He[d]uristics
– Heuristics for designing object oriented examples for novices

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

March 2009

He[d]uristics

Abstract

The use of examples is known to be important in learning, they should be "exemplary" and function as role-models.

Teaching and learning problem solving and programming in the object oriented paradigm is recognised as difficult. Object orientation is designed to handle complexity and large systems, and not with education in focus. The fact that object orientation often is used as first paradigm makes the design of examples even more difficult and important.

In this thesis, a survey of the literature is made to establish a set of characteristics for object orientation in general. This set of characteristics is then applied to the educational setting of introducing novices to object oriented problem solving and programming, resulting in a number of heuristics for educational purposes, called He[d]uristics. The proposed He[d]uristics are targeted towards educators designing small-scale examples for novices, and is an attempt to provide help in designing suitable examples, not a catalogue of good ones.

The He[d]uristics are discussed and exemplified and also evaluated versus the derived set of characteristics and known common problems experienced by novices.

Sammanfattning

Exempel är viktiga när man ska lära sig programmera, det tycker både utbildare och nybörjare. De ska fungera som modeller och vara "typiska", både vad gäller ämnesområdet och det koncept eller den praktik som ska illustreras. Det här innebär att det kan vara små skillnader mellan exempel som fungerar bra och exempel som i praktiken helt eller delvis motsäger det budskap man vill förmedla. Detta är en av anledningarna till att exempel är känsliga och därför måste vara genomarbetade för att inte ge upphov till felaktiga tolkningar och modeller.

Trots att det har diskuterats hur lämpligt det är, så är det vanligt att objektorientering används för introduktion till problemlösning och programmering. Men att använda objektorientering som första paradigm är delvis motsägelsefullt eftersom objektorientering är en ansats för att hantera komplexitet och för design av stora system. Objektorienteringens erkända principer och formulerade praktiker har storskalighet och komplexitet som utgångspunkt. I en undervisningssituation (lektioner, föreläsningar, konstruktion av undervisningsmaterial och övningsuppgifter, examination etc.) blir villkoren för objektorientering annorlunda jämfört med vid produktion av stora system för industriellt bruk. Vid konstruktion av exempel och övningar riktade till noviser vill man kunna illustrera ett koncept eller en praktik isolerat, –en sak i taget. Det är också önskvärt att exemplen ska vara små i betydelsen få rader kod, dels för att vara lätta att överblicka, dels för att inte dra uppmärksamheten från det som ska exemplifieras. Detta medför att det kan vara svårt att visa på styrkan i objektorientering

I objektorientering är decentralisering ett huvudbegrepp. Objekt ska vara självständiga, aktiva och ansvarstagande enheter i problemlösningen. Ett av problemen när man lär sig programmera objekt-orienterat är att välja ut vad som ska vara objekt i lösningen. Därför måste exemplen ge stöd för hur man väljer typiska objekt. Hur man väljer och designar objekt är inte entydigt, både val och utformning bestäms i hög grad av sammanhanget, dvs. för vilket ändamål behövs dessa objekt och vem vill utnyttja den service som objekten tillhandahåller. Objektens ansvar och beteenden måste vara trovärdiga i ett specificerat sammanhang och det måste framstå som realistiskt att sådana objekt kan förekomma i programvara. Detta innebär ökade krav på exempel i form av sammanhang och tillämpning för att de ska fungera som modeller för objekt-konstruktion.

Den som gör sina första stapplande försök att programmera har inledningsvis en mycket snäv referensram. Vokabulären är mycket begränsad, dels vad gäller begrepp i paradigmet och dels vad gäller språkliga element i det aktuella programmeringsspråket. Det finns alltså väldigt lite att använda sig av för att ge exempel när nya begrepp eller konstruktioner ska introduceras.

Att under dessa förhållanden konstruera exempel som upprätthåller de grundläggande principerna i objektorientering är icke-trivialt. Det är mycket enklare att säga vad man inte borde göra än hur man ska göra.

I det här avhandlingsarbetet görs en genomgång av etablerad kunskap och formulerade praktiker inom objektorientering för att ta reda på vad som anses vara mest karakteristiskt för objektorientering i allmänhet. Detta tolkas sedan till novisens situation med dess specialla bivillkor. Hur kan man vid konstruktionen av exempel upprätthålla de allmänna objekt-orienterade egenskaperna, samtidigt som man tar hänsyn till nybörjarens specifika problem? Baserat på detta föreslås ett antal riktlinjer/heuristiker för att ge stöd för konstruktionen av exempel. Heuristikerna utvärderas mot de koncept, principer och riktlinjer som konstaterats som centrala för objektorientering vid litteraturgenomgången. De föreslagna heuristikerna utreds och exemplifieras i konkreta exempel. Programspråket som används är Java.

Begreppet He[d]uristics är en konstruktion av det engelska ordet *heuristics* med en indikation ([d]) om att det handlar om utbildning, *education*, vilket i olika sammanhang på engelska symboliseras med förkortningen *edu*.

Preface

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

- 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., Nordström, M., Kallin Westin, L., Moström, J.-E., and Eliasson J. Transitioning to OOP–A Never Ending Story. In *Reflections on the Teaching of Programming*, J. Bennedsen, M.E. Caspersen, M. Kölling (Editors), Lecture Notes in Computer Science, LNCS 4821, Springer, 2008, Pages 86-106.

- 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.

Acknowledgements

x

Contents

Chapter 1

Introduction

Examples are important in learning, both teachers and learners consider them to be the main "learning tool" (Lahtinen et al., 2005). In a recent survey of pedagogical aspects of programming, Caspersen (2007) 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)

To be useful, the examples must help the novice to draw conclusions and to make inferences, to make generalisations, from the presented information (Chi et al., 1989; Pirolli and Anderson, 1985). Since examples do not distinguish incidental from essential, 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 superficial surface properties from those that are structurally or conceptually important. By continuously exposing students to **exemplary** examples important properties are reinforced. Students will eventually gain enough experience to recognize general patterns which helps them telling apart "good" and "bad" designs.

In a sharp critique of the contemporary state of business in educational resources for teaching and learning programming, Wirth says:

> *How can one learn such an art [of designing artifacts to solve intricate problems] without master examples worth studying and following? Surely, some people are more, some less gifted for good design, but nevertheless, the proper teaching, tools, and examples play a dominant role.* (Wirth, 2002)

Though largely debated, object orientation is commonly used for introducing problem solving and programming to novices. How to do this is not straightforward. The strength of object orientation lies in the handling of complexity in the design of large-scale system, with high demands on maintenance, efficiency and reusability. The educational situation however, is rather different. Introductory examples are small. The design space is restrained because of the limited frame of reference of

the novice, the limited number of syntactical elements available, and the fact that the number of lines of code preferably should be kept to a minimum.

In this thesis, we investigate principles, metrics, heuristics, patterns, code smells and similar concepts proposed by the software community. These are established practices and constitute condensed experience, and should therefore influence the design of examples for novices. We argue that examples developed according to established practices and experience will lead to suitable role-models. The difficulty is to design examples showing the strength of object orientation, and at the same time avoiding overly complex examples that leave the novice behind. If the example fails to, at least, indicate the strength, then novices may conclude that object oriented design introduces complexity rather than solving a problem, whereas when the example is too complex, students fail to understand the overall big picture.

Because of the constraints of the educational context, 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/exercise intended for novices in order to present or illustrate a certain concept or feature of object oriented problem solving and programming.

Learning problem solving and programming seems to be more difficult in the object oriented paradigm than in the imperative paradigm. 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 by 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). This adds to the difficulty of teaching object orientation, since distributed responsibilities is one of the major characteristics of object orientation.

There are diverging opinions of the consequences of different educational approaches, such as the order of concept presentation, objects-first vs. object late, responsibility driven design or domain entity driven design, the use of graphics and so on (Bruce, 2004). However, claims in favour and against objects-early have not been validated by research, see (Lister et al., 2006). We have little scientific theory and evidence to support us in deciding on how to introduce object orientation. An interesting example of the quasi-type of discussions often replacing theoretical ones, is the discussion on common examples, the 'HelloWorld'-type, that was imitated by Westfall (2001) in Communications of the ACM. There has been an ongoing debate on the object-orientedness of these type of examples (CACM, 2002; Dodani, 2003; CACM, 2005). Surprisingly, the discussion was focused on how to adjust the 'HelloWorld'- example to be more of an object, rather than the object oriented qualities of the example.

The goal of this work is to define a number of heuristics, called **He[d]uristics**, to aid in the design of small-scale examples.

We present a literature survey investigating the characteristics and principles for object orientation in general. The proposed He[d]uristics are then based on the findings of this survey.

It is our goal to keep the discussion focused on the object oriented quality of examples. It should be possible to use the proposed He[d]uritstics regardless of choice of instructional design. Concepts could be introduced in various sequences

and still be discussed from a general object oriented point of view.

However, choosing object orientation as paradigm must put the focus on objects. This means that we emphasize collaborating objects as the major component of object orientation (Booch, 1994). Even though much of the discussion is applicable to object orientation regardless of programming language, some of the arguments are based on the use of Java.

The problem of teaching object orientation to novices is not new, and good points have been made by many excellent professionals and educators, but not collected and applied in a systematic way. Based on commonly agreed upon object oriented principles, we propose a number of practical advice along with critical aspects of examples. The presented educational heuristics, or He[d]uristics, should aid in avoiding common pitfalls when designing small-scale applications.

Investigating this problem, we found it much easier to express the "no-no's" than to give constructive advice. This work is an attempt to be constructive, and to suggest practical guidelines for educators.

> *In software too, no book advice can replace your know-how or inge-nuity. The principal role of a methodological discussion is to indicate some good ideas, draw your attention to some illuminating precedents, and alert you to some known pitfalls.* (Meyer, 1997)

Outline

In Chapters 2–7, a survey of related work is presented and discussed. At the end of each chapter a short summary of factors influencing the design of small-scale examples is made. For the reader acquainted with the established practices and acknowledged methods, or interested in getting faster to the proposed He[d]uristics, it should be possible to read only the summaries of these chapters as background for the He[d]uristics.

The He[d]uristics are presented in Chapter 8. Examples and an extensive discussion follows in Chapter 9.

Chapter 10 contains an evaluation of the He[d]uristics versus the findings of Chapters 2–7.

Conclusions and future work concludes the thesis.

An overview of the structure of the thesis is shown in Figure 1.1.

Introduction   Problem statement

Related work/
Characteristics of OO   Background

OO Advice

Principles

Heuristics and Rules   Established practicies -
                                            How do they apply
Design Patterns and Code Smells   to small-scale?

Metrics

He[d]uristics   Proposal

He[d]uristics in Practice   Interpretation and use of
                                            He[d]uristics

Validation

Conclusions & FW
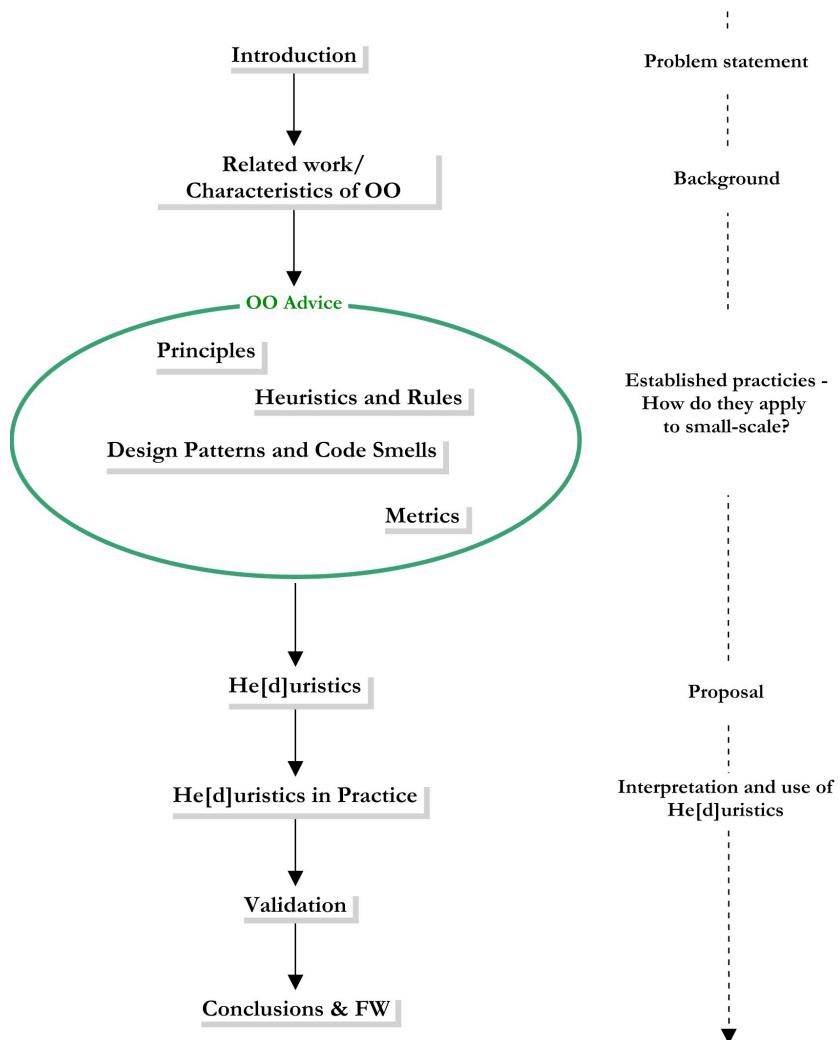
Figure 1.1: Structure of the Thesis

Chapter 2

Related Work

2.1 Introduction

The difficulty in teaching the object oriented paradigm to programming novices is nowadays acknowledged by the community of Computer Scientists. Computer Science Education (CSE) research is gaining more and more interest, but we still know too little about the difficulties and especially how to meet them. There are however, attempts to investigate different aspects of the problems we face. Understanding cognitive difficulties of object orientation is important and will aid in our educational efforts. Practical experiences and observation of introducing object orientation have been documented and analysed. One interesting approach is the measurement of cognitive and conceptual difficulties of introductory programming and problem solving. Other approaches are to investigate the understanding of basic concepts, and misconceptions when applying the conceptual model inadequately. Reports on poor learning behaviour and harmful examples are adding to our understanding of pitfalls, and can contribute to constructive instructional advice.

In this chapter we give a number of examples on research connected to the area of programming, some with general applicability to learning, and some with object orientation in focus.

2.2 General Learning Aspects

Cognitive load theory is focused on the limitations of working memory and how this is affecting the construction of knowledge. The construction of knowledge is admittedly no simple task to investigate. However, a large body of knowledge has been collected in cognitive science and learning theory, and is slowly getting its way into CSE research.

2.2.1 Cognitive load

Cognitive Load Theory (CLT) is said to be

> *a major framework for investigations into cognitive processes and instructional design.* (Paas et al., 2003)

The theory is concerned with the limitations of working memory and the forming and use of schemas (cognitive constructs that incorporate multiple elements of information into a single element with a specific function) in long-term memory to enhance working memory efficiency.

Cognitive Load (CL) has been defined based on the notion of *Element interactivity.*

Element interactivity Low-element interactivity are elements that can be understood and learned individually without connections to any other elements. Each element of high-element interactivity can be learned individually but not understood until all the elements and their interactions are processed simultaneously.

Three types of CL have been defined:

Intrinsic Cognitive Load The element interactivity imposes demands on working memory that are intrinsic to the material being learned.

Extraneous Cognitive Load The formulation of the instructional procedures lead to unnecessary cognitive load.

Germane Cognitive Load Mental work imposed by instructional activities that benefits the instructional goal.

Intrinsic CL can only be reduced by omitting essential interactive elements temporarily from the material being learned. Extraneous CL is most important when intrinsic CL is high, because the two forms are additive. Instructional designs intended to reduce CL are therefore primarily effective when element interactivity is high. The difference between extraneous and germane CL is that certain instructional activities can benefit the instructional goal in contributing to increase the cognitive resources devoted to a task.

Apart from this, effort and motivation can lead to increase in cognitive resources made available.

2.2.2 Knowledge acquisition and conceptual change

Looking at the more general research on science education, it seems to be consensus on the way we internalise new knowledge. The constructivist model is used for modelling our learning. In this model it is central to understand how concepts change when confronted with new information or ideas. Hewson (1981) formulates the following central issue :

> *Under what conditions will an individual holding a set of conceptions of natural phenomena, when confronted by new experiences, either keep his or her conceptions substantially unaltered in the process of assimilating these experiences or have to replace them because of their inadequacy?"* (Hewson, 1981)

In an individual a conceptual change takes place in different ways:

- addition of new concepts through further experiences, personal development and contact with others

- reorganisation of existing conception, triggered externally by some new idea and triggered internally as the result of some process of thought

- rejection of existing conceptions as a result of reorganisation and displacement of some new conceptions

Important implications for teaching:

Meaning-making The knowledge people possess is of critical importance in their attempt to interpret their experiences.

Sense-making Does the experiences make sense, and if not, why?

Misconceptions Individuals construct different conceptions from the same input.

The model presented by Hewson suggests that while introducing new concepts it is absolutely necessary to address alternative conceptions at the same time to support conceptual change.

