can be modified in a, in a predefined way.

On the lower scale of abstraction, R9 is thinking about his/her way into object orientation:

R9_972: [...] I found it very difficult to understand the reason for object orientation compared to procedural programming, but I have not arrived at the answer yet (laughing)

I: So how do you describe this [difference] to the students?

R9_978: (laughing) Luckily, they do not ask me that kind of strange questions... (laughing)

R9, a bit further into the interview, argues that platform independence is the main argument to make students accept to learn Java, and that object orientation in itself could not be used to motivate students.

4.2 *Objects*

The different categories describing the view of the object as a concept are: *Active, autonomous components in a solution*, *Model with limited and expected behaviour*, *Single task entity*, and *Containers*.

Active, autonomous components in a solution

Discussing the essence of object orientation, R10 comments on the difficulty moving on from an imperative approach to an object oriented one with active, participating modules:

> **R10_95:** [...] moving on to active modules, modules that can take on the responsibility for its neighbourhood, that is such a new and different way of thinking. Ehhh... and that is where I think the students get stuck and makes... tries to make passive modules instead, and then I don't think that you have taken this step [to object orientation].

R7 expresses similar ideas, he/she wants the students to see programming as a way of solving problems and states his/her view of objects:

> **R7_338:** well you know I... and this relates back to my aversion towards the usual examples that you find in Java-books or programming books and on the web and, that... I have a difficulty seeing.... I find Customer as a realistic example of something that many of them could be working on in a future profession.

R7 then reflects on the success of their efforts:

> **R7_380:** [...] I do think that we have managed to convey this way of viewing the world as consisting of objects that originates from some kind of abstract model or how to put it and yes...

But further into the interview, R7 has to admit that the students have a difficulty moving from the conceptual view to actually gaining skills in implementing their designs.

Model with limited and expected behaviour

In a slightly more limited view, some of the respondents discuss natural models. Object should be properly instantiated and behave naturally. R1 is looking at an example with a Die, and is surprised that there is no constructor that takes the number of faces as parameter:

> **R1_779:** [...] I would choose to have one more constructor

This particular Die-class has set- and get- methods for accessing the faceValue of the die. R1 finds this strange:

> **R1_779:** You can not set.... No, it ahh. That depends on what you want it for, but I don't see how you.... why you should set the value of a die-roll.
>
> **R1_791:** [...] well, you think about a die really, and the only thing you can do with it is to roll it and to read the face value. An that is kind of....

R4 is also discussing the limited behaviour of ideal objects:

> **R4_320:** an ideal object... an ideal object, if you consider having a very particular, if I may return to the gambling-context, an ideal object is an object that does a very specific thing that makes very specific, that can perform specific tasks. I think that you must not do too complex objects, how shall I put it? [...]

R4 then moves on to comment on the fact that the same entity can be a good candidate for a class in one context but too complex in another. This long elaboration ends with a summary of ideal objects:

> **R4_320:** [...] So to me, the objects must not be ehh, may not be too complex because then I would kind of split them into smaller pieces, because if they get too complex then it will be too many methods. It is like 586 different methods and 373 different attributes and that, no one will ever have the energy to learn that.

R9 indicates the possibility to take an outside view of objects, to think of objects the way a potential client would:

R9_423: Methods, often I compare this to consulting. I usually explain it that way. The, it's kind of a... you order a service from a consultant. Ok. And then I move on to what the consultant needs, the parameters. What the consultant needs to do my work.

This is the only occasion in all the data that anyone expresses the use of an outside view of objects.

Single task entity

Conceptually slightly less complex is the idea of small units. R3 was asked what kind of objects he/she would never show to students, and expressed his/her view like this:

R3_231: [thinking for some time] well, objects.... the classes should be properly built. The classes must be properly built, a class should do one thing, You can not mix a lot of things, I warn them about that. A class should be clean, it should perform one thing, the methods in that class should only do one thing. I am very particular about that, [...] I tell the students that, on class should be one thing, you build objects from it. Method should be... do one thing, you should not mix things in this. In that case you make a new class, in hat case with that object. Try to separate the code so that...

R4 also expresses the reasons for and consequences of simplicity:

R4_329: [...] and that is why we in Java and all these I mean, is it anyone who cares to count all the classes and objects that there is in Java, it is thousands of millions of objects in Java, and I think that this is due to the fact that you don't want objects to be too complex, but that objects in themselves should be fairly simple and that there might be other objects that inherits from this object, and altogether it becomes a huge amount of methods maybe if you think about all the methods they inherit and themselves.

Containers

Several of the respondents expresses the view of objects being mere containers.

R3_303: Well, because it is data, we look more at the data kind of that.., what will you collect data for? You look at it [the data], well then this should be an object of this data and then... well you can't mix this into this object but now this becomes another object. Now we have to build this to make it. So finally when you collect the data together you can see what you should have, the demands around that I think.

R5 has made a choice to base all his/her application around databases. Everything has to stored in some way, and the natural solution to this is a database.

R5_25: Well, we... we do have something that I think is really good when you're dealing with object orientation, that is that you have a database beneath, in some way... because they... the model-view-control thinking. Model is almost always... ehhh, a table in the database. That is not particularly.. it's almost always that way. Then we get into, what is it- then we get into something like instead of talking about objects we talk about, what is it that we need to store, what is there, are they related etc. And that is really the same as object orientation .

R5 continues to comment on the fact that this enables him/her to discuss the importance of storing information only in one spot and relationships among data.

R5_25: I feel much more comfortable with... with discussing how to normalise a database and how to design a database than discussing how you... which.. cl... objects we should have. And they are rather close, I think... in the end once you are starting to implement it.

R5 also finds graphics useful and driving object orientation:

R5_25: Those that... [for their project] make a game, that becomes o- that becomes rather object oriented. Because they have so- some character walking.. meeting a lot of zombies and then you

make a zombie-class and then it is generated.... a lot of zombies that you... fire at and then every bullet is... an object too and so on, so it becomes very... but they are... that it became so object oriented depends on the fact that they write their code in Flash which is- , and there's nothing else there but objects that in addition are graphical .. eh.. objects.

No other respondent takes such an explicit stand on how to define, and how to work with objects.

4.3 *Examples*

Discussing object orientation and how to exemplify it is easier than discussing the more generic characteristics of object oriented examples. Listening to the respondents, there are however bits and pieces here and there that gives a description of their view of examples. The emerging categories are: *Problem solving*, *Context based*, and *Data driven*.

Problem solving

R7 has previously stated his/her view of object orientation as a problem solving tool, and this is also reflected in his view of examples:

> **R7_344:** [moving on from the example of `Customer` **R1_338**] but, `Car` and `Bike` and things like that, ok it might work from a pure programming point of view, but I think that our students, if you generalize, I am not going to do that, but many of our students don't view programming as... as this, this pleasurable, self-sufficient activity. But they see it as a means for something else. And if I don't offer examples that... makes it credible that programming is a tool that they might actually need to do this, then I won't get through to them. They just don't see the use of being able to represent Cars in a computer program.

R2 is also expressing a view of of object orientation being so much more than the technical part of programming, and that this has to show in examples:

> **R2_246:** Ehh, I try to find examples that are as pedagogical and close to reality as possible, I think that, a shortage in some of the course literature is that it gets too technical, that is, they just show the technical details of programming and not kind of, connects to the problem, because I think that object orientation is much more about problem solving than programming. In... that job is, later when you're done, then it becomes pure programming and that is something... something slightly different, that is another skill.

Most of the respondent are at several occasions commenting on the lack, or shortage of, really good examples. Despite this complaint it was almost impossible to have them characterize a good example.

Context based

The need for examples to be contextually situated is mentioned by many of the respondents explicitly:

> **R7_287:** so what I'm looking for and try to design myself, that's classes that, if I try to make a connection to the course they have taken before programming, when they have been looking at activities, then I try to use classes of the kind `Customer`, ... and ... well, things that are relevant or how to put it. Something that you, that seem reasonable to.. [disrupted by a person entering the office]

R9 is very concerned of making the example connect to something in the every-day life of his/her students:

> **R9_204:** It is ... now I only have students from X [exchange students] , then it is kind of easier to explain to them. Ok, you have arrived here to study and your parents want to transfer money

to you. Do you want them to transfer the money to my account or yours? Then they want them to transfer the money to them because... Then you have to go to the bank an get an account. And then I explain to them that when you go to the bank if you... well, creates an account what does that mean? It means you have to have a social security number. You want to have an address and name and so forth. Then you create an object, each person that you create an account for becomes an object of the class `BankAccount`. Or something like that...