2.2.3   Worked examples

> *A worked example is a step-by-step demonstration of how to perform a task or solve a problem* (Clark et al., 2006)

Studying worked examples puts less extraneous cognitive load (Subsection 2.2.1) on the novice than working with practice problems. Clark et al. provide four guidelines (Guideline17-20) specifically for the design and display of worked examples. The approach is designed to accelerate expertise and the strategy is to modify the examples as the learner advances.

Guideline 17: *Replace Some Practice Problems with Worked Examples*
One suggestion is to have *worked example-problem pairs*. A worked example is then immediately followed by a similar practice problem, and the lesson alternates between these two. When studying worked examples, working memory can build a schema, and the analogy is available when solving the practice problem. Without the use of such an example when actively solving a problem most of the resources are consumed by trying to understand what the best approach might be, instead of building new schema.

Guideline 18: *Use Completion Examples to Promote Learning Processing*
To avoid the problem of learners skipping worked examples, it could be wise to use *Completion examples*. In this type of example some steps are demonstrated and other steps are completed by the learner, as in a practice problem.

Guideline 19: *Transition from Worked Examples to Problem Assignments with Backwards Fading*
As the learner is getting more skilled, the worked examples and the completion examples start to be less efficient, or even detrimental. In this case Backwards Fading is likely to accommodate a gradual learning process. Backward fading is when completion examples evolve into full problem assignments by gradually increasing the number of steps completed by the learner. This process can be applied during a single lesson, starting with a fully worked

example to provide a model, followed by completion examples that gradually fades into a full problem assignment. For an experienced learner worked examples can depress learning because of the need for mental resources to study the demonstration than to work with a problem directly. This is due to the fact that once a schema has been established for working problems, going through more worked examples can at best solidify the schema. This mental activity is redundant and further learning is disrupted.

Guideline 20: *Display Worked Examples and Completion Problems in Ways That Minimize Extraneous Cognitive Load*
A poorly formatted worked example could add extraneous CL, which is important to avoid. Integrated text formats (audio, visual, written) can be used to distribute the load between the visual and auditory storage centers of working memory. When using audio, it is recommended to have visuals supporting the attention of the eye as the description moves on.

There is no single way to design the instructional procedures[1] since the change in skill affects the behaviour of the learner. Cognitive load methods are schema substitutes and this explains why more experienced learners find methods designed for novices redundant. They are looking for known patterns.

> *Instructional methods that manage [cognitive] load effectively for novices are no longer needed once learners gain more expertise.*
> (Clark et al., 2006)

2.3 Cognitive Aspects of Programming

Attempts to investigate the cognitive complexity of programming started in the 1970'ies. Among them are studies of practices and techniques, as well as attempts to proposing methods for reducing the cognitive load of the programmer.

White and Sivitanides (2005) investigates the cognitive differences between procedural programming and object oriented programming. Based on research on brain hemisphere dominace, with studies showing that left-brain dominace correlated with procedural programming, they performed a corresponding study for object oriented programming. The cognitive hemispheric dominance is measured by the Hemispheric Mode Indicator (HMI) that stresses cognitive aspects. The characteristics for left/right hemispheric modes include: rational vs. intuitive, logical vs. hunches, differences vs. similarities, and objective vs. subjective judgements. Their results support the idea that object oriented programming is not affected by hemispheric dominance, i.e. object oriented programming suits left- and right-brain dominant programmers equally well.

Robins et al. (2003) provides a survey of different approaches to form models about the cognitive aspects of programming. Programming plans, schemas, programming strategies, the difference between experts and novices, are areas reported on.

A model for the cognitive complexity of the programming process, based on a review of the current metrics and their theoretical approach, has been proposed by

---

[1]Instructional procedures is a part of the instructional design, but not dealing with the question of what concepts to address and in what order.

Cant et al. (1995). In this model it is the cognitive processes of the programmer that is central, not the complexity of the final product. Basic is the notion of comprehension. Comprehension is present in all kinds of code related tasks, such as maintaining, modifying, extending, testing and understanding code. The cognitive processes in comprehension are termed *chunking* and *tracing*.

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". Comprehending the chunks is important in this process.

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

Cant et al. use chunking as a model for recognising "program plans", which consists of both an idea of control flow and of variable use. A graphical representation is suggested (the landscape diagram) as well a mathematical one. However, these representations are merely suggestions and not fully operational as is.

In a multi-national study Lister et al. (2004) few students articulate the intent of the code when asked to "think out loud" while taking a multiple-choice questionnaire.

Karahasanovic et al. (2007) have collected a number of studies of program comprehension. Software maintenance is the major target of many studies, but experiments have been criticised for lack of realism. Maintenance task have been know beforehand, and although the results are intended for large systems few of the experiments are made on large-scale programs.

Fleury (2001) investigates how novice programmers construct an understanding of encapsulation and reuse in Java. This is related to program comprehension:

> *Readers of programs, like readers of murder mysteries, must extract plans from scattered information. Readers of programs, like readers of instruction manuals, must typically work with a general formulation, not with a list of steps for specific cases.* (Fleury, 2001)

Findings of this study shows that multiple classes raises the effort to comprehend the program flow. Other sources of difficulties are parameters, because they make the transfer of values less transparent, and slightly more general classes that can be reused, since they are more difficult to comprehend. Some instructional advice is given to support educators:

- show good object oriented design from the beginning

- emphasize abstract comprehension on a higher level than lines of code (chunking)

- use case studies to make sure that novices are forced to deal with deciding on program organisation

- encourage students to reflect on their learning

- have students work with poorly encapsulated or duplicated code to make the point

- require the use and reuse of library classes

Cognitive aspects of learning object oriented thinking have been discussed in a number of papers. Based on an empirical study Or-Bach and Lavy (2004) proposes a cognitive task taxonomy regarding abstraction and inheritance, see Figure 2.1.
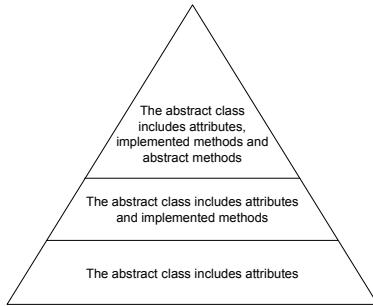
Figure 2.1: Taxonomy for abstraction and inheritance

This is a limited analysis of the general area of abstraction. The population is small (n=33) and only one problem is used for analysis. The analysis is based on simple numeric measures. They conclude that often stated advantages of object orientation are exactly the same issues that make object orientation so difficult for novices. The proposed taxonomy would still be useful, e.g in connection to the structure suggested by Bennedsen and Caspersen (2004), see Figure 2.2.

2.4 Instructional Design for Object Oriented Programming

In this section a number of examples of suggested approaches for teaching object orientation to novices are described.

2.4.1 Conceptual modelling

Working within the small-scale context, it would be of great importance to have an established plan for concept introduction. Suggestions has been made by Bennedsen and Caspersen (2004, 2008). They discuss the lack of structured approach to teach *Conceptual modelling*, which they term the defining characteristic of object orientation. A progressive educational structure using modelling as the driving force is suggested by the authors. The approach is based on an increased complexity in models, see Figure 2.2 for the initial stages. The remaining elements of the conceptual framework, composition and specialisation, is said to be treated in the same way.

2.4.2 Organisation of activities

To guide the instructional design of the introductory programming course Caspersen (2007) lean on nine principles for the organisation of activities at all levels of course

Figure 2.2: Increasing conceptual complexity

design:
1. Consume before produce
2. Present worked, exemplary examples
3. Reinforce specifications
4. Reveal process and pragmatics
5. Provide hands-on opportunities
6. Define progression in terms of complexity of tasks
7. Reinforce patterns and conceptual frameworks
8. Ensure constructive alignment
9. Provide care and support

2.4.3 Principles for teaching novices

Based on the work by Caspersen Gries (2007) is stating five principles for teaching novices:

Principle 1: *Reveal the programming process, in order to ease and promote the learning of programming.*
This is important throughout the entire work of introducing novices to problem solving and programming, and can therefore not be taught in isolation in a single lecture.

Principle 2: *Teach skills, and not just knowledge, in order to promote the learning of programming.*
Transferring knowledge must not be the our aim, but the development of skills.

Principle 3: *Present concepts at the appropriate level of abstraction.*
Describing concepts in terms of the computer can put extra cognitive strain on the learner. One example could be using memory allocation principles and binary storage to introduce variables.

Principle 4: *Order material so as to minimize the introduction of terms or topics without explanation: as much as possible, define a term when you first introduce it.*

Gries says that following this principle almost forces an object-first approach. This is due to the fact that almost any object oriented program deals with classes and object (at least should be). An example of this is to introduce Applets before discussing subclasses.

Principle 5: *Use unambiguous, clear, and precise terminology.*

The lack of vocabulary in object orientation gives rise a number of confusing terms. Inheritance is one, interface another. There are several terms for attributes, common terms are instance variables or class variables.

2.4.4 Focusing on conceptual ideas

Based on the idea of sidestepping syntactical details and focusing on conceptual ideas of object orientation Goldman (2004) suggests a Concepts-first curriculum. To achieve this the IDE JPie (JPie) is used. The curriculum is based on four "big ideas":

Fundamental Abstractions modeling, naming abstraction, types and values, classes and objects, methods and delegation, procedural abstraction with parameters and return values, sequential and conditional execution.

Software Design viewed as a creative process with a specific emphasise on functionality, usability and efficiency.

- Separation of Concerns: Encapsulation, Model/View Separation, and Local Coordinate Systems
- Type Systems: Class Hierarchy Design, Inheritance and Specialization, Polymorphism, and Program Correctness

Algorithms and Data Structures central to computer science, at least on an introductory level, it is necessary to make these tools known to novices.

- Iteration and Recursion
- Use of Fundamental Data Structures
- Persistence

Concurrency and Communication experiences made possible through the environment used.

- Synchronization and Deadlock
- Interprocess Communication

The experiences made are not for groups of CS-majors and are aiming at exposure to common computer science concepts more that learning to program in an object oriented way.

2.4.5   Concept-order in an objects-first approach

Objects first has been claimed in many suggested curricula and in many book-titles. It has turned out that doing objects first is easier said than done. An example for the ordering of concepts in an objects-first approach, was made by Gries at a keynote speech at SIGSCE 2008 (Gries, 2008).

Lecture 01: Expressions, variables, assignment
Lab: practice: types int, double, boolean, string; casting; assignment

Lecture 02: Objects (students see JFrame objects)

Lecture 03: The class and subclass definitions (simple function/proc declarations with return statements and method calls
Lab: write simple function/proc declarations in a subclass of JFrame

Lecture 04: Fields, getter/setter methods, simple constructors

Lecture 05: Static components, the class hierarchy, JUnit testing
Lab: use JUnit test cases to find and fix errors in a given program; practice with static components

Lecture 06: How a method call is executed. if- and if-else statements. local variables.

Lecture 07: Inside-out rule; super-this; stepwise refinement
Lab: practice: write functions (if- and if-else; no loops!)

Lecture 08: Constructors in subclasses; stepwise refinement

Lecture 09: Wrapper classes; stepwise refinement
Lab: Learn about class Vector

Lecture 10: Recursion

Lecture 11: Recursion Lab: Writing recursive functions

Lecture 12: Casting among class-types; operator instanceof; function equals.

Just looking at this course structure does not show the way concepts are discussed. It is mainly a record of when syntactical components are introduced. It might be argued that this approach is not objects-first, since objects only appears in the second lecture. The non-exemplary concepts of setter/getters and static is early in the curriculum and type-checking with the operator `instanceof` is included. Inheritance is used early to support the use of graphical components.

2.4.6   Taxonomy of learning object-technology

Attempts have been made to develop taxonomies for different areas of computer science. Mosley (2005) suggests a taxonomy for learning object technology to be used for sequences of programming courses, see Figure 2.3. Taxonomy is the science of classification according to a predetermined system, with the resulting catalog used to provide a conceptual framework for discussion, analysis, or information

retrieval. In theory, the development of a good taxonomy takes into account the importance of separating elements of a group (taxon) into subgroups (taxa) that are mutually exclusive, unambiguous, and taken together, include all possibilities.
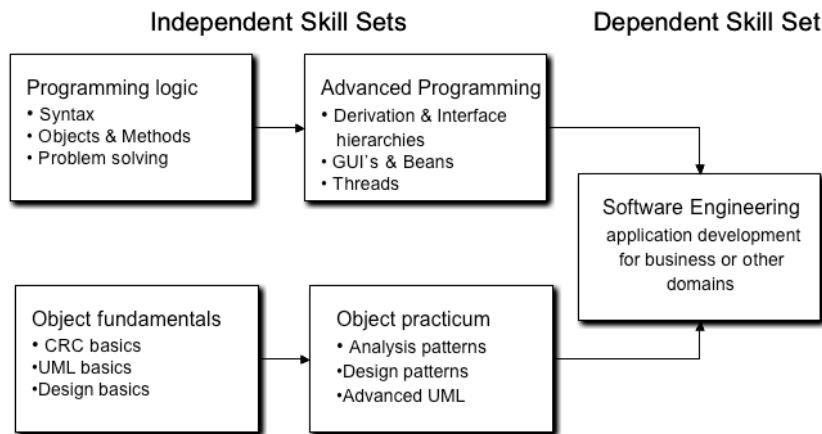
Figure 2.3: Taxonomy for learning object-technology

Basically these sets of skills can be viewed as skills in programming and skills in analysis and design, see Figure 2.4. Calling them independent does not seem quit correct for novices. Mosley's study is based mainly on professional programmers, to whom programming logic is a less complicated issue.
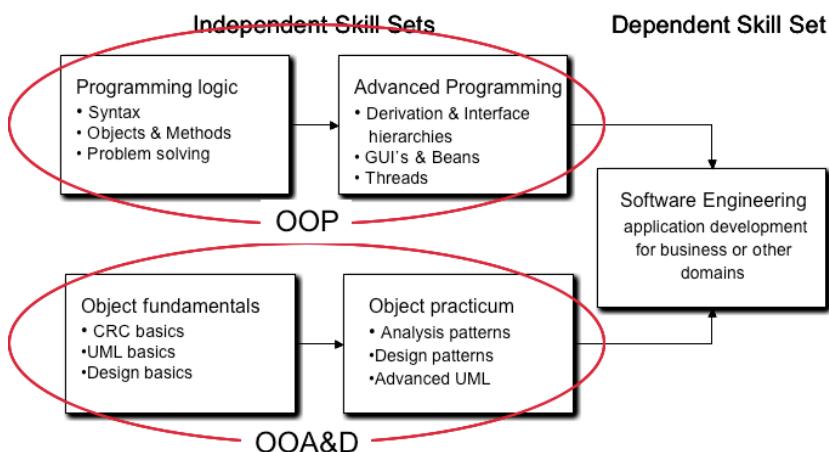
Figure 2.4: Taxonomy for learning object-technology, revised

2.5 Misconceptions

Several educators have documented observed misconceptions. Holland et al. (1997) discusses misconceptions concerning the concept of an object. They present a number of misconceptions and suggestions how to avoid them.

Avoiding object/variable conflation Single attribute classes could make the novice deduct that objects are wrappers for variables. Another problem is examples with classes having all attributes of the same class. This might lead to the misconception that all attributes must be of the same type.

Objects are not simple records One example is the music CD class, that merely stores information like any database record. This makes the behavioural aspect less obvious and might induce the misconception of classes as data-only entities. It is important to show how the behaviour of an object could vary substantially depending on its state.

Work in methods is not all done by assignment Early examples of methods often show assignments. This can be influential for a novice. If it is possible, we should avoid having all attributes of immutable object, such as primitive types. If some attributes have states themselves this misconception is less likely to stick.

Object/class conflation It is crucial to show more than one instance of a class to avoid the mix up of class and object.

Identity/attribute confusion Using the identifier *name* for an attribute could cause a confusion about what the identity of an object is, or the function and name of the reference to the object. This has further implications, i.e. only one variable can reference a certain object at a given time; a variable that references an object can not reference any other object; objects knows who references them; two objects with the same state are the same object; two objects with the same value of an attribute called *name* are the same. These misconception are best dealt with by exposing the learner to counterexamples.

Conflation of textual representation of objects and references to objects This is a common problem since we tend to use `toString` or printing to examine and show the behaviour of a method, and/or the flow of execution in a method. It is also common with problems to distinguish between the visual representation of an object in an IDE (e.g., BlueJ); the object itself; and a reference to that object. References has to be taught with care and as a separate concept along with the concepts of class and object.

In Ragonis and Ben-Ari (2005) a number of frequently observed difficulties and misconceptions are presented along with explanations of the probable source of the problem.

Difficulty 1 – Object state Novice students have difficulties in understanding how the invocation of a method influences the objects state.

Difficulty 2 – Method invocation Understanding how a sequence of method calls relates to solving the problem.

15

Difficulty 3 – Parameters Novice learners does not understand the relation between the formal and the actual parameter. A good point made by Gries (2007) is to use the terms parameter and argument instead.

Difficulty 4 – Return values Novice learners do not understand where the return value of a method goes.

Difficulty 5 – Input instructions Understanding the need for input instruction to the user when working with interactive programming.

Difficulty 6 – Constructors Difficulties in understanding the connections between the constructor declaration, the constructor invocation and the constructor execution.

Difficulty 7 – The overall picture of execution What is happening and when? This could mean difficulties in understanding the relation between the IDE, the code and the execution. Using IDE's makes teachers switch between development and destining, in an iterative fashion. This can contribute to the implicit notion that objects can be changed through the editor, because the results of trying out something in the class gives cause for improvement and thereby changes in the definition of the class.

One important conclusion is that program flow can easily be neglected due to the focus on objects. Sequential executing of statements, including the creation and referencing of objects manipulating the state of objects, must be presented early to avoid some of these misconception. However, it is our firm belief that there is a larger amount of topics that has to be covered when teaching and learning object orientation compared to procedural problem solving and programming.

Fleury (2000) elaborates on student-constructed rules and their view of basic concepts. Attempts are made to find out how certain misconceptions can be explained.

Student–Constructed Rule 1: *The Java compiler can distinguish between same-named methods only if they have differences in their parameter lists.* The principle that names within a class must be distinguished with the help of parameter lists was extended to methods in different classes.

Student–Constructed Rule 2: *The only purpose of invoking a constructor is to initialize the instance variables of an object.* The different handling of memory allocation for primitive values and for objects creates a problem in Java. The declaration of reference-variables is the same as the declaration for primitive variables, while the allocation of space for objects is implicit.

Student–Constructed Rule 3: *Numbers or numeric constants are the only appropriate actual parameters corresponding to integer formal parameters.* Passing explicit values is more easily comprehended, than passing the value of a formal parameter, who at the moment does not seem to have any value.

Student–Constructed Rule 4: *The dot operator can only be applied to methods.* Using encapsulation properly means that the dot operator rarely is applied to an attribute.

An interesting conclusion is that removing "only" or "the only" from the rules makes them true statements. So the construction of the rules are correct rules applied in the wrong way.

The "Who-am-I" problem

Not only are the examples we use important, but also how we use words to talk about programming and problem solving. When talking about problem solving/programming, educators often switch between different views of 'I':

- 'I' am the constructor of the code/class/program

- 'I' am the user/client of the object/program

- 'I' am the one executing the code/program

This can be highly confusing to a novice.

When discussing the design of a a class, it seem like a good idea to think about how the objects are going to be used, which behaviours we want to make available to the "user". This makes us educators switch back and forth between these views. Then when tracing the code to see if it works we switch perspective once more. This might cause the students to develop misconceptions. One could be the belief that objects change state and behaviour by editing the source-code of the class. Another might be that the client of a class could influence the behaviour of the object.

To our knowledge this practical detail of teaching has not yet been addressed.

2.6 Harmful Examples and Poor Learning Behaviour

An interesting collection of *harmful* examples has been collected by Malan and Halland (2004). They identify four common pitfall to avoid when designing examples:

Examples that are too abstract: If it does not make sense to the novice why anyone would need such a class it gives a wrong impression of the concept of a class.

Examples that are too complex: This is an obvious risk when attempting to make examples more realistic. There will be too much disturbing noise distracting the attention from the concept illustrated.

Concepts applied inconsistently: One example is passing attributes as parameters internally in a class, despite the fact that they could be accessed directly.

Examples undermining the concept introduced: More severe cases of undermining. Using data-only classes, with only set and get methods, is one example. This undermines the idea of a class being an autonomous entity, with both state and behaviour.

It is rightfully claimed that "selling" the concept is an important responsibility for educators, especially when working with novices.

Examples as vehicles for learning tendencies is investigated by Carbone et al. (2001). Poor learning tendencies was initiated by the following task characteristics:

Superficial attention: tasks involving copying and modifying code can lead to avoidance in trying to make sense of the information processed.

Impulsive attention: task that do not emphasise the key ideas, or introduces to many unfamiliar concepts, or demands long coding solutions can lead to focusing on the interesting parts and thereby ignoring some major point.

Staying stuck: novices without adequate strategies remain stuck in three different situations:

1. the initial designing stage
2. coding the solution and solving complications
3. run-time errors.

The poor learning tendencies are classified into three categories:

Non-retrieval: no attempt to retrieve ones' own views and understandings relevant to things presented.

Lack of internal reflective thinking: the novice is not reflecting within the boundaries of the subject. Each task, lesson, activity or even instruction is seen isolated from the other sources of information.

Lack of external reflective thinking: the novice is not reflecting outside the boundaries of the subject. The content of the subject is not linked to the outside world or other subjects.

The study performed by Carbone et al. is based on student interviews, tutor comments and student cases. It suggests a number of task-improving measures to be taken.

- Non-retrieval

  - Familiarity.
  - Reinforcement by repetition.
  - Retrieve existing understanding.

- Lack of internal reflective thinking

  - Tie the work into the 'Big Ideas" of the lesson.
  - Build on previous work.
  - Extract the links.

- Lack of external reflective thinking

  - Tasks should be designed to include components of other units.

2.7 Qualities of Examples

The importance of good examples seems to be agreed upon. But how does one tell good from bad? Evaluating examples from an educational point of view is often done on an intuitive basis. Just looking through some popular textbooks or trying to give an example on-the-fly makes it apparent that intuition often is misleading in this case. One attempt to construct a tool in aiding this work is a protocol suggested by **?** and in more detail described in (Börstler et al., 2008). The discussion in this section is to a large extent based on our work during the development of that tool. The suggested tool is based on the following definitions:

An *Example Program* is complete description of a program, i.e., the actual source code in connection to all supporting explanations related to the particular example. This is opposed to code snippets.

A *Good Example* is an example that relates to the needs of novices in both programming and object orientation.

Novices have a limited frame of reference. They rarely have any previous knowledge about (object oriented) programming languages or the programming process. To control the cognitive load and reduce unnecessary complexity, a number of conditions are stated for a good example.

- Example context should be as familiar as possible.

- Examples must be small and/or focus on a single concept.

- object oriented principles must be upheld and enforced.

A number of basic properties are stated for an object oriented example program to be effective as an educational tool. It must:

- be technically correct,

- be a valid role model for an object oriented program,

- be easy to understand (readable),

- promote "object oriented thinking", and

- emphasize programming as a problem solving process.

Inspired by the checklist-based evaluation by the Benchmarks for Science literacy-project (AAAS, 1989), three categories of qualities in examples have been defined: technical quality, object oriented quality and didactical/educational quality.

Technical quality: A good example must be technically correct. Code must be written in a consistent and exemplary fashion, i.e., appropriate naming, indenting, commenting and so on. This is not primarily connected to the object oriented paradigm, but programming in general.

Object oriented quality: A good example must be a valid role model for an object oriented program. Object oriented concepts and principles should be emphasized and reinforced. It is therefore important to show objects with mutable state, meaningful behavior, and communication with other objects.

Didactical quality: From a didactical point of view, a good example must be easy to understand, promote object oriented thinking and emphasize programming as a problem solving process.

A set of quality factors, denoted QF's, corresponding to desirable example properties for each of the categories above have been defined. The QF's, should adhere the following properties:

- Each QF is based on accepted principles, guidelines, and rules.

- Each QF is easy to understand.

- Each QF is easy to evaluate on a Likert-type scale.

- There are as few as possible QF's.

- There is as little redundancy as possible.

- All QF's are at a comparable level of granularity/importance, i.e., they contribute equally well to the overall "quality" of an example.

- The set of QF's covers all relevant aspects of "quality".

- The set of QF's must be applicable to any example, regardless of pedagogical approach and order of presentation.

Based on our experiences made during the work with the tool, it is reasonable to assume that technical quality factors generally are upheld, so our focus here is on object oriented and didactical QF's.

Object oriented quality (O1-O2)

O1: *Modeling.* The example emphasizes object oriented modeling. This means to emphasize the notion of object oriented programs as collections of communicating objects.

O2: *Style.* The code adheres to accepted object oriented design principles. E.g., proper encapsulation/information hiding, Law of Demeter (no in-appropriate intimacy), no sub classing for parametrization, etc.

The difference between O1 and O2 is that O1 focuses on design while O2 is concerned with the implementation. O1 could be highly rated if the attributes are well chosen, while O2 would rate poorly if the attributes were implemented as `public`. O1 would rate poorly and O2 highly if attributes were badly chosen, but still implemented as `private`.

Didactic quality (D1-D6)

D1: *Sense of purpose.* Students can relate to the example's domain and computer programming seems a relevant approach to solve the problem.

D2: *Process.* An appropriate programming process is followed/described, i.e., the problem is stated and analyzed, a solution is designed, implemented and tested/debugged.

D3: *Breadth.* The example is focused on a small coherent set of new issues/topics. It is not overloaded with new material or details introduced "on-the-fly".

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.

D5: *Visuals.* The explanation is supported by relevant and meaningful visuals.

D6: *Prevent misconceptions.* The example illustrates (reinforces) fundamental object oriented concepts/issues. Precautions are taken to prevent students from drawing inappropriate conclusions. The quality factors of particular interest here are: O1, O2, and D1.

These are important components and serve well in *evaluating* an example. One of the purposes with the development of this tool was to aid in developing a sense of good and bad, by doing the evaluation on small-scale examples used in textbooks, exercises, assessments and lectures. However, the object oriented qualities (O1 and O2) are not detailed enough to be particularly helpful when it comes to the *design* of examples.

2.8 CRC-cards and Role-plays

Introducing object orientation means introducing object oriented analysis and design. The approach for analysis to some extent determines the design. A popular method for analysis and design is text analysis in combination with the use of CRC-cards and role-play, described in (Bellin and Simone, 1997). CRC-cards are used to characterise object with respect to Class, Responsibility and Collaboration. The analysis is done by searching for nouns in a given problem description and using them as candidate objects, filtering and organising them into candidate classes. The verbs in the problem description can be used both for identifying responsibilities/behaviour and for identifying possible use cases or scenarios. In Figure 2.5 four classes are designed for a library application.

The approach is useful because it activates people and makes it everybody's responsibility to take part in the analysis and design of the system. It is also worth noting that the approach requires no knowledge of programming. An interesting application is the design of a Decanter centrifuges from Alfa Laval Separation A/S described in (Hvam et al., 2003). However, there are some problems using the original idea presented by (Beck and Cunningham, 1989). There is i.e. a great risk of confusing class and object, while the CRC-card at one point represents the generic description of all objects and at another point represents a single object (an instance). This problem can be addressed by introducing a notation called Role-Play diagrams (RPD's) with post-it notes representing the instances of a class and then using the CRC-card to function as a manuscript used when a particular object is active in the role-play (Börstler, 2004), Figure 2.6 is an example of a RDP for a scenario from a Library design.

Role-Play diagrams serves as a simple non-formal way of documenting the role-playing when working through the use cases or scenarios. The use of well known problem domains is promoted by studies reported in the literature, e.g., Biddle

Class: *Book*			**Class:** *Librarian*	
Responsibilities	**Collaborators**		**Responsibilities**	**Collaborators**
knows whether on loan			*check in book*	*Book*
knows due date			*check out book*	*Book, Borrower*
knows its title			*search for book*	*Book*
knows its author(s)			*knows all books*	
knows its registration code			*search for borrower*	*Borrower*
knows if late	*Date*		*knows all borrowers*	
check out				

Class: *Borrower*			**Class:** *Date*	
Responsibilities	**Collaborators**		**Responsibilities**	**Collaborators**
knows its name			*knows current date*	
keeps track of borrowed items			*can compare two dates*	
keeps track of overdue fines			*can compute new dates*	

Figure 2.5: CRC cards for Library application
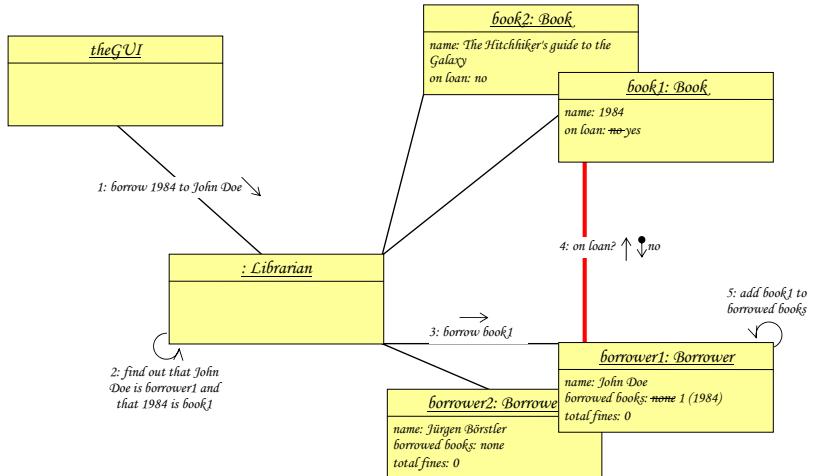
Figure 2.6: Role-Play Diagram for Library scenario

et al. (2002). Example-systems are often taken from every day life, such as libraries, banking, and economics. There is however a problem with every day life experiences, they tend to create confusion because of the conflict between the model and the entity being modelled. In real life a borrower at the library cannot be trusted to be responsible for keeping record of his/her unpaid fees for loans overdue. When modelling the borrower in the system, this responsibility might be perfectly reasonable.

2.9 Summary

In the area of introductory programming there are several aspects to be considered. Examples, order of presentation, instructional design and the more general aspect of knowledge acquisition and cognitive load to mention some.

Cognitive considerations are an underdeveloped aspect in Computer Science education research. Making its way into the area of introductory programming it should be beneficiaal for the development of instructional design. General theory on cognitive load and the use of worked examples must also be taken into consideration when designing the curricula.

In the literature there are suggestions for instructional design, made from different approaches, model-driven, responsibility-driven etc., but all in all, there is no general agreement upon *which* object oriented concepts are to be introduced, not *how* and not in what *order*. Though many of the instructional advices are fairly easy to agree with, Computer Science educators in general seem to have difficulties submitting to them. At least, the interpretation of them and the implementation of them seem to differ among educators, judging from the varying ways of exemplifying a certain concept or idea.

Chapter 3

Characteristics of Object Orientation

3.1 Introduction

Searching for characteristics of object orientation it is necessary to investigate how it all started. Early on, Nygaard and Dahl used ideas from ALGOL to name entities objects and to establish some characteristics (Nygaard, 1986). They stated that the basic concept should be *classes of objects*. The *subclass concept*, should be a part of the language, and direct, *qualified references* should be introduced.

Stroustrup (1995) makes the following (practical) definition of object orientation:

A language or technique is object oriented if and only if it directly supports: (1) *Abstraction* - providing some form of *classes* and *objects*. (2) *Inheritance* - providing the ability to build new abstractions out of existing ones. (3) *Run-time polymorphism* - providing some form of run-time binding.

He also states 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.

> *Though it has noble ancestors indeed, Smalltalk's contribution is a new design paradigm–which I called object oriented for attacking large problems of the professional programmer, and making small ones possible for the novice user.* (Kay, 1996)

Generally when talking about object orientation a number of concepts keep showing up. The difficulty is to be able to distinguish some sort of priority among them. When we are working with small-scale examples and programs, it is inevitable that we are forced to make sacrifices among the established principles of object orientation. The common understanding of "established principles" seem indisputable, but still remains to be defined.

In this chapter some attempts to establish basic characteristics, in terms of concepts, of object orientation are described.

3.2 Frequent Concepts

Litterature reviews

In an early literature survey by Henderson-Sellers and Edwards (1994) it is stated:

> *[. . .] on the surface there is no precise agreement about what features are vital for "object orientation." However, when one groups the terms frequently used, some consensus begins to emerge.* (Henderson-Sellers and Edwards, 1994)

The survey examines 20 books and journal articles by recognised pioneers of object orientation, such as Bertrand Meyer, Grady Booch, Bjarne Stroustrup etc. Terms found in these texts are collected into three groups and shown as corners in a triangle, see Figure 3.1. The three groups are:

The modularisation process that consists of *information hiding*, *encapsulation* and *objects*. This is related to the idea of expressing everything about an identifiable entity in one, and only one, place.

The grouping of collections to rationalise the world, which contains *classification*, *classes* and *abstraction*.

The notion of reusing code, which contains *inheritance*, *polymorphism* and *dynamic binding*.
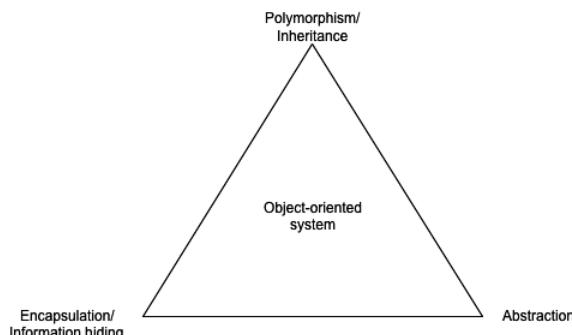
Figure 3.1: The object oriented triangle (Henderson-Sellers and Edwards, 1994)

A more recent study of frequent concepts can be found in (Armstrong, 2006). This study is based on 239 sources (journal, trade magazines, books and conference proceedings), spanning over the years 1966-2005. Searching for the key phrase *object oriented development*, Armstrong found 88 sources that discussed a specific set of concepts to characterise the object oriented development approach. All in all there were 39 concepts mentioned, and counting the frequencies resulted in the list seen in Table D.1.

The concepts were collected to support "practitioners in the midst of transitioning to object oriented development" and "researchers studying the transition to object oriented development". This is not the intended audience when discussing introduction to problem solving and programming using the object oriented

paradigm. Armstrong is not making any distinction of what is most critical, merely counting occurrences and picking the concepts most frequently used (in more than 50% of the papers). They were *Inheritance*, *Objects*, *Class*, *Encapsulation*, *Method*, *Message Passing*, *Polymorphism* and *Abstraction*.