The context of bank and bank accounts is a common theme in textbooks, and is mentioned by R3 as well

R3_171: Where you deposit and withdraw money. Because I got that from.. I think I got that from U [mentions the university where he/she received his/her degree]. That it is... `BankAccout` is the name of the class, and there you have only name and you have a balance. Two variables. String name, and int balance. And you have `getBalance` and `setBalance` and a constructor where you can supply values, the name and the lance that you have to begin with. So I think it is... they understand this sufficiently, a bank, because they understand this, deposit money, withdraw money from an account it's only two variables which isn't much, that is what I start out with.

In upper secondary school, all the teachers have to take the students from procedural to object oriented programming. R4 tries to exemplify this difference, and describes how he/she chooses his examples:

R4_182-266: Eh, that, that I take, I take an example close to them [students], [...] Ehm, and that in object orientation you explain, with the help of objects, a world. That, for instance, that we would like to describe a forest in procedural programming them you see things that might happen in the forest, functions, the sounds of birds plants leafs falling etc. In object oriented programming you see the objects that the forest is made up of, trees, rocks, birds, and so on, that, I try to, something like that, that was the forest before and this is the forest now.

Data driven

For many of the respondents, typical examples are focused on data to be stored. R5 can not think of any other origin for his/her presentation of a problem:

R5_25: Then we get into, what is it- then we get into something like instead of talking about objects we talk about, what is it that we need to store, what is there, are they related etc.

Some of the respondents use the term data type in connection to classes, generally meaning something like a record in Pascal:

R2_309: Well then, then it's more of a, a, well you, you have a need for a certain data type, a composed data type, eh, it might be a ... what later on would be a post in a database or something like that, or ... eh.... a country with, with certain data that belongs or, a document med certain characteristics. Eh.... where you collect, eh attributes and methods in a class and by this showing that it kind of belongs together.

5 Discussion and Conclusions

The purpose of this study has been to look for different ways of viewing object orientation, not to categorise individual educators. Several of the respondents discuss object orientation using several views, and some only express a single view.

An interesting observation is that, even though 8 of the 10 educators have received formal university training in object orientation, the level of abstraction in their views of object orientation is low.

Although many of the upper secondary school educators have formal training in terms of university courses, this is usually not included in their teachers training program. Only one of them has moved on from CS-major degree to complement with a teaching degree. But as

seen in the quotations, a university degree does not in itself guarantee that the view of object orientation is in line with generally accepted concept definitions. One interesting observation is the choices of application areas. Only two of the respondents explicitly want their examples and contexts to be realistic, in terms of software. Other respondents, who take on a more data driven design approach, settle for applications where it is obvious that a large amount of data should be handled. The storage of data then becomes the driving force for object definition.

Students' conception of object orientation after attending a CS1 course has been investigated through concept maps, by Sanders et al. (2008). CS1 is by necessity focusing on both introduction to programming as well as introduction to the object oriented paradigm, which shows in the students' concept maps. Few maps indicate any relationships other than among the syntactical components of programming. This corroborates the resulting categories of our study. For the majority of the respondents, the views of object orientation were displaying a focus on programming/syntax skills, rather than emphasising a conceptual view.

In the present study there is no single mentioning of abstractions, and behaviour is only discussed in terms of methods, or even functions. A majority of the educators view classes as primarily data types, consisting of primitive values to be manipulated. This does not promote an outside, problem-oriented perspective with collaborating objects as autonomous service providers for clients. Only one of the respondents indicates this view, using the metaphor of consultants to describe methods to students.

The way educators view object orientation and objects will have a large impact on students being introduced to the paradigm. Who teaches the teacher? In Sweden currently, there is no prerequisite for a teacher in upper secondary school to have a formal training in computer science. Even university lecturers give little evidence of a paradigmatic discussion.

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 views in the three aspects investigated.

There is no doubt, that in teaching we always have to compromise with principles for practical reasons. Audience, time, and space will restrict the possibilities to show object orientation at its best. There is a lot of work to be done in terms of practical suggestions for line of presentation and good examples to support the struggling educators. Nevertheless, it must be the ambition of educators to convey the essence of the object oriented paradigm even in details.

Threats to validity

There is always a possibility that the mere fact that a certain research area is discussed influences the statements of the respondents. Some might have a nagging feeling of being evaluated and might be tactical in their description of certain subjects. We seldom reveal what we consider to be unfavourable about ourselves.

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 examples for novices has been the main focus of my research for the last three years. The quality of examples in text books must be considered low, as evaluated in previous work (Börstler et al., 2008, 2009, 2010), and this raises questions regarding the view of object orientation among educators in general. Being aware of this, I have made an effort to set aside my preconceptions of object oriented quality.

6 Conclusions

Acknowledgement

The author is greatly indebted to the participating educators. Without their willingness to share their thoughts and experiences, and to devote time and effort to this project, this research would not have been possible.

References

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

Börstler, J., & Hadar, I. (2008). Eleventh Workshop on Pedagogies and Tools for the Teaching and Learning of Object Oriented Concepts. In ECOOP 2007 Workshop Reader, Vol. LNCS 4906 of *Lecture Notes in Computer Science* (pp. 182–192). Springer.

Börstler, J., Hall, M.S., Nordström, M., Paterson, J.H., Sanders, K., Schulte, C., et al. (2009). An Evaluation of Object Oriented Example Programs in Introductory Programming Textbooks. *Inroads*, *41*, 126–143.

Börstler, J., Nordström, M., & Paterson, J.H. (2010). On the Quality of Examples in Introductory Java Textbooks. *The ACM Transactions on Computing Education (TOCE)*, *Accepted for publication*.

Clancey, M. (2004). In S. Fincher & M. Petre (Eds.), *Misconceptions and Attitudes that Infere with Learning to Program*. (pp. 85–100). Taylor & Francis.

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

Du Bois, B., Demeyer, S., Verelst, J., & Temmerman, T.M.M. (2006). Does God Class Decomposition Affect Comprehensibility?. In P. Kokol (Ed.), SE 2006 International Multi-Conference on Software Engineering (pp. 346–355).

Eckerdal, A., & Thuné, M. (2005). Novice Java programmers' conceptions of "object" and "class", and variation theory. In ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education, Caparica, Portugal, July (pp. 89–93). New York, NY, USA: ACM.

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

Esaiasson, P., Gilljam, M., Oscarsson, H., & Wängnerud, L. (2007). *Metodpraktikan–Konsten att studera samhälle, individ och marknad (In Swedish)*. Norstedts Juridik.

Forman, J., & Damschroder, L. (2007). Qualitative Content Analysis. *Advances in Bioethics*, *11*, 39–62.

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

Guzdial, M. (1995). Centralized Mindset: A Student Problem with Object-Oriented Programming. In Proceedings of the 26th Technical Symposium on Computer Science Education (pp. 182–185).

Guzdial, M. (2008). Paving the way for computational thinking. *Commun. ACM*, *51*(8), 25–27.

Holland, S., Griffiths, R., & Woodman, M. (1997). Avoiding Object Misconceptions. In Proceedings of the 28th Technical Symposium on Computer Science Education (pp. 131–134).

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

Kaczmarczyk, L.C., Petrick, E.R., East, J.P., & Herman, G.L. (2010). Identifying student misconceptions of programming. In SIGCSE '10: Proceedings of the 41st ACM technical symposium

on Computer science education, Milwaukee, Wisconsin, USA (pp. 107–111). New York, NY, USA: ACM.

Lahtinen, E., Ala-Mutka, K., & 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 (pp. 14–18).

Milne, I., & Rowe, G. (2002). Difficulties in Learning and Teaching Programming—Views of Students and Tutors. *Education and Information Technologies*, *7*(1), 55–66.

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.

Pears, A., Berglund, A., Eckerdal, A., East, P., Kinnunen, P., Malmi, L., et al. (2007). What's the problem? Teachers' experience of student learning successes and failures. In R. Lister & Simon (Eds.), Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007), Vol. 88 of *CRPIT* (pp. 207–211). Koli National Park, Finland: ACS.

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

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

Shank, G., & Villella, O. (2004). Building on New Foundations: Core Principles and New Directions for Qualitative Research. *The Journal of Educational Research*, *98*(1), 46–55.

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.

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

Transcriva (????). , Transcriva Homepage. `http://www.bartastechnologies.com/products/transcriva/`.