One interesting point made is that many of the concepts with lower frequencies are either similar to or can be included in the more frequent concepts. This can be seen as a sign of the immaturity of the paradigm, and vocabulary and concepts remains to be agreed upon and defined.

These concepts were organised in a taxonomy, see Table 3.1. The most frequent terms have been categorized in two dimensions, one for structure and one for behaviour.

Table 3.1: Two construct object oriented taxonomy

Construct	Concept	Definition
Structure	**Abstraction**	Creating classes to simplify aspects of reality using distinctions inherent to the problem.
	Class	A description of the organization and actions shared by one or more similar objects.
	Encapsulation	Designing classes and objects to restrict access to the data and behaviour by defining a limited set of messages that an object can receive.
	Inheritance	The data and behaviour of one class is included in or used as the basis for another class.
	Object	An individual, identifiable item, either real or abstract, which contains data about itself and the description of its manipulations of the data.
Behaviour	**Message passing**	An object sends data to another object or asks another object to invoke a method.
	Method	Ways to access, set, or manipulate an object's information.
	Polymorphism	Different classes may respond to the same message and each implement it appropriately.

ACM's CC2001

The computing community has tried to formulate the basic requirements for a computer science degree, and in this work published curricula recommendations (ACM, 2008b).

ACM's Computing Currocula 2001 (ACM, 2001), with the update (ACM, 2008a) lists the following topics in the *Programming Fundamentals/ObjectOriented* category

- Object oriented design
- Encapsulation and information-hiding
- Separation of behavior and implementation
- Classes and subclasses
- Inheritance (overriding, dynamic dispatch)
- Polymorphism (subtype polymorphism vs. inheritance)

With the following *Learning objectives*:

27

1. Justify the philosophy of object oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism.

2. Design, implement, test, and debug simple programs in an object oriented programming language.

3. Describe how the class mechanism supports encapsulation and information hiding.

4. Design, implement, and test the implementation of "is-a" relationships among objects using a class hierarchy and inheritance.

5. Compare and contrast the notions of overloading and overriding methods in an object oriented language.

The concepts mentioned explicitly are *encapsulation*, *abstraction*, *inheritance*, and *polymorphism*.

The Java task force

In 2004 the ACM Java Task Force (JTF) (ACM, 2008c) was convened with the following charter:

> To review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a stable collection of pedagogical resources that will make it easier to teach Java to first-year computing students without having those students overwhelmed by its complexity.

Some problems addressed, are based on a survey of the literature of computer science education made by JTF in an attempt to identify the problems that have generated the greatest level of concern. The pedagogical problems were grouped into three categories:

High-level issues that are in some sense beyond the details of the language itself,

Language issues that arise from the design of Java itself, and

API issues associated with the application programmer interfaces provided as part of Sun's standard Java releases.

Most of the issues are cognitive rather than conceptual. The *Language issues* have some important educational implications. Being able to separate the interface (or protocol) of a class from its implementation (labelled L3) is crucial in object orientation and must be carefully addressed.

The JTF contribution is packages to simplify graphics, IO, GUI, and some more. Assignments and classroom demos have been developed. The use of these resources seem to be beneficial, but maybe more so to non-majors, see e.g., Mertz et al. (2008).

Anchor concepts

Based on ideas in cognitive research, the notion of anchor concepts is developed by Mead et al. (2006).

> *An anchor concept is a concept that is either foundational, i.e., it is a critical, basic concept in the knowledge domain, but not derivable in that domain or is both integrative i.e., it ties together concepts from the knowledge domain in ways that were previously unknown and transformative i.e., it involves a restructuring of schema, possibly integrating new information, resulting in the ability to apply what is known either differently and/or more broadly.*

By identifying anchor concepts within a curricula/discipline an anchor map is constructed to aid in the conceptual ordering and pedagogical approach to teach the desired skills, in this case programming. Using this approach on object oriented programming, the graph in Figure 3.2 is presented.
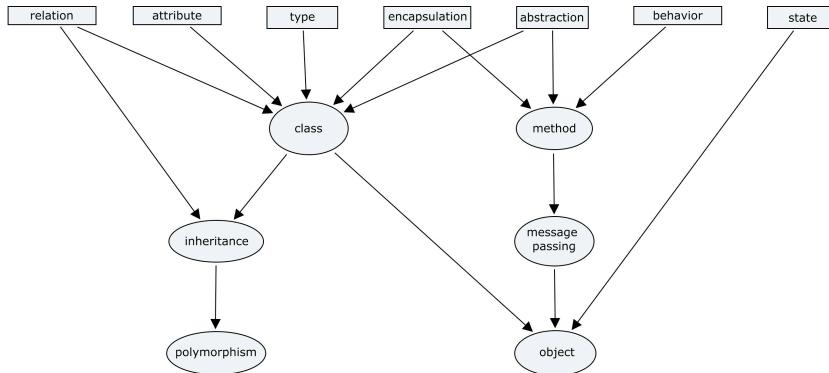
Figure 3.2: Anchor concept graph for object orientation by Mead et al. (2006)

The process of determining the anchor concepts should come from the underlying principles on which the target language is based. Fundamental concepts chosen are: *relation*, *attribute*, *type*, *encapsulation*, *abstraction*, *behaviour* and *state*. Anchor concepts for object-oreinted programming are: *Class*, *Method*, *Inheritance*, *Message passing*, *Polymorphism* and *Object*.

Truc's

Another interesting approach on cognitive elements in education is given by Meyer (2006). Introducing Trucs (Testable, reusable, units of cognition) Meyer states:

> *A Truc embodies a collection of concepts, operational skills, and assessment criteria. [...] The properties listed for Trucs recall some of the characteristics of design patterns. [...] Like patterns, Trucs need not claim originality; the primary effort is to catalog modes of thought that have proved their usefulness. Trucs could indeed be characterized as "education patterns".* (Meyer, 2006)

A number of specific properties are specified along with an accompanying rationale for teaching. Trucs depend on other Trucs, and this is shown by a dependency

graph. This graph must be acyclic, which may not be trivial, and it must not include transitive sub graphs: If C is dependent on B, and B is dependent on A then C must not depend on A. As an example, Meyer shows a graph for some basic object oriented concepts used in introductory programming textbooks, se Figure 3.3.
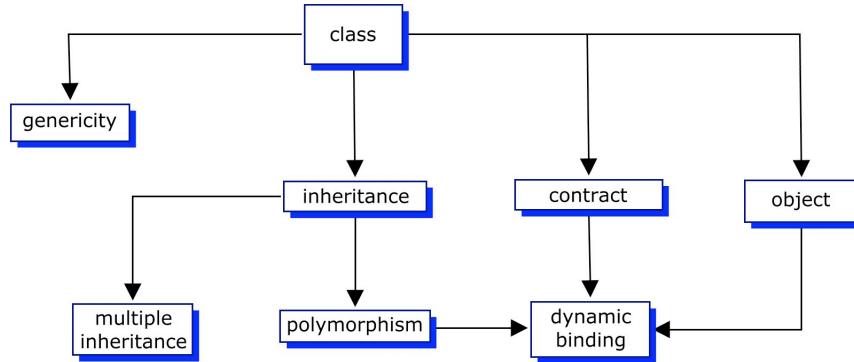
Figure 3.3: TRUC cluster with dependencies for object oriented programming (Meyer, 2006)

The main purpose of defining Trucs and their associated dependencies is to support education providers in the scaffolding for a course, a textbook, an exam, the design of a curriculum or other pedagogical use. They are intended to provide help in defining, understanding, teaching and assessing knowledge.

Threshold concepts

Interesting work on the notion of threshold concepts in computer science has been initiated by a multinational team. Threshold concepts are concepts that are

- *transformative*: they change the way a student looks at things in the discipline.

- *integrative*: they tie together concepts in ways that the were previously unknown to the student.

- *irreversible*: they are difficult for the student to understand.

- potentially *troublesome*: they conceptually difficult, alien, and/or counterintuitive.

- often *boundary markers*: they indicate the limits of a conceptual area or the discipline itself.

Eckerdal et al. (2006) suggests two candidate Threshold concepts for computer science, abstraction and object orientation. In a subsequent study (Boustedt et al., 2007) another concept is added: pointers. Identifying the Threshold Concepts seem to be non-trivial and much work still remains before there can be any practical implications of this approach.

Object oriented programming modelling and design

According to Rumbaugh et al. (1991) the exact characteristics of an object oriented approach are disputed, but generally include four aspects: *identity*, *classification*, *polymorphism* and *inheritance*.

3.3 Abstractions

Searching for the most critical components of object orientation, the word abstraction keeps showing up frequently (Armstrong, 2006). Some would claim that it is the key concept (Devlin, 2003; Kramer, 2007; Meyer, 2001; Parnas, 2007).

Despite that the CS community holds abstraction very highly, it is not explicitly a part of the educational curricula. Kramer (2007) bases his discussion of abstraction on Webster's definition of abstraction, which emphasise the process of removing details to be able to focus on others, and the process of abstracting common properties of instances to accomplish a generalisation. Kramer states that mathematics implicitly provides this skill. If not taught, to what extent can a person be expected to be skilled in abstraction. Starting with Jean Piaget (1896–1980) and his four stages for development: the sensorimotor stage, the pre-operational stage, the concrete operational stage, and the formal operational stage, Kramer notes that studies have shown that as much as 65-70% of adolescents have not reached Piaget's formal operations stage. This is a crucial stage in which individuals develop increased ability to reason hypothetically independently of concrete situations, and to describe ones own reasoning processes (Piaget, 1962). The use of symbols representing abstract concepts is made available. So Kramer poses the question "Is abstraction teachable?", In his opinion, the science to answer this question is not yet mature enough. There is too little knowledge about abstraction abilities including the problem that it is not obvious how to measure abstraction abilities. There is still much work to be done in this area. There is a need for tests to be able to monitor students progress in abstraction abilities, and to promote teaching techniques.

If abstraction is crucial to modelling entities in the problem domain, how do we learn how to develop abstractions?

In a reply to Kramer's article Parnas (2007) contrasts this to Dijkstra's definition:

> *An abstraction is one thing that represents several real things equally well.* (Dijkstra)

According to Parnas this definition is more useful than a definition that emphasise extraction. Dijkstra's definition states what must remain. According to Parnas, Dijkstra's' work shows that two distinct skills are related to abstractions: being able to work with given abstraction; and being able to develop an abstraction.

The use of abstraction in problem solving and programming emphasises the need for mathematical training. Mathematics helps in developing a formal and abstract way of reasoning and designing solutions for given problems, which is crucial for computer scientists. This is motivated and strongly advocated by a number of computer scientists in the CACM special issue *Why universities require computer science students to take math* in September 2003 (Devlin, 2003).

> *Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner. As it happens, that is something we humans have been doing successfully for more than three thousand years. We call it mathematics.* (Devlin, 2003)

Computer science originally being a part of mathematical computations has developed its vocabulary and formal notations largely based in the mathematical tradition. Vital to Computer science it is important to find ways to work with; and words to express abstractions. In (Rumbaugh et al., 1991)the following statements about abstractions are made:

> *Abstraction is the selective examination of certain aspects of a problem, [. . .]*
> *Abstraction must always be for a purpose, because the purpose decides what is and is not important. [. . .]*
> *All abstractions are incomplete and inaccurate. [. . .]*
> *A good model captures the crucial aspects of a problem and omits the others.* (Rumbaugh et al., 1991)

These are important insights: to be aware of the fact that abstractions are situated. E.g. the abstraction "car" can be modelled in various ways depending on the application. Possible contexts are, a system for taxes, a system for insurances, a system for a car-manufacturer, a system for advertising purposes, a game with moving vehicles, and so on. It becomes problematic if an abstraction should be designed without a context. What services and behaviour are reasonable if the purpose is unknown?

An interesting result for the connection between abstraction ability and mathematics can be found in (Bennedsen and Caspersen, 2006). The authors reports on a study where the hypothesis *General abstraction ability has a positive impact on programming ability* is tested. The study shows no correlation between level of cognitive development and mathematical ability. This is somewhat contradictory to the general assumption that mathematics supports abstraction abilities, see for example (Devlin, 2003) and (Kramer, 2007). Furthermore, the study shows no support for a correlation between abstraction ability and programming ability. The authors concludes by questioning their own indicators for programming ability; and the appropriateness of choice of test for abstraction ability (a commonly used pendulum problem).

During the process of object oriented analysis and design the developer has two primary tasks according to Booch, (Booch, 1994):

- Identify the classes and objects that form the vocabulary in the problem domain.

- Invent the structures whereby sets of objects work together to provide the behaviours that satisfy the requirements of the problem.

These classes and objects are termed *key abstractions* of the problem and the structures are termed *mechanisms* of the implementation. Initially the focus must be on the outside view of these abstractions and mechanisms. Even though the

design of classes and objects is an iterative process it is important to be able to know if a given class or object is well designed. Booch suggests five metrics to aid in this determination:

Coupling: a notion of the dependence between classes

Cohesion: a measure of how strongly related or focused the responsibilities of a single class are.

Sufficiency: the class captures enough characteristics of the abstraction to be useful in terms of interaction.

Completeness: the interface of the class captures all of the meaningful characteristics of the abstraction.

Primitiveness: primitive operations are those that only can be efficiently implemented given access to the underlying representation of the abstraction; i.e. operations that cannot be implemented by combination of other operations in the interface. According to Booch it is favourable that classes are primitive.

3.4 Object Thinking and Metaphors

Abstraction is central to object orientation. The difficulty is to know *what* to abstract. Looking at some general characteristics of an abstraction is not much help when dealing with the actual problem. To a novice it is vital to have guidelines for deciding what to abstract when designing a solution to a programming problem.

> *The inability to think about programs in an implementation-independent way still afflicts large sections of the computing community.* Edsger W. Dijkstra (Dijkstra, 1999)

Object thinking by anthropomorphisation is promoted by, among others, West (2004). Projecting human characteristics on to objects helps us think about objects in a different way than the traditional data-centred way of thinking. According to West the essential thinking difference can be stated: *Think like an object.* He is contrasting this to what he considers to be the prevailing mental habit among developers: Think like a computer. The problem with "Think like a computer" is that it leads to thinking in terms of the means of the problem solution, guided by how computers execute their instructions. Developing this kind of data centric solutions, means working with relations and relationships, and the solution is based on the tools available in the solution domain. Thinking as an object should be based in the problem domain rather than the solution domain. Objects are a community of cooperating virtual persons. A virtual person is an entity taking on responsibilities, such as offering/performing certain services.

3.4.1 Prerequisites to object thinking

West defines the following prerequisites to enhance object thinking:

Everything is an object

Any decomposition will result in the identification of a small number of objects (only objects!), almost regardless of how complicated the problem domain might be.

Simulation of a problem domain drives object discovery and definition

Decomposition is accomplished by applying abstraction. Data and function are poor choices as decomposition tools; behaviour should be used instead. West discusses the use of the three human methods for organisation: differentiation, classification and composition. This is the Behaviour should be used to find the natural entities in the problem domain. Differentiation is how we decide that one thing is different from the other. Classification makes it possible to talk about a group of similar entities as a group. Composition is the recognition that complicated things can be a combination of several simpler things. Behaviour is the key to finding the "natural" joints in the problem domain. Finding behaviours requires that we leave the traditional software developer glasses and start looking at the problem with the eyes of the user. One of the sources for advocating simulation as object discovery is the idea of *user illusions* (Kay, 1990). Thinking in terms of magic rather than metaphors should benefit decomposition of the problem domain. Another is the arguments of a "design decision hiding" approach to decomposition *(Parnas, 1972)*.

Objects must be composable

Composing is closely related to decomposing; i.e. it must be possible to assemble and to take objects apart. Reusability and flexibility is heavily dependent on this. It is therefore vital to state the purpose and capabilities clearly. What can potential clients expect? Since it is important to describe the object from the perspective of the problem domain, the terminology used must be chosen within the domain. The capabilities of the object stay the same, independently of the context. When it comes to taxonomies, they should imply specialisation so that substitution is made possible. If the specialisation is made through extension, objects lower in the taxonomy can be substituted for ancestors.

Distributed cooperation and communication must replace hierarchical centralised control as organisational paradigm

Objects are autonomous and it can be difficult to conceive how autonomous objects can be coordinated without a centralised control. Giving up centralised control is one of the hardest lessons to be learned in object thinking. Objects manage themselves and do not necessarily know anything about other objects, although they might notify other objects of a change of state. The effect of this notification may lead other objects into action depending on their interest in this information. Everybody is minding their own business announcing certain events, but never caring about whether this knowledge is used or not.

3.4.2 Metaphors: The use of anthropomorphisation

West discusses metaphors as an import way of getting the right mindset for object thinking. One of the earliest, according to West, is the software integrated circuit (IC), coined by Brad Cox. The metaphor is applicable to object orientation because of the goal to construct reusable plug-in software components. The shortcoming of this metaphor is in object discovery and object specification. A far better metaphor, according to West, is the metaphor of a person. An object is like a person. In his opinion, and others, using this metaphor is a good idea for a number of reasons:

- It aids in discovering objects in the problem domain

- Support is given in design decisions: asking the object for a certain service indicates what kind of information the object need to be responsible for and even indicates the form of this information (asking for a persons ID may result in her/him showing a drivers license which might be represented by a dedicated object instead of primitive attributes)

- It aids in remembering principles of object orientation: persons are lazy, so if anything looks too hard, get more help (split and distribute the work onto more objects).

- The mindset is heavily influenced by the vocabulary, so asking an object to perform a certain service is more helpful than to decide what the program/the machine should do next.

Software objects should be lazy and specialise, they should simulate the services provided by real-world objects. There is however a limit to the extent of (de-)composition, sending parameters back and forth between collaborating objects might be counter productive. The metaphor should guide the decomposition and reflect the demands of the problem domain when assigning responsibilities to the software objects.

Close to this is the Object-as-agent metaphor. According to West the usefulness of this metaphor is severely limited by the fact that an agent is not completely autonomous. In his interpretation autonomy means independence and freedom of action, which object and agents share, but furthermore it means behavioural integrity which object have but agents do not. The behaviour of an object is intrinsic but an agent acts on behalf of a client as an extended part of the client. This means that the agent is partly dominated by the idiosyncrasies of the client and only partly acting in its own nature. Being a good agent is important. An object has no knowledge of any clients and is acting completely according to its own nature.

West also mentions two human-derived metaphors: Inheritance and Responsibility. They stem from the observation of human beings. They are important but suffer from not being uniquely defined between contexts. Trees commonly illustrate inheritance and the concept is furthermore blurred by the use of the terms parent and child. This might lead to confusion with genealogical charts with children inheriting from their parents. In a genealogical chart parents and children are never the same, but in the context of objects, inheritance is strictly a specialisation of behaviour. Responsibility means that a stated capability of providing a service must be consistently performed no matter what the circumstances.

Of course a philosophy has its critics and Dijkstra (Dijkstra, 1983) thinks that anthropomorphism is the worst of all metaphors and analogies. Dijkstra rejects the idea of disguising computers' greatest strength: the efficient embodiment of a formal system. However, this is more aiming at anthropomorphising computers than designing solutions.

3.4.3   Object vocabulary

The idea of problem solutions being based in the problem domain and the use of assisting metaphors is of course empowered by the choice of a suitable vocabulary. The vocabulary shapes our ability to think and helps us communicate our thoughts. In object thinking the vocabulary is partly chosen to differentiate object concepts from traditional software concepts. Sending a message and invoking a function might be the same when it comes to syntax and implementation, but sending a message involves a sender and a receiver, an interpretation of the message and a possible response. West differs between words in the object vocabulary in terms of importance. In Table 3.2 it is seen that the essential terms in the object vocabulary, according to West, are Object, Responsibility, Message and Protocol. The protocol is the part of the interface listing the messages the object is willing to respond to. West distinguishes these messages from the state changes clients could register for, in all called the interface of an object.

By the use of these four words the essentials of object thinking can be captured. They constitute the essence of objects needed to decompose a problem domain, to identify and distribute responsibilities to a collection of objects. They provide a way to specify the objects, not necessarily sufficient to implement them. The internals of the objects must be described with the supporting terms of the vocabulary.

Table 3.2: Categorised Object Vocabulary (West, 2004)

Essential	Extension	Implementation	Auxillary
Object	Collaboration/ collaborator	Method	Domain
Responsibility	Class	Variable	Business requirement
Message	Class hierarchy	Dynamic binding	Business process reengineering (BPR)
Protocol	Abstract/ concrete		Application
	Inheritance		
	Delegation		
	Polymorphism		
	Encapsulation		
	Component		
	Framework		
	Pattern		

According to West one of the greatest benefits of object thinking is that it helps emphasising the need to understand the problem domain before doing anything else.

- Use the metaphor of domain anthropology to extract domain understanding.

- Decompose the problem domain into behavioural objects, according to user expectations.

- User illusions should be maintained, unless an alternative story with a different set of domain entities, with different responsibilities, interacting in a different way, can be constructed.

- Decompose problems in terms of conversations among groups of objects.

- Model objects as simple as possible, with the collective understanding clearly communicated.

According to West the initial focus must be on identifying stories and some potential objects of the problem domain. It is imperative that the social relationships among potential objects are discovered. One of the big problems for the developer is the preconception about implementation details. Each object must be defined in terms of how it is perceived by those using it, and this should aid in the ambition "Model objects as simple as possible".

3.5 Summary

What are the characteristics of object orientation? Searching the literature does not give a conclusive answer to this question. However there a number of concepts often mentioned. *Abstraction*, *class*, *method*, *object*, *inheritance* and *polymorphism* are among the most common ones. Terminology and vocabulary have not been agreed upon within the community, but according to West (2004) the object vocabulary consists of *Object*, *Responsibility*, *Message* and *Protocol*. All other terms are used to elaborate on aspects of terms or to extend the essential terms.

Apart from the concepts characterising object orientation, the mindset must be taken into account. The way we talk about objects and formulate systems should be done in terms of the problem instead of the solution. Defining the objects from the perspective of the problem domain and the clients are key ideas in object thinking. This is important for teaching and learning abstraction and object orientation.

Chapter 4

Object Oriented Principles

4.1 Introduction

What are the most critical object oriented principles to uphold? Searching the literature; most discussion relates to the principles collected and/or formulated by Martin (2003). These principles can be grouped into three categories, see Table 4.1.

Table 4.1: Object oriented design principles

Class Design	SRP – The Single Responsibility Principle. OCP – The Open Closed Principle. LSP – The Liskov Substitution Principle. DIP – The Dependency Inversion Principle. ISP – The Interface Segregation Principle.
Package cohesion	REP – The Reuse Release Equivalency Principle. CCP – The Common Closure Principle CRP – The Common Reuse Principle.
Package coupling	ADP – The Acyclic Dependencies Principle. SDP – The Stable Dependencies Principle. SAP – The Stable Abstractions Principle.

The aim of these principles is to uphold and aid good object oriented design. This must, of course, also be the aim of small-scale problem solving and programming. When restricting the discussion to the context of small-scale problems it seems reasonable to focus primarily on the first category of principles: Class Design. The concept of packages is not normally covered when introducing object oriented programming to novices.

A short description of the Class Design principles:

SRP – The Single Responsibility Principle Each responsibility should be 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.

Apart from these principles The law of Demeter can be added.

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

The following sections describe and exemplify the Class Design principles introduced above.

4.2 SRP – The Single Responsibility Principle

The more devoted a class is to a well specified and restricted abstraction, the more stable it will be, since the reasons for making changes to it can, by definition, be very few. At the class level this principle is similar to the definition of cohesion.

In this context responsibility is defined as "a reason for change". An example of the difficulty of detecting responsibilities is shown in Listing 4.1, from (Martin, 2003).

```java
interface Modem
{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char receive();
}
```

Listing 4.1: `Modem.java` - SRP Violation(Martin, 2003)

Even though the four methods listed are reasonable for a modem there are in fact two separate responsibilities involved. One is connection management (`dial` and `hangup`), and the other is data communication (`send` and `receive`). According to Martin these two should be separated since they have little in common and will change for different reasons.

Another example from Martin is shown i Figure 4.1.

In this case the `Rectangle`-class is used by two completely different applications. One that deals with geometrical figures and one that draws shapes on some graphical surface. The design will benefit from separating the two responsibilities
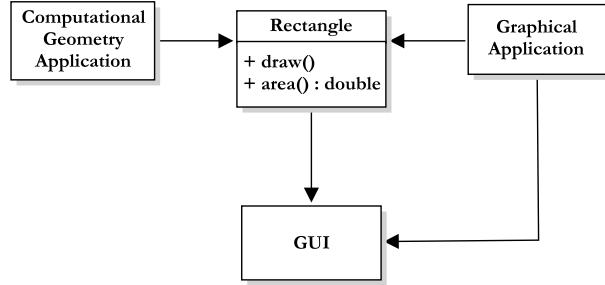
Figure 4.1: SRP Violation example
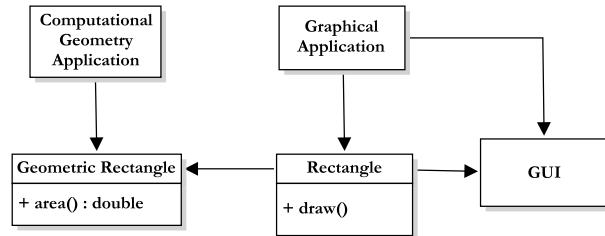
Figure 4.2: SRP Separation of Responsibilities

into separate classes. In Figure 4.2the computational responsibility is moved to a class specialising on the rectangle as geometrical figure.

Any changes to the graphical presentation of the rectangular forms will not affect the `ComputationalGeometryApplication`. Recognising these differences is non-trivial.

> *The SRP is one of the simplest of the principles, and one of the hardest to get right. [...] Finding and separating those responsibilities from one another is much of what software design is really about.* (Martin, 2003)

4.3   OCP – The Open Closed Principle

A class should be closed for changes and at the same time open for extension. This is a rather complicated principle, since it demands that two contradictory purposes are met. How can a module change without having to change the source code? To achieve this one must rely on abstractions.

Keeping the attributes of a class private is one way of making the application of OCP easier. The reason for this is that when an attribute changes, every function that depends upon those variables must be changed. Thus, no function that depends upon a variable can be closed with respect to that variable.

Any form of run time type identification is dangerous and should not be used. The dependency between classes, or coupling, is problematic and violates the OCP.

Example by Gupta (2008): Loan requests are handled in a banking application. In Figure 4.3 the class `LoanRequestHandler` is strongly coupled

to `PersonalLoanValidator`. If the bank decides to provide more kinds of loans, e.g. a business loan, the handling will be different and demand a change in both classes, even being forced to use condition to handle different kinds of loan. These classes are not closed for modification and violates the OCP.
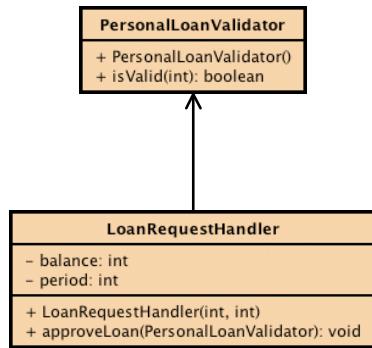
Figure 4.3: OCP Violation

How can the strong coupling be avoided? Validating different kinds of loan requests is a reason for change and should be handled more generally. The solution to this is abstraction. Introduce a general `Validator` and have `LoanRequestHandler` treat all request uniformly. Never mind who asks, always treat the request the same way and have the specific request supply the appropriate implementation of the behaviour, see Figure 4.4.
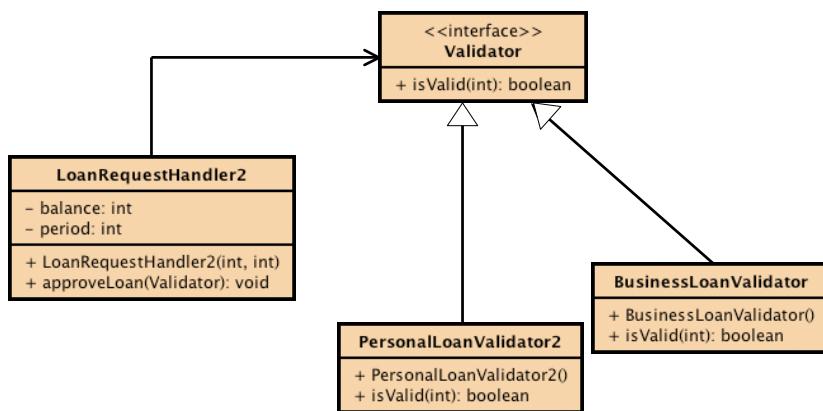
Figure 4.4: Avoiding OCP Violation

Now the application is open to extension but closed for modification.

According to Martin, OCP is the one characteristic that yields the greatest benefits of object technology, i.e. flexibility, reusability, and maintainability. This does not mean that abstraction is to be used unconditionally, but applied with care to parts of solutions that is prone to change.

> *Resisting premature abstraction is as important as abstraction itself.*
> (Martin, 2003)

4.4   LSP – The Liskov Substitution Principle

The Liskov substitution principle can be paraphrased as:

> *Functions that use pointers or references to base classes must be able*
> *to use objects of derived classes without knowing it.* Martin (2003)

This is the consequence of polymorphism. To utilise polymorphism it is imperative that in class hierarchies, it should be possible to treat subclass-objects as if they were base class objects. Sub-classes must not change any behaviour of the base class that invalidates assumed characteristics of the base class. Looking at the classical Square-Rectangle example in Figure 4.5, it can be noted that `Square` does not need the two methods `setWidth` and `setHeight` inherited from `Rectangle`, since the width and height of a square are identical. This is an indication of poor design.
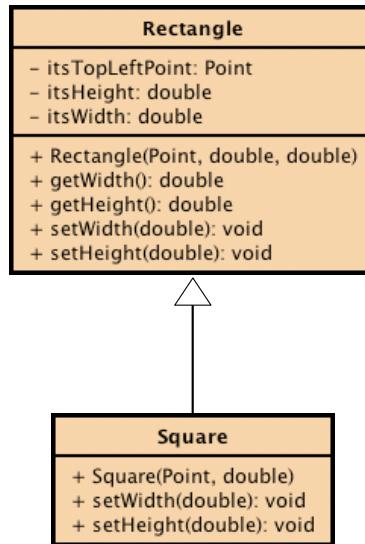
Figure 4.5: LSP Violation

One way to deal with this is to override the two methods, see Listing 4.2.

43

```
import java.awt.*;
public class Square extends Rectangle
{
  public Square(Point t1, double s)
  {
    super(t1, s, s);
  }
  public void setHeight(double h)
  {
    super.setHeight(h);
    super.setWidth(h);
  }
  public void setWidth(double w)
  {
    super.setHeight(w);
    super.setWidth(w);
  }
}
```

Listing 4.2: LSP Violation

This means that the square will be correct from a mathematical point of view and objects will behave as expected. Consider the method in Listing 4.3, using a reference to a `Rectangle`.

```
public void m(Rectangle r)
{
  r.setWidth(32);
}
```

Listing 4.3: LSP - Using the references polymorphic

This will work properly even when `r` is a reference to a `Square` object and will cause both width and height to be assigned the same value. This means that we seem to have two classes that work as intended.

The problem is that a derivation from a base class forces us to make changes to the inherited behaviour. A client using objects through the method `mm` in Listing 4.4 may assume that changing the height of the object referenced by `r` will not cause any change to the width.

```
public void mm(Rectangle r)
{
  r.setWidth(5);
  r.setHeight(4);
  System.out.println("w*h = 20 : "+ r.getWidth()*r.getHeight());
}
```

Listing 4.4: LSP Unexpected behaviour

This is a reasonable assumption, but will cause problems if a `Square` object is passed as parameter. Methods, like `mm`, that accepts references to `Rectangle` objects but does not operate properly on `Square` object are therefore violating the Liskov Substitution Principle. To avoid these kinds of mistake it is vital to remember that the relation "is-a" is all about behaviour, and when it comes to behaviour a `Square` is not a `Rectangle`. The OCP and LSP can be upheld by thinking in terms of pre- and post conditions as described by Meyer (1997):

> *A routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.*

A client should be able to use objects through the protocol of the base class. This means that derived classes must accept anything the base class would accept and their behaviour must not violate any of the base class' behaviour, so that clients may not be confused.

Another important lesson learned from this principle is that "Validity is not intrinsic".

> *A model, viewed in isolation, can not be meaningfully validated. The validity of a model can only be expressed in terms of its clients.*(Martin, 2003)

This makes it important to provide context when demonstrating concepts and design ideas in a small-scale context.

4.5 DIP – The Dependency Inversion Principle

This principle is a structural tool to achieve the Open-Closed principle and the Liskov Substitution Principle. Martin (2003) formulates it like this:

> a. *High level modules should not depend upon low level modules. Both should depend upon abstractions.*
> b. *Abstractions should not depend upon details. Details should depend upon abstractions.*

Depending upon abstractions is beneficiary because abstractions are less likely to change than concretisations. Modules that contain detailed implementations should not be depended on, rather they themselves depend on abstractions. It is vital to separate high-level policies from low-level implementations.

Martin (2003)gives the following practical consequences of this :

- No variable should hold a pointer or reference to a concrete class.

- No class should derive from a concrete class.

- No method should override an implemented method of any of its base classes.

The creation of instances is the most common dependence on a concrete class and is obviously hard to avoid, especially in a small-scale context. In the general case this can be handled by using a design pattern (AbstractFactory), see Chapter 6.

Example from Martin (2003) : a `Button` object senses the external environ-
ment in some way. It can be polled whether the button has been activated
or deactivated. A `Lamp` object affects an external unit, and on receiving
a `turnOn` message it illuminates some light, and on a turnOff message it
extinguishes the light.

The design in Figure 4.6 shows a direct dependency between `Button` and `Lamp`.
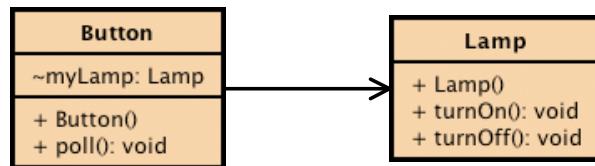
Figure 4.6: DIP Violation (Martin, 2003)

This design makes it difficult to reuse the `Button` class to control some other
kind of object, e.g. a `Motor` object and thereby violates the DIP. The high-level
policies has not been separated from low-level implementations of the application.
Without separating them, the abstraction will automatically depend on the details.