Winslow, L.E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17–22.

Appendix A: Programing Education 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. Because of this it is well known what the syllabi and requirements for programming courses in upper secondary school are ((Skolverket, 2010b). The Swedish upper secondary school is entered at the age of 16, and consists of three year programs. 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. All courses concerning computers and programming are organized in a subject called Computer technology. In Figure A1 the structure and relationships among the programming courses in upper secondary school is shown.

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 inter-

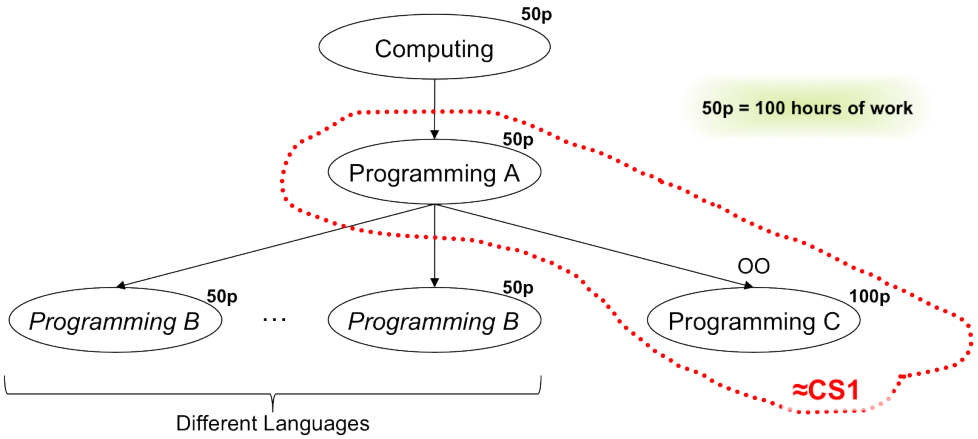


Figure A1. Computing courses in upper secondary school.

faces. 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.

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. 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.

Paper VI

Educators' strategies for OOA&D

Educators' Strategies for Object Oriented Analysis and Design

Marie Nordström
marie@cs.umu.se
Department of Computing Science
Umeå University, Sweden

Abstract

Almost all research on the teaching of object orientation has been focused on the students and their learning. One important aspects that will affect how object orientation is taught, is the educators personal views on different issues of the paradigm. In this paper we present some results of a qualitative study on educators views on the teaching of object orientation. We specifically focus on how teachers address object oriented design and analysis. Data was collected through interviews with ten educators.

1 Introduction

An important aspect of teaching object orientation to novices is to introduce the students to a problem solving approach. Without knowing how to approach a problem and how to look for suitable objects in the problem domain, it is difficult to acquire skills in object oriented problem solving and programming.

We know very little about the way educators themselves think about object oriented analysis and design. We do not know of the methods used by experienced educators to enhance the introduction to object oriented problem solving and programming. There are, to our knowledge, no studies on these perspectives of teaching object orientation to novices. The lack of previous work in teachers views on object orientation has made this study exploratory in nature.

Discerning different views of educators teaching object orientation to novices, seems critically important due to the implications for student retention, the quality of higher education as well as the quality of the professional training of teachers for upper secondary schools.

The method used to investigate the area is a qualitative approach using qualitative content analysis of semi-structured interviews.

The research question investigated in this paper is *How can educators' views on OOA&D be characterised*?

2 Related Work

It has been argued that object orientation is a "natural" way for problem solving. However, several studies question this claim (Guzdial, 2008), when asked to describe a given (algorithmic) situation, e.g., situations and processes that occur in a Pacman game, non-programmers did not indicate any use of categories of entities, inheritance or polymorphism. It has also been shown that novices have more problems understanding a delegated control style than a centralised one (Du Bois et al., 2006).

Dale (2005) presents the result of a survey asking educators about the most difficult thing to teach in CS1. In the category object-oriented constructs, polymorphism and inheritance were major topics mentioned. Seemingly minor topics reported were: instance methods, instance variables and static variables. Even such basic ideas as user defined classes and objects were considered difficult to teach by some of the respondents. The struggle to find a balance between object orientation and more general programming constructs was also frequently mentioned. Object oriented analysis and design was considered hard to exemplify because of the, by necessity, small examples.

Most of the research involving teachers has been to investigate what they think of their students' difficulties, and on different approaches to teach object orientation (Clancey, 2004; Holland et al., 1997; Kaczmarczyk et al., 2010; Eckerdal et al., 2005). Thompson (2008) has been exploring practitioners perceptions of design characteristics in object oriented programs. The results show a span of five perspectives, from the lowest with a focus on language, to the highest category where the cognitive process is the primary focus. 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.

3 Programming Education

To investigate educators' views on object oriented analysis and design we have chosen to focus on both lecturers at the university level and upper secondary school teachers.

In Sweden, in general, the computing curricula of university programmes with CS majors or minors, contain traditional CS1 and CS2 courses.

The Swedish upper secondary school is entered at the age of 16, and is regulated by the Swedish National Agency for Education (Skolverket (2010a)). Because of this, the syllabi of programming courses in upper secondary school is known, which makes it meaningful to include teachers at this level.

All courses concerning computers and programming are organised in a subject called Computer technology. In Figure 1 the structure and relationships among the programming courses in upper secondary school is shown.

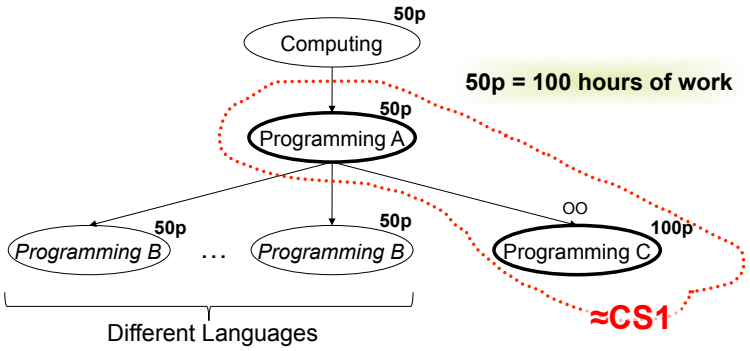


Fig. 1: The structure of programming courses in Swedish Upper Secondary School

Computing 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* provides 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, see Skolverket (2010b) for more details.

4 Method

The research question was thematically operationalised according to four themes; the paradigm itself, the concept of an object, examples and the problem solving process of object orientation. Each theme was viewed from three different aspects, the educator's personal view, the educator's view of student difficulties and the educator's choice of methodology to address those difficulties, see Figure 2 and (Nordström, 2010) for further details.

The goal of all qualitative research is to understand a phenomenon. Shank and Villella (2004) use the metaphor of a lantern to describe qualitative research. In this particular study the purpose was to explore and investigate different ways of thinking about object orientation among educators.

The data for this study has been collected through semi-structured interviews and analysed through qualitative content analysis.

4.1 Sample

In all, 10 interviews were conducted, 6 with teachers from upper secondary school (students at the age 16-19) and 4 with lecturers at the university level. The group of interviewees consists of nine men and one woman, all, except one, with many years experience of teaching and experience with object orientation.

	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?

Fig. 2: Interview guide.

The educators participating in this study show diverse backgrounds. The demographic data shown in Figure 3 consists of what type of degree the respondent holds, if object orientation has been acquired through formal training or through own studies. The figure also shows whether the interviewee teacher at upper secondary school (USS) or university. The relative size of the schools as given as well as the number of years of experience in teaching programming, regardless of paradigm.

The demographics of the respondents are shown in Figure 3, and the content described below.

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

Fig. 3: Demographics of Interviewees. T*: R2 is on the way to a teaching degree at the time of the interview.

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 3 shows the respondents' experience, in years, in teaching programming (*Exp*).

It was not easy to find women teaching object orientation, so we are grateful to have one woman among the respondents. Sample size for qualitative studies is often discussed, and Sandelowski (1995) concludes that the quality of information obtained per unit is the most critical measure. Sample size is difficult to determine and one recommendation is to proceed until analytical saturation is received. Another recommendation for this particular kind of study is to include about six to ten participants (Sandelowski, 1995; Morse, 1991).

All of the upper secondary school teachers, with the exception of R3, are trained teachers in maths and/or physics. Another background not uncommon in Sweden, is to have a bachelors degree in some major subject and then to add courses for the fulfillment of a teachers degree (e.g. R3). This variety in teacher background in upper secondary schools is due to the fact that Computer Science is not recognised as a subject within the teacher training programmes in Sweden. The lack of trained CS-teachers makes it common for schools to assign science teachers, even without formal CS training, to teach programming courses. They are often autodidacts and on many schools the sole teacher in this subject. CS being a young discipline it is not unusual for university educators to teach before and during their PhD-studies. One of the university lecturers in this study earned a PhD in Chemistry before switching to CS.

4.2 Interviews

The interviews lasted in the range of 45 minutes to 1 hour and 16 minutes. The interviews were all conducted by the author, in Swedish. A verbatim transcription was conducted by, or supervised by, the author, using Transcriva. All interview quotes throughout the present paper have been translated by the author.

In an attempt to limit the effect the interviewer might have on the interviewee, the vocabulary was kept at a non-formal level, to avoid any unnecessary influence or intimidation of the interviewee. The interviewer is always part of the research, and the relationship between the interviewer and interviewee affects the outcome of the interview.

In all cases, the interview was performed at a locality of the interviewees choice, on most occasions in their office. Every interview starts with the interviewer asking the interviewee to describe his/her background, how he/she came to this point in his/her professional life, teaching object orientation to novices.

4.3 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; 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. The primary objective is the manifest view of object orientation investigated through a number of different aspects. A thorough description of the study as well as results concerning educators personal view on object orientation, objects and examples for object orientation can be found in (Nordström, 2010).

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 3), and *row* is the row number of that particular transcript.

5 Results

In the present paper, two aspects are analysed: *Educators choice of methodology for introducing OO* and *Educators choice of methodology for teaching OOA&D*.

5.1 Introducing the Paradigm

To investigate how the paradigm was introduced, information had to be collected throughout the interviews. A more direct question would often be perceived as asking about objects or programming in general.

Three categories of strategies for the introduction of object orientation as a problem solving approach have emerged from the interviews. They can be characterised as: *Building a world of objects*, *Induced by contexts, databases and concepts*, or *Not addressed*.

Building a world of objects

In upper secondary school the students have already been introduced to programming through the procedural/imperative paradigm. Some of the teachers utilize this prerequisite knowledge to show the difference in approach using the object oriented paradigm.

R4_182: *eh, i use, i use an example close to them, that i, that we do it differently, [...] so far we have been working with functional programming, eh and..., and try to show what the difference is. eh.. and that in object orientation you explain this with the aid of objects, a world. that, for example, that we are to describe a forest in procedural programming then you see things that might happen in the forest, functions, birds sing, trees grow, leafs are falling and so on. in object oriented programming you see the objects that make up the forest, trees, stones, birds, and so on... that i try, something like that, that was the forest before and this is the forest now.*

In many university programmes the introduction to programming is done directly in the object oriented paradigm. In general, R10 and R7 apply the most theoretical approach.

R10_164: *Well... I was going to say that the first thing I do, usually is.., eh... an example that I use.. an enclosed field where it, eh.. where there are different types of beings. There are carrots that grow, and bunnies that jump around and a wolf maybe that chases the rabbits and things like that. So then, then we have different types of objects interacting in this field.*

R7_527: *Well... I am not sure actually, I told before that I try to work scenario ... based, or how I should phrase it, I try to put things into a potential...*

There are however not many instances in these interviews, where the educators explicitly attempt to illustrate object orientation. Much more common is the use of contexts that force some kind of object oriented solution.

Induced by contexts, databases and concepts

R8 avoided discussing how he/she introduces object orientation, using different excuses, this and that being exceptions for a particular course he/she is teaching, so finally the interviewer poses a direct question:

I_194: *but since it has been decided to introduce object orientation, you must have an idea of what that means.*

R8_197: *Yes right, but then it is, the main, the basic idea so to speak. That is, that I want to introduce classes and objects, and try to make small examples where object interact to solve a certain problem, Eh.. and... I don't think I succeed because this is a very small part of the course, eh...*

Using contexts like games seem to be a popular way to "show" the students what object orientation is.

R8_209: *In a way, you might say that object orientation presents itself. The students do a project, because, they make a project, many... and get to chose [project] for themselves, more or less. But since I hint about what to do it ends with them building small games. And that is smart to, then they use an existing module, that enhances this game construction. Eh, and then inheritance show up naturally, they have to inherit from some module classes, so that they have existing Sprite-classes that they inherit to their own Sprite and particularly, well and things like that, so it becomes kind of a drill in , in certain object oriented concepts in that project.*

One of the educators is consistently using databases as a point of departure, even when implicitly introducing object orientation.

R5_9: *[This particular programme] demands a different kind of motivation than the Science programme, you can not introduce the concepts theoretically to them, they need to see the practical uses. This is done through databases.*

Not addressed

Most interviewees (eight) do not specifically introduce object orientation as a paradigm.

I_299: *So, you are not making any introduction to object orientation in general, as a problem solving approach? The difference compared to the imperative approach?*

R1_302: *Nae.... I ... no not in that respect.*

Prompted for any particular characteristics that would definitely be considered non-object oriented, R2 answered:

R2_435: *Well, it... But I, yes, no I have not... not decided on any high standards when it comes to that [object orientation], no I haven't.*

R6 is really enthusiastic about objects and object orientation, but still has a hard time finding a way to formulate the essence of object orientation to his/her students:

R6_455: *you're standing there trying to convince them that objects are IT.*

I_458: *And how do you get this idea through to them?*

R6_461: *I think I do that through.. partly through... eh, it is something that has emerged lately, it has... it has not been around for long. [...] you see it almost everywhere [...] it is a reality [...] hopeless to avoid*

One of the educators even admits to having trouble understanding him-/herself:

R9_972: *I don't remember... but... well, yes of course I have... they know it is about Java, so I try to give some background on Java and why it is object orientation and what the goal of object orientations is. Although I had a hard time accepting Java when it it first showed up, at the end of... or when I started to learn some object orientation, I had a hard time understanding the motives why object orientation compared to procedural [programming] that we had then, but I really haven't gotten the answer yet (laughing).*

5.2 Teaching 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 is seen in Figure 4.

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

Fig. 4: A&D method

Lexical analysis

R8 is the educator most explicitly discussing active objects as an important part of an object oriented solution.

R10_734: *Well I think I am a little towards, maybe that you start from some kind of... eh, well,... how should I phrase it, this thing with nouns, just to have a starting point. And the I am rather fond of analysing the interaction, to work with that.*

Design Patterns

Several of the interviewees mention design patterns. R5 is explicitly working with the Model-View-Controller pattern. To make the students realise the need for a structured approach he/she makes them "box themselves into a corner" before showing them this pattern. Even though not in the explicit way of MVC-patterns, R2 promotes a separation of the core of the programme from the user interface:

R2_270: *Yes we do that... eh, I have designed a couple of assignments, or I have had a couple of assignments that they start out to solve in a pure... in an environment with only text representation and then they have to add GUI's to that. I do think... that it is a part of the problem that you can separate, that the graphical interface is something that is beside or surrounding the problem, and can be solved separately.*

One of the educators uses design patterns as tools for object oriented problem solving:

R10_647: *[...] we talk about some, some chosen design patterns, I have chosen a couple that I find, eh,... that, that in some sense deals with the more general ideas of object orientation. Eh, like for instance specialisation.*

Design reasoning

The most commonly mentioned way of handling analysis and design is talking while doing. The educator is using his/her own work during teaching as illustration. This is not done in a formal way, but conveyed through discussion of different approaches and decisions while developing a solution to a problem.

R8_563: *Well not that much, well. I don't think, well I don't think that class diagram is that interesting when there is only one class. But it, eh, then you can think and write a little about what, what things we need to know about this Elevator, and then, eh, then I connect the computer and try to run, to write an example corresponding to what's on the blackboard.*

Collecting ideas from the students, makes them active and part of the implicit design process:

R4_266: *[...] In fact, I ask the students what we should do, for instance in a school context. Well, well, Student then, and what.. and then it pops up what kind of attribute a Student should have. And what a Student can do and so on, and what happens if you study? Well, IQ might increase and so on. And so on, and well... as naive as possible [...] they have to participate and it should be as simple as possible.*

Gently guiding the students through problem solving, R10 expresses a strategy of "thinking with the students".

R10_512: *Eh, in this course I also work a lot with me just standing there with the exercises, and I don't bring prepared solutions, because I want to walk the students through the process of thought. Or that they should contribute with input, and from that input we do the thinking process.*

R10 also tries to show several solutions and discuss differences, and consequences of different approaches. This is also to encourage students to try out different ideas, to be a little bit more creative without knowing where it will end. R10 wants to encourage them not to be afraid to try their ideas.