What would the high-level policy of this application be? Martin calls it the
metaphor of the system. In this small example the underlying abstraction is to
detect an on/off gesture from a client and relay that gesture to the target object.
Target object and mechanism to detect the gesture is irrelevant, these are details
that do not affect the abstraction. The design in Figure 4.6 can be improved by
inverting the dependency on the `Lamp` object, see Figure 4.7.
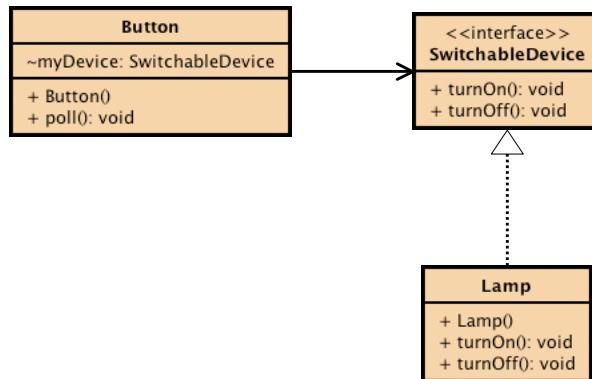
Figure 4.7: DIP implemented

Now the `Button` is dependent on a `SwitchableDevice`. In this design a
`Button` object can be used to to turn any device on or off, and `Lamp` is depending
on an abstraction instead of being depended upon.

> *Indeed, it is this inversion of dependencies that is the hallmark of
> good object oriented design. [...] If its [the programs] dependencies are
> inverted, it has an OO design.* (Martin, 2003)

4.6 ISP – The Interface Segregation Principle

Many client specific interfaces/protocols are better than one general-purpose interface/protocol. A class used by many clients could be better utilised if several interfaces targeted towards different clients is created. This is much better than imposing the entire class on every client.

> *Clients should not be forced to depend on methods that they do not use.*(Martin, 2003)

Example from Martin (2003): In a security system there are `Door` objects that can be locked and unlocked, and they know whether they are locked or not. In the spirit of the DIP, the `Door` class is made abstract thereby allowing clients to use objects with this behaviour without having to depend on a particular implementation . A `TimedDoor` is an implementation of `Door` that sounds an alarm when when the door has been left open too long. To manage this, the `TimedDoor` object collaborates with a `Timer` object. The `Timer` object keeps a reference to a `TimerClient` object whose `timeOut` method is to be called when time expires.

In Figure 4.8 an UML class diagram of a straightforward solution is shown.
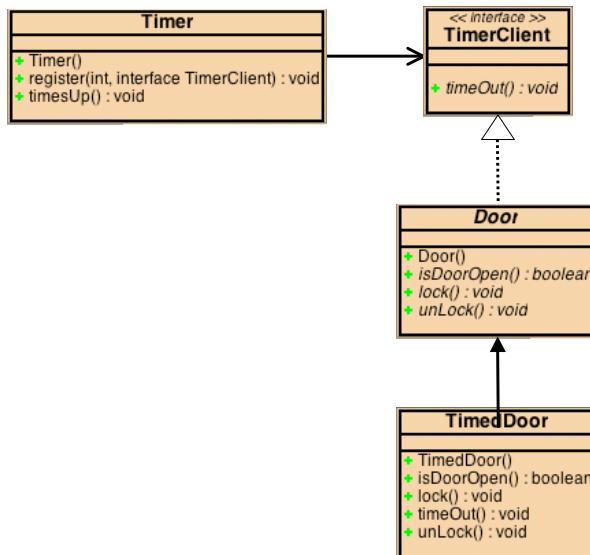
Figure 4.8: ISP Violation

The `Door` class is now dependent on the `TimerClient` class. This is an example of interface pollution, since the protocol/interface of `Door` is polluted by a method (`timeOut`) that it does not require, since not all doors need timing. Timing-free subclasses of `Door` need to make fake implementations of the method, which might lead to violation of LSP. The method is there to support a particular kind of door, but if this line of design is continued it can be necessary to add more methods to `TimerClient` because of the extension of the hierarchy of doors.

`Door` and `TimerClient` are protocols/interfaces used by different clients. The idea of ISP is to keep the protocols/interfaces separate too.
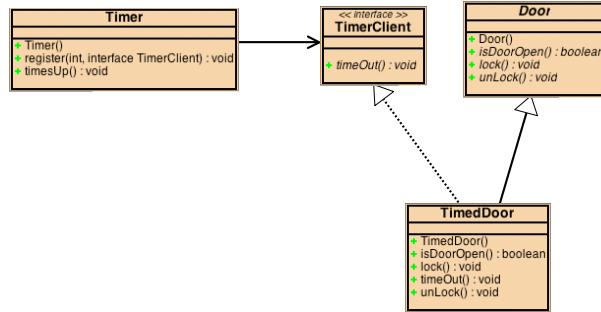
Figure 4.9: ISP Solution

With the design shown in Figure 4.9 clients use both `TimerClient` and `Door`, but are not dependent on `TimedDoor`. This means that they are able to use the same object through different protocols/interfaces. With Java allowing only single inheritance it is necessary to use interfaces, either for one of the classes or both. In this case it might be argued that being a `Door` is a structural relationship, while being a `TimerClient` is a simple behaviour.

4.7   LoD – The Law of Demeter

Do not talk to strangers - only talk to your immediate friends.

The idea of this "law" is to restrict the message-sending in methods. The law prevents a method from retrieving a part of another object. Intermediate methods must be used to limit the coupling with respect to "uses" relations.

Object form of Law of Demeter (DemeterW3; Martin, 2003)

A method m of a class C, can only call methods of:

- C
- An object passed as an argument to m.
- An object created by m.
- An object held in an instance variable of C

Methods should not invoke methods on any object returned by any of the above methods (Martin 2008). One consequence of this is that methods should not return a reference to an object that is participating in the internal implementation of the object. This is to prevent clients to operate on objects that are parts of other objects (Horstmann, 2004).

Example by Bock (2008): A `Paperboy` has to get a payment from a `Customer`. The `Customer` has a `Wallet` that contains the cash. A simple design is shown in Figure 4.10.
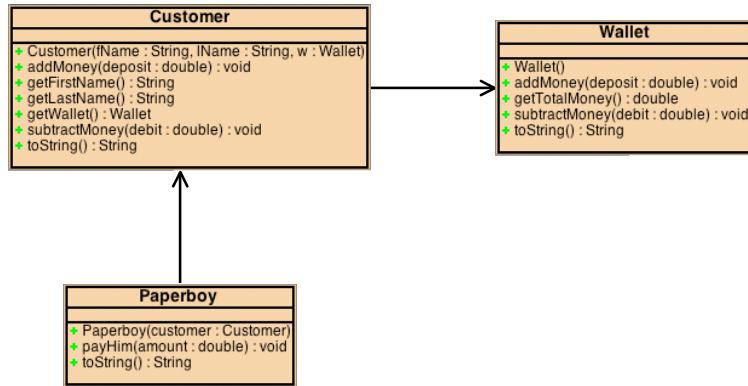
Figure 4.10: LoD Violation

To get paid the `Paperboy` asks for the `Customer`'s `Wallet` and withdraws the amount of money needed, see Listing 4.5.

```java
public void payHim(double amount)
{
  Wallet theWallet = myCustomer.getWallet();
  if (theWallet.getTotalMoney() > amount)
  {
    theWallet.subtractMoney(amount);
    payment = payment + amount;
  }
  else
  {
    System.out.println("---Not sufficient amount in wallet---");
  }
}
```

Listing 4.5: LoD Violation

In this example the `Paperboy` is being exposed to more information about the `Customer` than necessary. There is no need for the `Paperboy` to have access to the `Wallet`. The `Wallet` object is not any of the specified object listed in the law of Demeter. It is crucial to leave the responsibility to the right object, in this case the `Customer` does not need to show the internal representation of the cash to any client.

```
public void payHim(double amount)
{
  double paidAmount = myCustomer.getPayment(amount);
  if (paidAmount == amount)
  {
    payment = payment + amount; // say thank you and give
        customer a receipt
  }
  else
  {
    System.out.println("---I will come back later!---");
  }
}
```

Listing 4.6: LoD Solution

In general it is wise to think in terms of clients asking for services rather than objects or values.

4.8   Summary

The principles discussed in this chapter are, in our opinion, the most general and condensed description of what characterises object orientation.

One problem with principles is that they tend to be too general, and the object oriented design principles described above have not been formulated for the benefit of educators struggling with small examples to be used for novices, neither for novices trying to learn and understand object orientation.

Within the small-scale context it can be difficult to fully subordinate to these principles. From an educational perspective we are tempted to save space in terms of lines of code to increase readability and to make the example more focused. Too often this leads to violations of more than one of the above mentioned principles. On the other hand, following the principles will likely make the example grow, the number of classes increase and all this overhead is not easily justified to a novice. Nevertheless, we have to keep these principles in mind when designing small-scale examples.

The Single Responsibility Principle (SIP) is perhaps the most basic of the six principles presented. Many heuristics, rules and guidelines are aspects or consequences of this principle. Being true to the Dependency Inversion Principle (DIP) should imply using interfaces and abstract classes to a larger extent than common in most introductory text books. The Law of Demeter gives an indication of how to promote privacy and responsibilities of objects.

Chapter 5

Heuristics and Rules for Software Design

5.1 Introduction

In this chapter, we review examples of heuristics and rules for object oriented design, formulated by Computer Science researchers . This is done to see what object oriented characteristics different researchers have considered when formulating design advice.

The words heuristics and rules are used interchangeably and there is no clear distinction between them. Looking at the dictionary description of heuristic shows the difficulty in making such a distinction.

> *Heuristic - a commonsense rule (or set of rules) intended to increase the probability of solving some problem.* (The Free Dictionary, 2008)

Often these commonsense rules build upon experience and hard earned insights.

The level of detail varies from very general (e.g., Riels heuristic 03.07 *Eliminate irrelevant classes from your design,* Appendix *A.*) to detailed code instructions (e.g., Bloch's Rule #31 *Avoid Float and Double if exact answers are required.* Table D.4), even within the same collection of advice. There is also a large variation in the number of advices given, the number ranges from 7 to 61.

5.2 Johnson and Foote's Heuristics

The need for object oriented design rules or guidelines was discovered early and the paper by Johnson and Foote (1988) is often cited as one of the first written collections of design rules. Johnson and Foote are primarily concerned with reusability and techniques that makse object oriented software more reusable.

Rules for Finding Standard Protocols

Rule 1	Recursion introduction: use the same method names in a chain of calls to perform an operation.
Rule 2	Eliminate case analysis: it is almost always a mistake to check the class of an object. The same is true for case analysis of variables.
Rule 3	Reduce the number of arguments: many arguments make messages hard to read.
Rule 4	Reduce the size of methods: makes subclassing easier.

Rules for Finding Abstract Classes

Rule 5	Class hierarchies should be deep and narrow.
Rule 6	The top of the class hierarchy should be abstract.
Rule 7	Minimize accesses to variables.
Rule 8	Subclasses should be specializations.

Rules for Finding Frameworks

Rule 9	Split large classes.
Rule 10	Factor implementation differences into subcomponents.
Rule 11	Separate methods that do not communicate.
Rule 12	Send messages to components instead of to self.
Rule 13	Reduce implicit parameter passing.

Design should be given enough attention to result in general, reusable components.

> *Object oriented techniques offer us an alternative to writing the same programs over and over again. We may instead take the time to craft, hone, and perfect general components, with the knowledge that our programming environment gives us the ability to re-exploit them. If designing such components is a time consuming experience, it is also one that is aesthetically satisfying. If my alternatives are to roll the same rock up the same hill every day, or leave a legacy of polished, tested general components as the result of my toil, I know what my choice will be.* (Johnson and Foote, 1988)

5.3 Riel's Heuristics

A number of design heuristics (61) has been collected and described in detail by Riel (1996). They can, in general, be classified as practical consequences of the design principles of Table 4.1.

The heuristics are categorised and presented by chapter in the book:

Ch-2: Classes and Objects: The Building Blocks of the Object oriented Paradigm (11 heuristics)

Ch-3: Topologies of Action-Oriented Vs. Object oriented Applications (10 h)

Ch-4: The Relationships Between Classes and Objects (14 h)

Ch-5: The Inheritance Relationship (19 h)

Ch-6: Multiple Inheritance (3 h)

Ch-7: The Association Relationship (1 h)

Ch-8: Class Specific Data and Behavior (1 h)

Ch-9: Physical Object oriented Design (2 h)

These heuristics are well formed and straightforward. In general the principles in Chapter 4 are recognisable. Taking a look at the basic heuristics concerning classes and objects (heuristics 2.01-2.11), the ideas of abstraction and encapsulation are well covered.

02.01 All data should be hidden within its class

02.02. Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

02.03. Minimize the number of messages in the protocol of a class (protocol of a class means the set of messages to which an instance of the class can respond)

02.04. Implement a minimal public interface that all classes understand

02.05. Do not put implementation details such as common-code private functions into the public interface of a class

02.06. Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.

02.07. Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.

02.08. A class should capture one and only one key abstraction

02.09. Keep related data and behaviour in one place.

02.10. Spin off non-related information into another class (that is, non-communicating behaviour)

02.11. Be sure the abstractions that you model are classes and not simply the roles objects play

See Appendix A for a complete list of Riel's heuristics.

5.4 Gibbon's Heuristics

In an attempt to aid novices in the development of design skills,Gibbon (1997) formulated a set of design heuristics targeted for automation. A tool (TOAD) was developed to assist students in evaluating their design during the design phase.

One of the objectives for this work was to identify and use design heuristics as a vocabulary for common object oriented design problems. The main focus of design quality in this tool is maintainability.

In this work design heuristics and design metrics are combined to identify, analyse, submit feedback and suggest solutions to common design problems. The evaluation is seen as a means to show the learner what and where certain heuristics are violated and at the same time educate the learner why certain approaches are more preferable than another. The suggested design heuristics, build upon the heuristics by Riel (1996), but is compounded to support the understanding of what solution to suggest (Gibbon and Higgins, 1996).

The heuristics have a two letter identification code. The first letter documenting the *type* of heuristic; either class (C) or relationship (R). The second letter indicates the *model* used by Gibbon, C for class model, U for using, A for aggregation, and I for inheritance.

CC1 Limit the number of methods per class

CC2 Limit the number of attributes per class

CC3 Limit the messages an object can receive

CC4 Minimise complex methods

CC5 Limit enabling mechanisms that breach encapsulation

CC6 Hide all implementation details

CU1 Limit the number of collaborating classes

CU2 Restrict the visibility of interface collaborators

RU1 Identify and stabilise common place interface collaborators

CA1 The aggregate should limit the number of aggregated

CA2 Restrict access to aggregated by clients

RA1 Aggregation hierarchies should not be too deep

RA2 The leaf nodes in an aggregation hierarchy should be small, reusable and simple

RA3 Stability should descend the hierarchy from rich aggregates to their building blocks

CI1 Limit the use of multiple inheritance

CI2 Prevent over-generalisation of the parent class

RI1 The inheritance hierarchy should not be too deep

RI2 The root of all inheritance hierarchies should be abstract

RI3 For deep hierarchies upper classes should be type definitions

RI4 Minimise breaks in the type/class hierarchy

RI5 Strive to make as many intermediate nodes as possible abstract

RI6 Stability should ascend the inheritance hierarchy

RI7 Inheritance is a specialisation hierarchy

Individual heuristics covers a number of design concepts for a design problem, but to support the design analysis Gibbon (1997) constructs a number of *concept categories*, see Table 5.1. These concept categories is used to give feedback on design principles.

> *Experience teaching OOD has shown that these concept categories are the once that inexperienced designers typically require feedback on. Grouping design heuristics into categories provides a uniform way of representing higher level design problems relating directly to essential paradigmatic concepts.* (Gibbon, 1997)

Table 5.1: Concept categories used in TOAD(Gibbon, 1997)

Concept category	what the category reports on
Inheritance	the use of abstract classes the shape of the hierarchy inheriting from concrete classes issues of type vs. class
Encapsulation	use of non-private data mechanisms used to breach encapsulation
Collaborators	the behavioural content of the class centralisation of the designs' behaviour possession of high fan-out
Bandwidth	if low or high if class cohesive abstract or concrete position in the inheritance hierarchy
Implementation	size contributions by user-defined types attribute well encapsulated

It has been impossible to trace the reasoning for the choice of heuristics used, since the catalogue itself is irretrievable due to a missing report (Gibbon).

5.5 The MeTHOOD Heuristics Catalogue

The MeTHOOD project is a formalisation and integration of design heuristics, measures and transformation rules (Grotehen, 2001). The focus of the project is limited to object bases, e.g. shared object oriented databases. This methodology extension supports designers in finding the road to better designs for these important components of many systems. MeTHOOD combines measures describing the objective, heuristics showing and transformation rules representing more concrete decision support. As a part of the project, a heuristics catalogue containing 24 heuristics is presented, see Table D.2.