R10_980: *[...] Since I do it on the board [solves problems] and collect ideas from the students, and kind of, someone says something, then "well, then you thought in that direction instead, well then this is the result" and then you hear "well you thought about it this way", and then you really confirm and legitimise different ways of doing it. And encourages the students to do not one, but three solutions.*

R10_986: *and then, then when you have made three, then you start thinking if anyone of them is better than the others.*

Less articulate, in terms of strategy, but still with a conscious goal to foster design:

R2_162: *Well, well I try to help them as best as I can, how to think and to isolate... eh, attributes and phenomena.*

Few of the interviewees uses an explicit method for the introduction of object oriented analysis and design. However, looking at what and how they are describing their work, some implicit strategies for analysis and design show up.

Scenario based

Trying to make sure that problems and exercises are non-artificial, R7 seeks problem domains that are realistic from a software developing point of view, and tries to formulate events in this context.

R7_527: *[...] I try to work from scenarios, or what to call it, try to put things into a potential context*

Even more clearly, R5 expresses that the design is defined from possible events in the system.

R5_37: *Well, this is done through cases, you write down all cases there are and plan for what you need and fetch and so on.*

Metaphors

Some of the educators deliberately uses the metaphors of clients and service providers, to convey an outside view of objects. What can be expected from the clients point of view and what could be supplied from the providers point of view.

R9_423: *Methods, I often compare them to a service supplied by a consultant. I try to explain it that way. That you.... you order a service from a consultant. OK. And then I move on to this, what the consultant need, the parameters. What the consultant need to accommodate my request. And that is how I...*

In support of object thinking, some educators explicitly discusses the use of `main`, or imagined clients.

R10_872: *In my examples, actually we, eh... I have made a separate class, some kind of client-class kind of.*

R10_884: *It is much about acquiring the feeling that programmes are not necessarily something that is directly used by the one sitting at the terminal, but that it might as well be used by another programme, that the the user must not be a human. [...] to return something does not mean to display it on the screen, but it is something sent to someone who have asked for this value, kind of.*

R2_276: *[...] I have talked a lot about the communication between different parts, eh and that is... discussing object orientation based on well defined interfaces among objects. And then, it helps those pieces as well.*

If one way to support the perception of interacting objects is to take the clients view, then another, maybe complementing, approach might be to *play* the role of an object:

R6_392: *When you get into what an object is and ehh.... how to think about an object.. [...] when you call a method, but you may also think about it in terms of talking to the objects, that you pose a question, that it becomes more of a conversation among objects [...] I play a game pretending to be the object and eh-.. maybe pretend that a student is an object...*

Promoting the idea of active objects, as opposed to passive data containers, R10 humanises the objects:

R10_173: *Let's say that someone would like to move this chair, then this person will have to communicate with this chair-object saying "now I push you" and then the chair will react to this in some way. And then it might check "is someone sitting on me?" because then I'm supposed to react in one way, and if not my reaction will be something else, and so on. But again, then the chair becomes active all of a sudden.*

Formal notation – UML

When it comes to notational support for object oriented analysis and design, surprisingly few interviewees brings this up, and when asked directly, they vaguely refer to lack of time and focusing on the "core". Some use UML and others some semi-formal notation of their own.

R7_65: *Yes, we do [use a formal notation] and they learn at least class diagrams in UML and some relations; Association, aggregation and inheritance.*

However, students are seldom required to achieve any notational skills.

R10_275: *But, I have set the requirements to be that if they see a UML-diagram they should be able to recognise and understand it briefly.*

R2_369: *I lecture UML.*

Neither R10 nor R2 require their students to use it in their own development or documentation.

R6_215: *Well, [exemplifying with* `String`*-objects] I drew a circle with data in the middle and functions,* `at`*,* `length` *surrounding on the outside. And then I tell them that what is inside is hidden. And you have to communicate with this text over those functions, something like that.*

Object-rich contexts

One way to compensate for the lack of a methodological approach to object oriented analysis and design is to make sure that the problem domain in itself naturally contains many objects. This way the possibilities of making reasonably good choices when modeling the problem are much higher than when dealing with more general problem definitions. Several of the educators have made a choice to use object-rich contexts. Robots, games and web applications can all be viewed as populated with easily recognised objects. Sometimes this is in combination with databases and graphical interfaces.

R4_236: *Well, when you're dealing with games it becomes much easier to put all those classes into, eh, a game. Now we have this game, these are 3D-models or something, and they, they accept that they understand that this is 3D-models, this may be the camera and so on, and it is much easier for me to argue that if we have this game, we need to have a class that pulls these objects into our game. We need something to control the camera, when you move the mouse or keyboard you have to be able to move the camera and view from different angles, and that, they accept that. And somehow I think it's a little bit fun too and then they almost get along with, well let's do a separate class for this. And then, precisely in this situation it becomes easier for them to accept that you design classes of your own.*

R4 also uses the forest to illustrate the object view of a problem domain, as shown in the quote R4_182 in section 5.1.

R8 teaches a slightly differently organised course, with a shorter introduction to object oriented programming followed by a larger project.

R8_374: *Well they have to, the first day they have to come up with an idea for a project, and then they have a couple of days to move on with the design, or mainly functionality and by that time we have not yet introduced use-cases and things like that, it is more to write...*

Most of the students pick games for their projects:

R8_209: *[...] they can chose almost anything for their project. But since I hint them possible things to do, it usually ends up with them building small games. And the smart thing is that they get to use an existing module [library] that enhances the game building. Eh, and then inheritance is automatically included, they have to inherit from module classes [...] so it becomes an exercise in object oriented concepts in this project.*

R8_377: *But, ... in this you might say that it is quite some thinking to do about what could be needed, and then you could say that, when building small games, well then it pretty much shows itself, that they need a hero and some enemies and a field/track and so on. Eh, and then the rest so to speak, the really difficult parts are handled by this framework with events and things like that, so they don't have to think about that much themselves.*

Other, slightly more restricted, object-rich contexts are hotels, banking and student administrative systems where there are a limited number of entities to deal with:

R9_228: *And then I use for example the results to be formally registered by the student administrator, that could be a class too. Then I move on, from the basics...*

R7_416: *Then I might use an example of a... I have used a hotel scenario where you have a class Booking, and this booking has, besides being placed by someone, it is a booking of something and this something might be a room, and a room then becomes a class of its own, ...*

5.3 Not supporting Object Oriented Analysis and Design

Not supporting object oriented analysis and design does not mean avoiding objects. There will always be objects, and that could in one sense be regarded as supporting analysis and design implicitly through examples. However, in some cases the students got to decide on very few classes of their own. Either the design was given by the lecturer, or through libraries, and in some cases through formal notation or in words. The practices not giving any support for object oriented analysis and design emerging from the interviews can be summarized as: *Data driven*, *Objects supplied*, *Physical objects* and *Design supplied*.

Focusing on databases makes the design entirely driven by the data to be stored and does not leave much room for object oriented design decisions. Using graphical contexts such as games, makes an extensive use of library classes necessary, and the design of objects is to some extent already decided through the library classes available.

Some of the educators use physical objects like chairs and cars, to convey the idea of objects. Used without a motivating context this often results in a static behaviour. It is also common to practice programing skills through implementation of given designs. These designs are sometimes given in UML-diagrams or as a detailed descriptions of named attributes and methods to be implemented.

6 Conclusions

In the educational context we have conditions limiting the possible approaches to teach the subject at hand. What and how an educator does in his/her teaching is depending on several factors. Student group, time frames, curricula to be covered, but also a personal belief of what is essential and what is difficult in the subject, will affect the teaching. The purpose of this study has been to listen to educators describing how they introduce object orientation, and how they introduce object oriented analysis and design.

Most of the categories for analysis and design emerging from the ten interviews are used by single educators, and often only mentioned or superficially demonstrated to the students. Only two of the interviewees expressed a more systematic approach to introduce object oriented analysis and design. However, the students were not required to practice these approaches themselves.

For educators in upper secondary school the choice of problem domain is important. It is vital to keep the interest and attention of the students with something considered fun, and at the same time find a context that would assist in teaching and learning object oriented problem solving. The two university educators aiming for a more conceptual approach, preferred contexts that are software oriented.

It seem that in some instances it is a language being taught, rather than a paradigm. This can not be criticised per se, but for all ten interviewees the task given, according to syllabi, was to teach object oriented programming, or object orientation. If the mission is to teach object oriented programming, we can not restrict ourselves to teaching the syntax of a language and do it in an basically imperative fashion.

If object orientation is to be taught properly, as a problem solving approach with a distinct focus on proper objects, it is my belief that some discussion of object oriented analysis and design need to be present. Introducing a strategy or method for choosing objects in a proper fashion, to be able to make reasonable abstractions, modeling entities in the problem domain, is absolutely necessary.

> *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)

Threats to validity There is always a possibility that the mere fact that a certain research area is discussed influences the statements of the respondents. Some might have a nagging feeling of being evaluated and might be restricted in their description of certain issues. This is unavoidable.

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 examples for novices has been the main focus of my research for the last three years. Being aware of this, I have made an effort to set aside my preconceptions of object oriented quality.

The question of validity in qualitative research is a matter of standards to be upheld as ideals (Whittemore et al., 2001).

In this study the classification of statements 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 major part of differences in classifications, was due to the interpretation of the aspects of the theme *Examples*, which is not part of the study presented here.

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.

Bibliography

Clancey, M. (2004). *Misconceptions and Attitudes that Infere with Learning to Program*, pages 85–100. Taylor & Francis.

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

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., 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.

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

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.

Guzdial, M. (2008). Paving the way for computational thinking. *Commun. ACM*, 51(8):25–27.

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.

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., and Herman, G. L. (2010). Identifying student misconceptions of programming. In *SIGCSE '10: Proceedings of the 41st ACM technical symposium on Computer science education*, pages 107–111, New York, NY, USA. ACM.

Morse, J. M. (1991). Approaches to qualitative-quantitative methodological triangulation. *Nursing research*, 40(2):120–123.

Nordström, M. (2010). Educators' views on object orientation. *(to be submitted)*.

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

Shank, G. and Villella, O. (2004). Building on new foundations: Core principles and new directions for qualitative research. *The Journal of Educational Research*, 98(1):46–55.

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.

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

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

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

Paper VII

Improving OO Example Programs

Improving OO Example Programs

Marie Nordström and Jürgen Börstler, *Member, IEEE*

***Abstract*—When teaching object oriented programming, educators rely heavily on textbook examples. However, research shows that such examples are often of insufficient quality regarding their object-oriented characteristics. In this paper, we present a number of guidelines for designing or improving object oriented example programs for novices. Using actual textbook examples, we show how the guidelines can help in assessing and improving the quality of object oriented example programs.**

***Index Terms*—Object oriented programming, Example programs, Guidelines, Quality, Education.**

I. Introduction

EXAMPLE programs are a key resource for the teaching or learning of programming [1], [2]. It has been argued that object orientation is a "natural" way of thinking and therefore well suited for problem solving using program development. However, several studies question this claim [3]. Novices, for example, have more problems understanding a delegation-based control style, which is central to object-orientation, than a centralized one. This adds to the difficulties of teaching object orientation.

We have little scientific theory or evidence guiding us in how to introduce object orientation. However, there should be no doubt that we need to strive for examples that emphasize the general characteristics of object orientation. Research shows, though, that the overall quality of object oriented example programs in introductory textbooks is insufficient [4], [5]. The quality of common example programs, like the famous "`HelloWorld`" program, have been critically discussed for a long time [6] and there have been ongoing debates on the object-orientedness of these and similar examples [7]–[9]. However, all of these discussions have focused on superficial technicalities, rather than the inherent object oriented qualities (and suitability) of the examples.

In this paper, we present quality guidelines for object oriented example programs for novices that have been derived from key object oriented concepts and design principles [10]. We then show how these guidelines can be used to assess and improve typical examples from introductory programming texts. The reader should note that programming is not an exact science and there are varying opinions on and perceptions of quality. Improving an example is always easier than developing it in the first place. This paper suggests a critical attitude towards the object oriented quality of example programs. If we want students to write quality code, we have to give them "good" examples and insist on commonly agreed object oriented principles, guidelines, and rules even in small scale examples for novices.

Marie Nordström is with the Department of Computing Science, Umeå University, Umeå, Sweden. Corresponding e-mail: marie@cs.umu.se

Jürgen Börstler is with the Department of Computing Science, Umeå University, Umeå, Sweden (on leave) and with the School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden. Corresponding e-mail: jurgen.borstler@bth.se

Manuscript received November X, 2010; revised month date, 2010.

II. Related Work

Textbooks are a major source for example programs and also work as a reference for how to solve specific problems.

Becker [11] reviewed 16 introductory object oriented programming texts and analyzed how objects and classes are introduced, how I/O is handled, and how they support software engineering concepts. His results show that it is difficult to find a text that meets every need, but that there are books that do a good job.

McConnell and Burhans [12] have examined the coverage of basic concepts in programming textbooks over time and observed a shift in the amount of coverage of various topics with each new programming paradigm. With object orientation they observe a decrease in the treatment of subprograms but also a decrease in basic programming constructs.

De Raadt et al. [13] examineed 49 textbooks used in Australia and New Zealand and found quite poor compliance with the ACM/IEEE curriculum guidelines. One explanation they offered for the poor compliance was that many texts are more focused on the syntactical details of a programming language than on conveying a more holistic view of programming as a problem solving tool.

In a multi-national study Lister et al. [14] concluded that few students are able to articulate the intent of code when asked to "think out loud" while taking a multiple-choice questionnaire.

Research also revealed that students hold a range of misconceptions about object orientation. Ragonis and Ben Ari [15] identified seven such misconceptions about program flow in object orientated programs. The same authors also present a comprehensive list of frequently observed difficulties and misconceptions of novices along with probable sources of these problems [16].

Holland et al. [17] discuss misconceptions concerning the concept of an object. They present a number of misconceptions that might relate to particular features of example programs and suggest guidelines for example construction to avoid these problems. Fleury [18] identifies a number of student constructed rules, where students generalize from valid models in erroneous ways.

Börstler et al. [4], [19] developed a checklist for evaluating the quality of object oriented examples for novices. Their checklist covers three aspects of example quality; technical quality, object-oriented quality, and didactic quality. Object oriented quality is captured by the following five quality factors: Reasonable Abstractions (O1), Reasonable State and Behaviour (O2), Reasonable Class Relationships (O3), Exemplary OO code (O4), Promotes "Object Thinking" (O5).

Results of a large scale study shows that the object oriented quality of example programs in common introductory text books is low [4], [5].

III. Educational Heuristics for OO Examples

Based on key concepts found in literature, and on key object oriented design principles used by the software developing community, a number of educational heuristics were developed and described in [10]. The intention of these heuristics is to support the design of exemplary examples, with respect to the characteristics of object orientation, and at the same time conceptually addressing the particular needs of novices. Based on the experiences with the design, evaluation, and use of an evaluation tool for object oriented examples for novices [4], [5], [19], further fine-tuning of these heuristics have resulted in five *Eduristics*.

The Eduristics are targeted towards general design characteristics, which means that more detailed practices, like keeping all attributes private, are not stated explicitly. Furthermore, they are designed to be independent of a particular pedagogic approach (objects first/late, order of concepts, ...), language, or environment.

In short, the Eduristics can be summarized as follows:

1) Model Reasonable Abstractions
   - 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 [20].
2) Model Reasonable 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 textual representations.
3) Emphasize Client View
   - Promote thinking in terms of services that are required from explicit clients.
   - Separate the internal representation from the external functionality.
4) Promote Composition
   - Emphasize the idea of collaborating objects; use non-trivial attributes to emphasise the distribution of responsibilities.
   - Do not use inheritance to model roles [21].
   - Inheritance should separate behaviour and demonstrate polymorphism.
5) Use Exemplary Objects Only
   - 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: avoid 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 [22]).
   - Avoid shortcuts.

Using exemplary objects in particular means to support students in their generalizing from concrete examples to general properties. Emphasising improper role models might lead to erroneous generalisations or misconceptions. Many introductory (Java) textbooks use classes like `String` or `Math` to introduce the concepts of object and class. However, neither of them is a good role-model of a class. String-objects are immutable and can therefore not be shared, which means that they do not behave like "proper" objects. `Math` is just a container for methods and cannot even be instantiated.

We are well aware that real software features more non-exemplary than exemplary objects (see for example [23]), but from a teaching and learning point of view initial examples need to be prototypical for a concept [24], [25].