5.6 Heuristics for Thinking Like an Object

West (2004) proposes eight heuristics for discovering and assigning object responsibilities. These heuristics are based on anthropomorphisation, see Section 3.4, in this case thinking like an object.

H1: Let objects assume responsibility for tasks that are wholly or completely delegated to other objects in cases in which the responsibility reflects a natural communication pattern in the domain.

H2: Delegate responsibilities to get a better distribution and increase reusability.

H3: Use anthropomorphisation and foreshadowing to determine whether an object should assume a given responsibility.

H4: Responsibilities should be distributed among the community of objects in a balanced manner.

H5: Always state responsibilities in an active voice describing a service to be performed.

H6: Avoid responsibilities that are characteristic specific, that focus on providing a potential user with the value of a simple characteristic of the object.

H7: Create proxies for objects outside the domain that are sources of information required by objects within the domain.

H8: Look for components, i.e. larger cluster of strongly coherent responsibilities.

5.7 Design Rules

Garzás and Piattini's design rules

An attempt to combine principles, design patterns, rules and heuristics has been made by Garzás and Piattini (2007b). The term rule has been chosen to incorporate principles, code smells (described in Section 6.6), best practices and so on. These rules are applicable to object oriented design at a class design level. The rules are further explored and described in (Garzás and Piattini, 2007a) A short description of the rules can be found in Table D.3.

Even though Garzás and Piattini claim to present a catalogue of object oriented design knowledge, the formulation of the rules are towards existing code. They aid in defining what to look for and how to deal with it. How can this condensed mass of experience be formulated to enlighten the novice before the mistakes are made?

Bloch's effective rules for Java

Another set of rules are defined in (Bloch, 2001) with, in the first edition, 57 items that according to Bloch, describe practices generally held to be beneficial by recognised programmers. Bloch's rules in Table D.4 are given in the original thematical order. They are

Wick's design rules for concepts

There are several suggestions for rules on more limited areas of object orientation. One example is the design rules for encapsulation and abstraction used by Wick

et al. (2004). Only one of the seven rules is said to applicable for novices:

> *Whenever possible and practical, avoid writing getters and setters. First ask, "What is the client attempting to do through this access method?" Then write a method for achieving that behavior rather than defining a getter/setter pair* (Wick et al., 2004).

Meyer's properties of the ideal class

Typical properties of the ideal class should according to Meyer (1997) adhere to the following:

- Clear Abstraction: a clearly associated abstraction.

- Adequate Class name: a descriptive noun or adjective characterising the abstraction.

- Representation: the class represents a set of possible objects.

- Interface for Properties: distinct access to properties of an instance

- Interface for Manipulation: methods for changing the state of an instance and to ask object to perform according to the behaviour of the abstraction.

- Abstract properties stated: invariants, pre- and post conditions preferably formally described.

5.8   Summary

Heuristics and rules for object oriented design, are advice regarding different aspects of design and even code. They can very well be contradictory, and even though they may be detailed, most of them do not explicitly declare limits or thresholds for different concepts or situations. Riel's grouping of heuristics is beneficial, but the amount of heuristics makes them less practical for the small-scale situation. Garzás and Piattini (2007b) combine acknowledged design principles, rules and heuristics. However their rules are formulated as to detect situations in existing code, if RULE ... . They are advice of what to do when things have gone wrong, rather than being advice for design. This is similar to the "code smells" discussed in Chapter 6. When thinking about how to design a small-scale example we lack advice on what to aim for, rather than what to avoid. Therefore the suggested He[d]uristics will be stated in a positive and constructive way.

Despite the difference in number, the heuristics presented by different authors have some things in common. It is our view that they point towards abstraction as a key concept (Bashar Molla, 2005). It is also commonly stressed that protocols should be separated from the internal representation of the abstraction.

As we see it, some of these advices are not applicable to the small-scale context, and those that are, can in most cases be regarded as consequences of the more general principles considered in Chapter 4.

Chapter 6

Design Patterns and Code Smells

6.1 Introduction

The first reference of (design) patterns always seem to be to the work done by Christopher Alexander, an architect reasoning about construction of buildings, cities, rooms etc. He has been frequently cited since the late 60'ies and apparently he has had a great influence on software engineering.

When it comes to design patterns for software construction, early references are made to work done in connection to Smalltalk in the 70's. The real starting point seems to be with the 23 design patterns collected and presented by Gamma et al. (1995) [1]. There are a number of definitions of what a design pattern is, some shown below.

> *[..] descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.* (Gamma et al., 1995)
> *Patterns provide a mechanism for rendering design advice in a reference format.* (Fowler, 2003)
> *Design patterns are not just about the design of objects, but about the communication between objects.*
> *[...] design patterns describe how objects communicate without become entangled in each other's data models and methods. Keeping this separation has always been an objective of good IO programming.* (Cooper, 2000)

It does seem undisputable though, that patterns are years of experiences of problem solving and design that have been formed into generalised solutions to common problems.

Another benefit is that patterns provide a vocabulary for designers. Undertaking a standard documentation style, though not entirely agreed upon, they make a

---

[1] Often refered to as the Gang of Four, or GoF

basis for a better understanding of solutions. They are powerful and aids, if used wisely, in making code more easily understood.

However, patterns in general require an understanding of abstract concepts and an ability to recognise these abstractions. A novice programmer setting out to learn the object oriented paradigm is no way near this understanding. Patterns are solutions to recurring problems that have evolved over time, i.e. not the kind of solutions a novice would come up with initially.

Nowadays patterns for various applications are designed and discussed both within and outside of software development, e.g., educational patterns. The granularity of patterns varies and maybe this is one of the reasons for the increasing number of patterns.

6.2 The Gang of Four Patterns

The design patterns described by Gamma et al. (1995) are classified in families of patterns. The classification is done by two criteria, purpose and scope. Purpose is what a pattern does and scope specifies whether the pattern is primarily applicable to classes or objects.

Class patterns deal with relationships between classes and their subclasses and are established through inheritance, which implies that the relationship is static since it is fixed at compile time. Object patterns deal with relationships between objects and are more dynamic since they can be changed at run time.

Creational class patterns: defers object creation to subclasses.

Creational object patterns: defers object creation to another object.

Structural class patterns: uses inheritance to compose classes.

Structural object patterns: describe ways to assemble objects.

Behavioural class patterns: uses inheritance to describe algorithms and flow of control.

Behavioural object patterns: describe how a group of objects cooperate to perform a task.

The patterns are classified according to two criteria. *Purpose*, which reflects what a pattern does, and *scope* which specifies if the pattern is applicable to classes or objects. See Table 6.1 for this classification and Appendix B for a short description of the patterns.

In general, patterns are dealing with rather complex problems that probably will be meaningful rather late in an introduction to object oriented concepts. But even experienced software constructors need concrete examples to understand how to apply these principles. When attempting to introduce object orientation the main contribution of patterns lies in the mind of the presenter.

Two important principles are stated as a base for the GoF-patterns:

- *Program to an interface, not an implementation.*

- *Favour object composition over class inheritance.*

Table 6.1: Design Pattern Space

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Singleton	Bridge	Command
		Builder	Composite	Iterator
		Prototype	Decorator	Mediator
			Facade	Momento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

6.3 The Model-View-Controller Pattern

One early pattern not included in the collection of The Gang of Four is the Model-Controller-View pattern. This pattern is based on the idea of separating the data from its representation. The *model* maintains the data, the *view* is one or more ways of displaying all or parts of the data and the *controller* is responsible for the communication between the views and the model. Communication between the user, the GUI and the data is carefully controlled and this separation of functions accomplished that nicely. Three objects talking to each other using this restrained set of connections is an example of a powerful design pattern.

Sun's Swing architecture is rooted in the model-view-controller (MVC) design that dates back to Smalltalk, see Figure 6.1 [2].

Model Code that implements the behaviour of some abstraction. Models a functionality without regards to presentation and client. The model informs its registered views when any of its functions cause its state to be changed.

View The view, or views, presenting the results of the model to different clients. The view informs the controller of any desired state-changing events generated by the user. In Java the views can be built from AWT or Swing components.

Controller The controller maps user interactions to state updates in the model. Dependent on events in the view, and the resulting changes in the model, the controller selects an appropriate view. In Swing the controllers are the listeners.

The example in Figure 6.2 by Bergin (2007)[3] shows a simple temperature model. The *model* encapsulates the notion of a temperature. It is possible to ask the model for the temperature in either Fahrenheit or Celsius. There are a number of different *views*, textual with buttons (`FahrenheitGUIt`, `CelsiusGUI`), a slider (`SliderGUI`) and a gauge (`GraphGUI`). The *controllers* in this example are the

---

[2]Picture: http://java.sun.com/blueprints/patterns/images/mvc-structure-generic.gif

[3]Slightly adapted for this presentation. Listeners are shown as separate classes to be "visible".
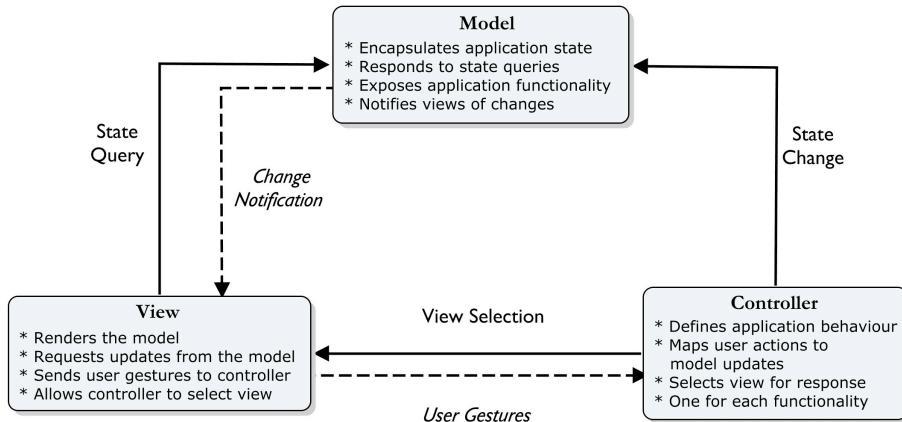
Figure 6.1: Model-View-Controller Pattern

listeners that implements the `Observer` interface. The listeners registers as observers with the model, that is `Observable` and notifies all `Observer`'s when the state changes. With this design it is possible to have multiple instances of the views located at different positions and still all views are reacting to the same notification caused by a state change in the model. The model sends an update message to all registered observers and they in turn queries the model for information and starts updating itself according to the retrieved information.
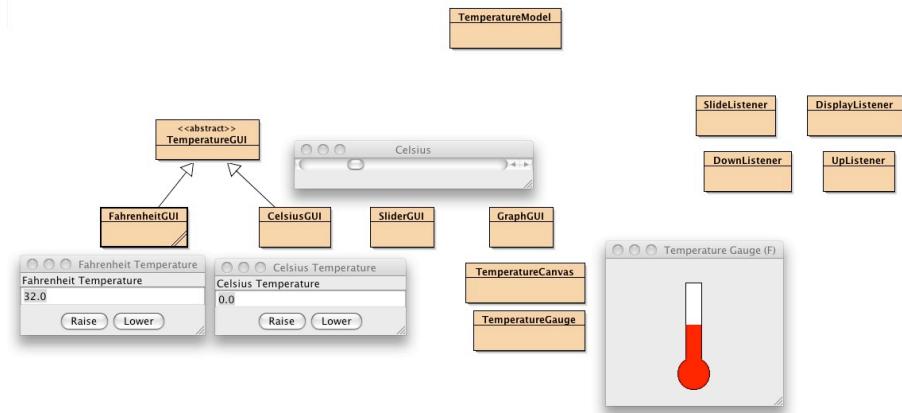
Figure 6.2: MVC Temperature

This design results in a flexible organisation of the solution. Adding another view does not require any modification of neither the model nor the existing views.

6.4 Micro Patterns: Low-level Patterns

An interesting development of patterns is the recently suggested idea of *Micro Patterns* by Gil and Maman (2005). These patterns are similar to design patterns but closer to the implementation, i.e. on a lower level of abstraction. They provide a way to establish a vocabulary, which makes it possible to talk about implementation details in a uniform way. They are based on object oriented concepts and reflect the structure and organisation of smaller components in an object oriented solution. Context is not important for these patterns since they are dealing with details that have to be taken care of in many different situations. This might be compared to algorithms for sorting. Micro patterns are defined as "class-level traceable patterns".

> *A pattern is said to be traceable if it can be expressed as a simple formal condition on the attributes, types, name and body of a software module and its components.* (Gil and Maman, 2005)

A small example fromGil and Maman (2005): a class with a public constructor and with one or more static public fields of the same type as the class itself, is defined as the *Sampler pattern* belonging to the *Controlled Creation category*. The purpose of such a class is to give clients access to pre-made instances of the class, but also a possibility to create objects on their own. The `Color`-class in Javas API is a typical example of this micro pattern.

In the paper (Gil and Maman, 2005) 27 micro patterns are collected in a catalogue. For visibility the patterns are grouped into 8 categories in a two-dimensional space with State and Behaviour as axes, see Figure 6.3.

As seen in Figure 6.3 there is overlapping among the 8 categories, i.e. one pattern can belong to more than one category. Gil and Maman also propose the term nano-patterns for traceable patterns, which stand at the method or procedure level. Furthermore they suggest the term *milli-patterns* to be used for traceable patterns at the package level (or to any other kind of class grouping or mode of cooperation). It is however interesting to notice that the occurrence of degenerated classes and methods is vast. Out of the 27 micro pattern presented in the paper 19 deals with degenerate state and/or degenerate behaviour. A short description of each micro pattern can be found in Table D.5.

6.5 Refactoring

One aim of object oriented software developers is to design reusable software. Object orientation is said to be particularly suitable for this purpose. Reusable software components are mostly the result of a iterative process, guided by experience. Nonetheless, reuse of software components is greatly enhanced if the components are made "changeable" from the beginning.

Refactoring is about changing the structure of software code without changing its external behaviour. Opdyke Opdyke (1992) makes the first definition of refactoring:

> *Refactorings are reorganization plans that support change at an intermediate level.*

63
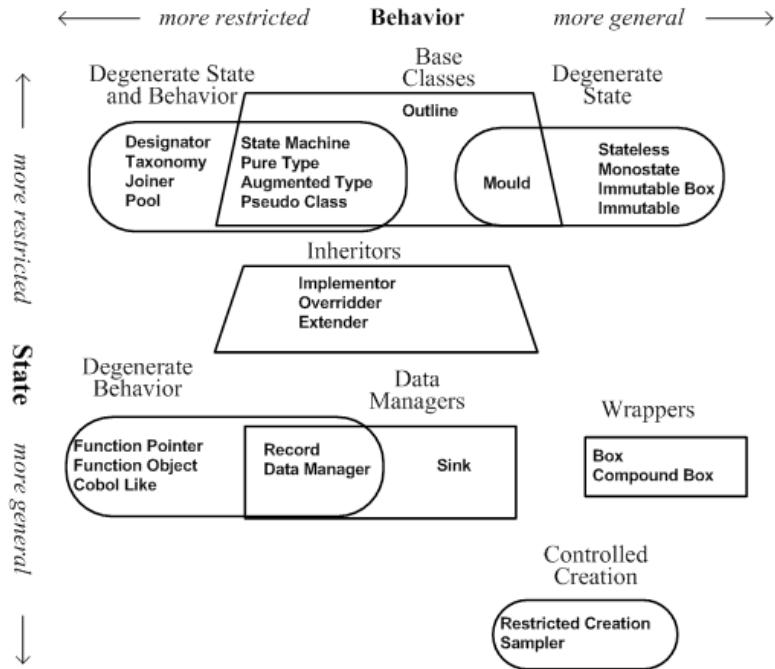
Figure 6.3: Micro patterns (Gil and Maman, 2005)

> *[. . .] Refactorings do not change the behavior of a program*
> *[. . .]they can support software design and evolution by restructuring*
> *a program in the way that allows other changes to be made more easily.*

In the first definition of refactoring Fowler (Fowler et al., 1999) has in a systematic way described seventy refactorings: the motivation for doing them, mechanics of how to do them safely, accompanied by simple examples. Refactorings are described in a standard format. The format consists of five components:

Name: important for creating a common vocabulary

Summary: short description of the situation that need refactoring and what kind of refactoring that solves the problem. Important in aiding the search for an appropriate refactoring.

Motivation: the reason for refactoring and exceptions when refactoring is to be avoided.

Mechanics: a detailed step-by-step description how to perform the refactoring.

Examples: illustrates the refactoring with a simple example.

In connection to design, refactoring is not a tool in itself, since the need for refactoring is a result of code evolving as a result of changes in design, added functionality and often over time.

6.6 Code Smells

To be able to detect if refactoring is an option Fowler et al. (1999), has named a number of candidate situations for refactoring. These are called "code smells". A smell is an indication of a potential problem, not a guarantee of an actual problem. Smells will occasionally find false positives—things that smell, but are actually better than the alternatives. However, there is no easy way to detect or measure the amount of "smell" or to what extent refactoring is needed, since code is context-dependent. A short description of the 22 code smells given in (Fowler et al., 1999) follows.