Our five Eduristics correspond well with the object oriented quality factors of the evaluation tool used in [4], [5]. We therefore argue that they help to increase the object oriented quality of examples for novices.

IV. A Closer Look at Examples

An important category of examples is the first user defined class (FUDC). For a novice, this example sets the stage for a typical class and must therefore be carefully chosen. Defining a small, simple, and easy to understand example, that exhibits high object oriented quality is a challenging task, in particular since novices only have a very small repertoire of concepts and syntactical constructs. As part of a larger study, a number of FUDCs from common introductory textbooks were evaluated and analysed in [4], [5]. The results of this analysis are discouraging; many examples score low in particular regarding their object oriented quality.

In the following sub-section, we use the Eduristics to examine a typical FUDC-example. This is followed by a more general discussion on common deficiencies in FUDCs. Alternative designs are proposed and finally we suggest a number of suitable abstractions to use in examples with appropriate contexts.

A. The `BMI`-example

Fig. 1 shows an example of a FUDC from a common textbook [26, chpt. 10].

The class is modeling Body Mass Index[1] objects. The class is presented as an attempt to make a previously introduced static method (see below) for calculating the BMI reusable.

[1]"Body Mass Index (BMI) is a number calculated from a person's weight and height. For adults 20 years old and older, BMI is interpreted using standard weight status categories that are the same for all ages and for both men and women. For children and teens, on the other hand, the interpretation of BMI is both age- and sex-specific" http://www.cdc.gov/healthyweight/assessing/bmi/adult_bmi/index.html

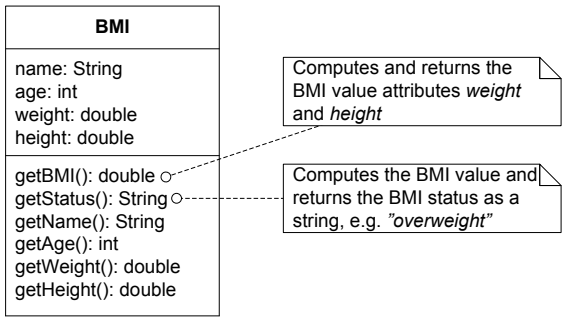


Fig. 1. UML class diagram for `BMI`.

```
public static double getBMI(
    double weight, double height);
```

The example text argues that if there is a need to associate the weight and height with a person's name and birth date, the ideal way to couple them, would be to create an object that holds them all. According to the text, this example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. Using our Eduristics, we would evaluate the object oriented qualities of this example in the following way:

1) *Model Reasonable Abstractions*: Abstraction is a mechanism to cope with complexity. A good abstraction focuses on the essential properties of a phenomenon while ignoring irrelevant details. BMI is basically a value or value/weight status pair and the `BMI`-class could be said to focus on these essential properties. However, it is difficult to understand why `name` and `age` should be properties (state) of a `BMI`-object. In the present case, the attributes `name` and `age` are never even used, although BMI's for children and adults are calculated differently. It is therefore unclear what `BMI` is actually modelling.
2) *Model Reasonable Behaviour*: The `BMI`-class is just a "wrapper" for some immutable values set on instantiation. The state of an `BMI`-object never changes, although it would seem reasonable that at least `age` and `weight` should be mutable. Nevertheless, the BMI-value and its corresponding status are recalculated for every call of the methods `getBMI()` and `getStatus()`. This is highly unreasonable behaviour, not only from the class' interface point of view, but also from an implementation point of view.
3) *Emphasize Client View*: `BMI`-objects do not offer any significant services. The only thing clients can do is to group a set of immutable values and ask `BMI`-objects for "their own values":

```
BMI bmi = new BMI(...);
double bmiValue = bmi.getBMI();
String bmiStatus = bmi.getStatus();
```

Martin argues that a model can only be meaningfully validated in terms of its clients [27]. In the `BMI` case it is, however, difficult to imagine a client.

5) *Use Exemplary Objects Only*: `BMI`-objects do neither have mutable state nor meaningful behaviour. Although the accompanying test program instantiates two `BMI`-objects that does not help much to emphasise the difference between objects and classes, since the only thing done with the objects is to get their BMI-values and print them to `System.out`. The example does not promote object thinking; the objects are no autonomous entities with clearly defined responsibilities.

This example also illustrates the importance of naming. Judging from the attributes of the `BMI`-class, it seems more adequate to name this class something like `PersonalData`, which is responsible for keeping track of the weight-history for an individual. Then calculating the BMI would be a smaller task to be performed for some supplied service. However, naming is also dependent on the context. The name of a class must adequately describe a phenomenon from the problem domain's point of view. The name should also give proper associations of what to expect from objects of this type.

B. Common example deficiencies

Simple "structures", like `Clock`, `Card`, `GradeBook` etc. are commonly used as early examples in introductory programming textbooks. They are characterised by a number of attributes of basic types and `String`, and methods to set and get those attributes individually. See Fig. 2 for typical designs, taken almost straight out of common introductory Java textbooks. Formally, all examples in Fig. 2 follow the fundamental notion of a class as encapsulating data and methods that operate on that data. However, we would argue that a method does not really operate on the data, if the only thing it does is to assign or retrieve the value of an instance variable.

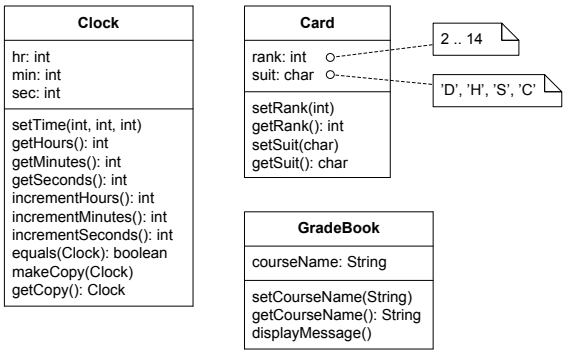


Fig. 2. UML class diagram for common container classes.

All three abstraction in Fig. 2 are reasonable. They model intuitive entities that can be easily placed in some context, but we would still argue that they lack certain object oriented qualities. Primarily, they do not exhibit reasonable behaviour (Eduristic 2), and they do not emphasize client view (Eduristic 3).

However, it is possible to improve examples like the ones above without too much effort. The `Clock`-example might be turned into a `StopWatch` (see Fig. 3), or a `ClockDisplay` (see Fig. 4) that both add reasonable behaviour beyond just set- and get-methods. The `ClockDisplay`-example goes even

a step further and shows non-trivial collaboration between objects without making the example overly complex.

The `Card`-class could be a central class of some card game, but this would require substantial extensions, both in size and complexity. One would at least need classes for rank and suit to complete the actual card abstraction (see [28] for an excellent discussion on when to make a type). Furthermore, additional classes for deck and hand objects might be needed to do something meaningful with `Card`-objects.

A `GradeBook`-object basically is a container for student-grade pairs. Reasonable behaviour could for example be added by methods for the calculation of average scores etc. However, it would still basically be a container, but now with a "bloated" interface/protocol that violates the Single Responsibility Principle [27].

C. Alternative clock-designs

To use the idea of a clock, and still make it an exemplary example, we have found two appealing designs. The first example is the timer, or the stopwatch, see Fig. 3.

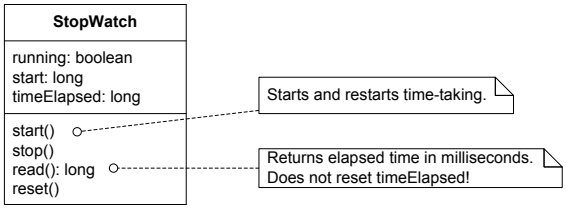


Fig. 3. Class diagram for `StopWatch` [29].

In this design, the actual representation of time is delegated to a system supplied service. The state is easy to understand, either the timer is running, or not. Time is not updated continuously by the object, but calculated when needed (method `read`). The possibilities to accumulate time, and to reset the timer provides clients with the typical features one would expect from a timer or stop watch. This makes `StopWatch` an intuitive and useful abstraction with reasonable behaviour.

Another example of a simple and elegant design, is the `ClockDisplay`-example, shown in Fig. 4.

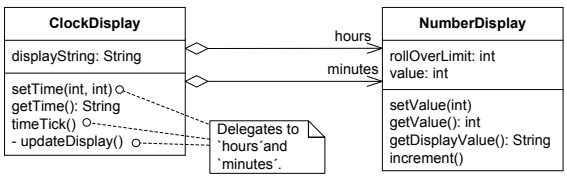


Fig. 4. Class diagram for `ClockDisplay` and `NumberDisplay` [30].

This example even illustrates collaborating objects, without adding much to the overall complexity of the example. Using instance variables of class-type early on makes it easier for novices to understand that instance variables don't need to be of basic types. Although this is a common novice misconception [17], we found that early examples almost exclusively feature instance variables of basic types.

D. Further Examples

In this section we give a short list of suggestions for suitable early examples.

- `Die`: A simple, intuitive abstraction that can be easily placed in meaningful contexts and it is easy to imagine applications using several Die-objects. Rolling a die is a meaningful non-trivial behaviour. Die-objects could even be instantiated with different numbers of faces to illustrate that quite different objects can be instantiated.
- `BankAccount`: Can be a very good example, if used with care. A bank account is a simple and intuitive abstraction and bank account-objects have simple but non-trivial behaviour. It is also fairly easy to imagine meaningful contexts and clients. However, educators must be aware that specializations of bank accounts are localized phenomena and students might have a hard time to figure out the meanings of checking and saving accounts.
- `ClickCounter`: A very simple abstraction modelling a manual counter that increases a value on every click. Has many features in common with the `StopWatch` shown in Fig. 3, but is even simpler.
- `TrafficLight`: Slightly more complicated abstraction that illustrates state dependent behaviour.

Further suitable examples can be derived from situations where we—out of convenience—usually just use variables of some base type. For example to model monetary values, it is very common to use just a variable of floating point type. However, money arithmetic works slightly different than floating point arithmetic. Furthermore, things will become complicated in case one needs to handle different currencies. Fowler [28] presents an interesting discussion on this subject and recommends to define a type (class) whenever some entity needs special behaviour in its operations that a primitive type do not have.

V. Conclusion

The object oriented quality of examples for novices is often insufficient. To prevent students from making premature erroneous generalisations, we find it important to always be faithful to the object-oriented paradigm. Since students will use examples as role models, they must not contradict the concepts and rules we intend them to pick up. In this paper we have described how the design of examples can be enhanced by the use of educational heuristics. Small details can often make a big differences when it comes to object oriented quality. When dealing with examples for novices, it is important to respect that every example serves two purposes. On one hand it demonstrates a particular concept or feature of the paradigm or the language. On the other hand it is also a stepping stone in conveying the general idea of the paradigm, and this must be given appropriate attention.

An important ingredient of an example is its context. A good abstraction focuses on the essential characteristics of some object from a particular point of view [27], [31]. Without this point of view, it can be difficult to make sense of an abstraction. The context, or cover story [4], provides meaning and motivation and supports novices in understanding of the

source code: What problem is being addressed and what is the reason for this particular solution?

REFERENCES

[1] J. Anderson, R. Farrell, and R. Sauers, "Learning to program in LISP," *Cognitive Science*, vol. 8, no. 2, pp. 87–129, 1984.

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

[3] M. Guzdial, "Paving the way for computational thinking," *Commun. ACM*, vol. 51, no. 8, pp. 25–27, 2008.

[4] J. Börstler, M. S. Hall, M. Nordström, J. H. Paterson, K. Sanders, C. Schulte, and L. Thomas, "An evaluation of object oriented example programs in introductory programming textbooks," *Inroads*, vol. 41, pp. 126–143, 2009.

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

[6] R. Westfall, "'hello, world' considered harmful," *Communications of the ACM*, vol. 44, no. 10, pp. 129–130, 2001.

[7] CACM, "Hello, world gets mixed greetings," *Communications of the ACM*, vol. 45, no. 2, pp. 11–15, 2002.

[8] M. H. Dodani, "Hello world! goodbye skills!" *Journal of Object Technology*, vol. 2, no. 1, pp. 23–28, 2003.

[9] CACM Forum, "For programmers, objects are not the only tools," *Communications of the ACM*, vol. 48, no. 4, pp. 11–12, 2005.

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

[11] B. Becker, "Pedagogies for CS1: A survey of java textbooks," 2002.

[12] J. J. McConnell and D. T. Burhans, "The evolution of CS1 textbooks," in *Proceedings FIE'02*, 2002, pp. T4G–1–T4G–6.

[13] M. De Raadt, R. Watson, and M. Toleman, "Textbooks: Under inspection," University of Southern Queensland, Department of Maths and Computing, Toowoomba, Australia, Tech. Rep., 2005.

[14] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hame, M. Lindholm, R. McCartney, J.-E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," *SIGCSE Bull.*, vol. 36, no. 4, pp. 119–150, 2004.

[15] N. Ragonis and M. Ben-Ari, "On understanding the statics and dynamics of object-oriented programs," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005, pp. 226–230.

[16] ——, "A long-term investigation of the comprehension of OOP concepts by novices," *Computer Science Education*, vol. 15, no. 3, pp. 203–221, 2005.

[17] S. Holland, R. Griffiths, and M. Woodman, "Avoiding object misconceptions," in *Proceedings of the 28th Technical Symposium on Computer Science Education*, 1997, pp. 131–134.

[18] A. E. Fleury, "Programming in java: Student-constructed rules," in *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, 2000, pp. 197–201.

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

[20] A. J. Riel, *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[21] M. Fowler, "Dealing with roles," in *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP)*, 1997.

[22] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol. 6, no. 5, pp. 38–48, 1989.

[23] J. Gil and I. Maman, "Micro patterns in java code," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. San Diego, CA, USA: ACM, 2005.

[24] R. K. Atkinson, S. J. Derry, A. Renkl, and D. Wortham, "Learning from examples: Instructional principles from the worked examples research," *Review of Educational Research*, vol. 70, no. 2, pp. 181–214, 2000.

[25] R. D. Tennyson, F. R. Woolley, and M. D. Merrill, "Exemplar and non exemplar variables which produce correct concept classification behaviour and specified classification errors," *Journal of Educational Psychology*, vol. 63, no. 2, pp. 144–52, 1972.

[26] Y. D. Liang, *Introduction to Java programming : comprehensive version*, 8th ed. Upper Saddle River, N.J.: Pearson Education, 2009.

[27] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Addison-Wesley, 2003.

[28] M. Fowler, "When to make a type," *IEEE Software*, vol. 20, no. 1, pp. 12–13, 2003.

[29] D. A. Bailey and D. W. Bailey, *Java Elements–Principles of Programming in Java*. McGraw Hill, 2000.

[30] D. J. Barnes and M. Kölling, *Objects First with Java: A Practical Introduction Using BlueJ: International Edition, 4/E*, 4th ed. Pearson Higher Education, 2009.

[31] G. Booch, *Object-Oriented Analysis and Design with Applications, 2nd edition*. Addison-Wesley, 1994.

Marie Nordström is finishing her PhD in Computer Science didactics, after many years of teaching introductory programming to both CS majors and minors.

Jürgen Börstler is a professor of Computing Science with many years of experience in object oriented analysis and design and a strong interest in computer science education research.

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.

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.

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

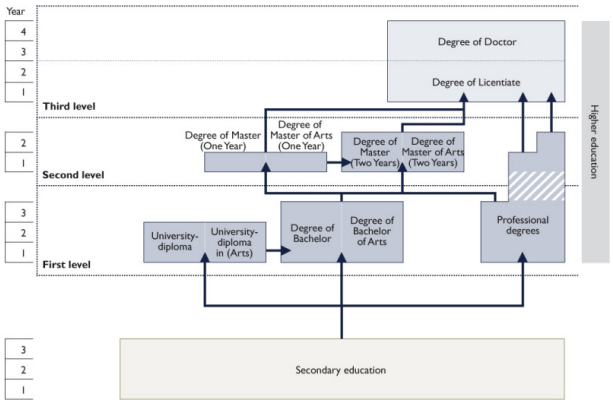


Figure A.1: The swedish educational system.

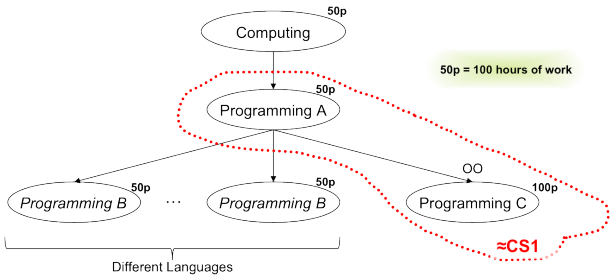


Figure A.2: Computing courses in upper secondary school.

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.