Smell name	Description
Alternative Classes with Different Interfaces	lack of common interface for related classes.
Comments	can be misused to compensate for bad structure, should only be used to clarify why, not what Data Class a class that contains data but no "real" behaviour
Duplicated Code	redundant code, never do anything more than once!
Data Clump	the same few data items passed around together, should be turned into an object.
Divergent Change	one class needs to be modified continuously??
Feature Envy	a method more interested in other classes than the one it belongs to
Inappropriate Intimacy	two classes too tightly coupled, classes should know as little as possible about each other.
Incomplete Library Class	using a library that does not offer sufficient services
Large Class	a class trying to do too much, indications are many attributes and/or methods.
Lazy Class	a class that does not do a proper job. Classes should pull their weight. Every additional class increases the complexity of a project. If you have a class that is not doing enough to pay for itself, can it be collapsed or combined into another class?
Long Method	Fowler and Beck strongly believe in short methods

Smell name	Description
Long Parameter List	Don't pass in everything the method needs; pass in enough so that the method can get to everything it needs.
Message Chains	Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise.
Middle Man	A class mostly delegating to other classes.
Parallel Inheritance Hierarchies	Every time you make a subclass of one class, you must also make a subclass of another. Consider folding the hierarchy into a single class.
Primitive Obsession	primitives are used instead of small classes/types. A really good analysis of this can be seen in (Fowler 2003b).
Refused Bequest	a subclass that does not fully support the data and method it inherits. If you inherit from a class, but never use any of the inherited functionality, should you be using inheritance?
Shotgun Surgery	the opposite of Divergent Change, small changes involves many classes. If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class.
Speculative Generality	code has been generated in case of future changes.
Switch Statements	type codes or type detection is used instead of polymorphism or type codes are passed on to methods.
Temporary Field	a class has an attribute used only in certain situations and probably should be a method variable.

Code smells might be considered the opposite of patterns, more like anti-patterns, i.e. code that has certain "bad" object oriented-characteristics that need to be replaced. Originally Fowler et al. described 22 code smells, which is a lot to keep in mind, especially if the aim is to avoid them from the very beginning. To make it somewhat easier Mäntylä has defined a taxonomy for a more easy use of these code smells (Mäntylä, 2003). The ambition is to organise the smells into a higher level of abstraction. The result is five categories, see Table 6.3. However, as

stated in (Mäntylä, 2003) the grouping could be debated and many of the smells would be appropriate in several of the groups. Later on the taxonomy has been revised (Mäntylä).

Table 6.3: Taxonomy of code smells (Mäntylä, 2003)

Group	Smells	Explanation
The Bloaters	Long Method Large Class Primitive Obsession Long Parameter List DataClumps	Something that has grown so large that it cannot be effectively handled.
The Object-Orientation Abusers	Switch Statements Temporary Field Refused Bequest Alternative Classes with Different Interfaces	Cases where the solution does not fully exploit the possibilities of object oriented design
The Change Preventers	Divergent Change Shotgun Surgery Parallel Inheritance Hierarchies	Prevents or hinders the change or future development of the software.
The Dispensables	Lazy class Data class Duplicate Code Dead Code Speculative Generality	Something unnecessary that should be removed from the source code. [Dead Code is Mäntylä's suggestion of an added code smell]
The Couplers	Feature Envy Inappropriate Intimacy Message Chains Middle Man	Three of the smells represent high coupling. Middle Man smell on the other hand represent a problem that might be created when trying to avoid high coupling with constant delegation.
Others	Comments Incomplete Library class	Have nothing in common except that they do not fit into any of the other groups.

Based on the smells by Fowler et al. (1999), another set of "Smells and Heuristics" are formulated by Martin (2008). Martin categorises his advice in six categories: Comments (5), Environment (2), Functions (4), General (36), Java (3), Names(7), and Tests (9). Yielding a total of 66 smell and heuristics. One of the mot important rules, according to Martin, is *G5: Duplication*. It is also known as the DRY principle.

> *The DRY (Don't Repeat Yourself) Principle states: Every piece of knowledge must have a single, unambiguous, authoritative representation within a system. (Hunt)*

6.7 Anti-patterns

A pattern starts with a recurring problem and suggests a solution that has proven to work well and inhibit appreciated characteristics. The idea of Anti-patterns is to provide a way to capture bad software development practices. Some descriptions of AntiPatterns:

> *A literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences* Brown et al. (1998)
> *An AntiPattern is a pattern gone bad. It is a frequently used, but largely ineffective solution to a problem. A well-formed AntiPattern*

> *provides a structured way to describe the initial problematic solution and the most appropriate way to refactor that solution.* Jimenez (2006)

As the case with patterns, AntiPatterns provide a vocabulary to talk about software designs and solution, but in this case "bad" ones. Once a problem is detected the AntiPattern suggests a refactoring to obtain a more functional solution.

Intent of patterns Logically unique solution

Intent of AntiPatterns Awareness of situation and alternative solutions

Part of the vocabulary is the use of an AntiPattern template. This way the mapping of problematic situations is simplified. One of the components of the AntiPattern template is Root causes, a section that lists typical causes for the failed solution.

Apart from these components there are others to be added depending on the context. Antipatterns are categorised in three groups, based on the view of the Developer, the Architect or the Manager.

- *Software Development AntiPatterns:* aiming at solving programming problems encountered by the programmer.

- *Software Architecture AntiPatterns:* focuses on problems dealing with the structure of systems. The functional partitioning of software modules., the software interfaces between modules and the selection and characteristics of the technology used to implement the interface connections between software modules.

- *Software Project Management AntiPatterns:* describe how people issues, processes, resources, and external relationships impair software projects. Many of the AntiPatterns can address problematic situations in more than one of the views. Since development is of most interest to small-scale situations some examples of Development AntiPatterns are listed in Table 6.4.

Patterns and AntiPatterns are intrinsically linked since over time and/or due to change of requirements, maintenance, bug fixing etc. Any well-behaved code developed using patterns might evolve into an AntiPattern. An AntiPattern is a pattern in an appropriate context, Se Figure 6.4 from Brown et al. (1998)

In Brown et al. (1998) the seven sins of programmers are stated as root causes and the fundamental context for AntiPatterns. They are:

Haste: Hasty decisions lead to compromises in quality.

Apathy: Unwillingness to attempt a solution. In object orientation it might lead to lack of partitioning, and the definition of proper interfaces between classes.

Narrow-mindedness: The refusal to practice solutions that are otherwise known to be effective.

Sloth: Poor decisions based on the easiest answers leads to lack of configuration control.
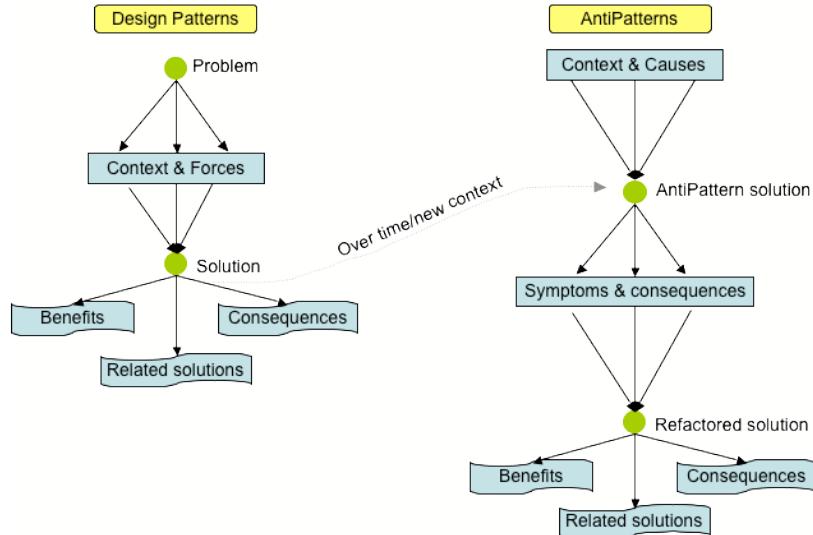
Figure 6.4: Design patterns and AntiPatterns relation. (Brown et al., 1998)

Avarice: (Greed) Architectural avarice, which means the system is overcomplicated with excessive detail, which leads to excessive complexity due to insufficient abstraction.

Ignorance: The intellectual sloth, not seeking to understand.

Pride: The not-invented-here syndrome. Object technology is the opposite of this and code reuse simplifies work and makes development less risky.

One example is *Cut-and-Paste Programming*. The root cause for this AntiPattern is *Sloth*. And in the template entry *Anecdotal Evidence* for this AntiPattern it says:

> *"Hey, I thought you fixed that bug already, so why is it doing this again" "Man, you guys work fast. Over 4000,000 lines of code in three weeks is outstanding progress!"*Brown et al. (1998)

The work of Brown et al. (1998) was published before the Refactoring/Code smell work published by Fowler et al. (1999) and does not seem to have been further developed. In its initial form AntiPatterns are less formalised and provides less constructive solutions than the ideas of Refactorings and Code smells.

6.8 The Grand Mistake in Design

According to Meyer (1997) the most common and most damaging mistake, what he calls *The Grand Mistake*, is designing a class that isn't. The following are indications of this mistake:

- "My class performs...": a class is supposed to offer a variety of services on objects of a certain type, not dedicate itself to something that ought to be a method of some other class.

- Imperative names: class names consisting of imperative verbs could very well be an indication of a class that does only one thing (printing, sorting etc.). The name should be carefully chosen to clearly reflect the abstraction. For classes this means a noun, and for interfaces in Java an adjective.

- Single-routine classes: a class that contains only one non-private method might be just a wrapped subroutine/procedure.

- Premature classification: a common mistake among novices is to introduce inheritance to early in the analysis and design work. Since abstractions are the core of object orientation, inheritance is only relevant as a relation among well-understood abstractions. Inheritance may well be a result at a later stage due to the iterative way of working that is necessary to discover abstractions and relations. To a novice this indication might be pointing at the class-object confusion.

- No-command classes: these are classes without any behaviour at all; they are the equivalents to records in Pascal and structs in C.

- Mixed abstractions: a class whose features relate to more than one abstraction. This is summarised in the Class Consistency principle: All the features of a class must pertain to a single, well-identified abstraction.

6.9   Summary

The origin of code smells is code that has evolved over time, and this makes code smells less applicable in the small-scale context. Knowledge of code smells can be motivated by the need to avoid pitfalls, but for practical use in an educational setting their value is limited. It is well known that the use of negation is dangerous in teaching and knowledge transfer. In a recent study, Kotzé et al. (2008) found that students showed signs of confusion when being exposed to negative guidelines. However, it is important from an instructional point of view, to address common misconceptions (discussed in Section 2.5) to aid novices in their understanding of concepts (Hewson, 1981).

Patterns are the results of years of experience made by professional software developers. In a small-scale example, the use of design patterns creates an unmotivated overhead that complicates the example beyond the scope of its purpose. The use of patterns in a small-scale situation is therefore limited and not used as an explicit resource in this work to discuss examples. It is however important to keep the two basic ideas of pattern in mind when designing examples: *Program to an interface, not an implementation* and *Favour object composition over class inheritance*. The intention of these two principles is implicitly useful, think twice before using inheritance and use interfaces to show the possibilities of extension and more general code. In other words: delay the introduction of inheritance until the concept can be rightfully justified by proper examples. Patterns in general, favors the use of interfaces and composition. This is important to take into consideration in the small-scale context.

Table 6.4: Software development AntiPatterns (Brown et al., 1998)

AntiPattern	Synopsis	Refactored Solution
Cut and Paste Programming	Code reused by copying source statements leads to significant maintenance problems.	Black Box reuse reduces maintenance issues by having a common source code, testing, and documentation for multiple reuses.
Functional Decomposition	Non-object oriented design (possibly from legacy) is coded in object oriented language and notation.	No straightforward way to refactor: redesign using object oriented principles.
Golden Hammer	A familiar technology or concept is applied obsessively to many problems.	Expanding the knowledge of developers through education, training, and book study groups exposes developers to new solutions.
Lava Flow	Dead code and forgotten design information is frozen in an ever-changing design.	Configuration control processes that eliminate dead code and evolve/refactor design towards increasing quality.
Poltergeists	Classes with a very limited roles and life cycles, often starting processes for other objects.	Allocate the responsibility to longer-lived objects and eliminate the poltergeists.
The Blob	Procedural-style design leads to one object with numerous responsibilities and most other objects only holding data.	Refactor the design to distributed responsibilities more uniformly and isolate the effect of changes.

Chapter 7

Software Metrics

7.1 Introduction

Software metrics are used for different purposes. One is to estimate the work spent on constructing the software. Another purpose is to predict maintenance costs. In this chapter, a number of metrics are presented, without the ambition to be exhaustive.

7.2 Classical Metrics

Historically there are three different metrics that is the starting point for code-metrics.

LOC – Lines of code

This is to be considered more a measure of length and might be regarded as the size of program representation because of its insensitivity to the language and paradigm of the measured program.

HSS – Halstead software science

This measure was suggested in 1977 and is trying to estimate the programming effort. The number of appearing operators and operands are estimated. Operands can be "+" and "*" but also an index "[...]" or a statement separator ";". Operands are litteral expressions, constants and variables. A brief description of HSS can be found in Christensen et al. (1981). There are four basic program measures:

- η_1 Number of unique operators used.

- η_2 Number of unique operands used.

- N_1 Number of times operators are used.

- N_2 Number of times operands are used.

Vocabulary (η) of a given program is defined as the sum of unique operators and operands used in that program, and is a measure of the repertoire of elements that a programmer must deal with to implement the program. *Vocabulary* is therefore defined as: $\eta = \eta_1 + \eta_2$

Length (N) of a given program is defined as the sum of the operator usage and the operand usage. Intuitively, length is a measure of program size and measures the number of times a programmer deals with each of the programming elements. *Length* is expressed as follows:

$$Length: \ N = N_1 + N_2$$

Halstead suggests a relationship such that *Length* can be estimated from *Vocabulary*. The formula for *Estimated Length* ($\widehat{N}$) based on *Vocabulary* is the following:

$$Estimated\ length: \ \widehat{N} = \eta_1 log_2 \eta_1 + \eta_2 log_2 \eta_2$$

According to (Christensen et al., 1981) *Estimated Length* tends to be low for large programs and high for small programs and seems to most accurate in the range of 2000-4000 units of length. Halstead also suggests a two-dimensional measure of program size. This measure considers the frequency of elements in connection to the range of possible elements. This is defined as the *Volume* of a program:

$$Volume: \ V = N log_2 \eta$$

It is stated in (Christensen et al., 1981) that these measures are consistent with lines of code as relative measures of length.
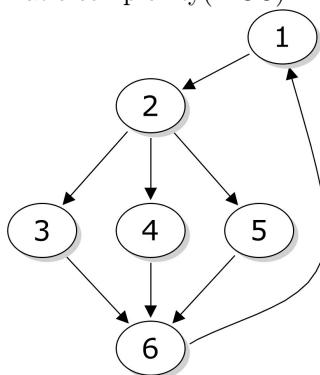
MCC − McCabes Cyclomatic Complexity

This complexity measure is based on a mathematical approach to measure and control the number of paths through a program (McCabe, 1976). Graph theory is used to represent decision structure from which measure of control flow is derived. An example is shown in Figure 7.1.

Figure 7.1: Example of McCabes Cyclomatic complexity(MCC)


```
For x = 1 To 5
  If x = 1
    Then "x=1"
    ElseIf x = 2
        Then MsgBox "x=2!"
        Else: MsgBox "x is
            higher than 2!"
  End If
Next
```

Listing 7.1: Code

Corresponding Graph

For practical reasons an edge from exit to entry is added.

The cyclomatic number, the number of independent paths through strongly connected directed graphs, is calculated as:

$$M = E - N + 1 = 8 - 6 + 1 = 3$$

where

E = the number of edges of the graph

N = the number of nodes of the graph

and the cyclomatic complexity is calculated as:

$$M = E - N + 2 = 8 - 6 + 2 = 4$$

This means that the example in Figure 7.1 has the cyclomatic complexity:

$$M = 8 - 6 + 2 = 4$$

Since Cyclomatic Complexity is based entirely on control flow, its major application in object orientation is on a method level. Complexity raises in methods that have a lot of if statements, for and while loops etc. Values over 10 are generally viewed as being bad.

A practical use of the cyclomatic complexity is that it is said to be a count of the number of test conditions of a program.

Hybrid metrics

Based on the ideas of the aforementioned metrics, hybrids have been developed that are more representative of the complexity of a program since they combine measures for size control and flow information flow.

Maintainability Index (MI), developed at the University of Idaho, is using Halstead's effort and McCabe's cyclomatic complexity, and other factors relating to the number of lines of code (JHawk Webpage). Maintainability is the focus of MI and it is stated to be language independent and was validated in the field by Hewlett-Packard (HP).

7.3 Object Oriented Metrics

Traditionally in metrics, complexity that was inherited was ignored. This is a mistake since inheritance adds additional complexity to the modularity equation.

There has been an adoption of classical metrics to object orientation, and the Halstead metrics and McCabe's cyclomatic complexity is still relevant. All of the measures can be applied at method level, and can be extended as averages on a class level.

When it comes to design, the interpretation of different metrics can aid in indicating design flaws.

7.3.1 The first theoretically founded object oriented metric

Gibbon (1997) states that the object oriented-metrics by Chidamber and Kemerer (1991) were the first to be both theoretically and empirically validated. They form a basis for many subsequent metrics. This set of metrics is based on the essential features of object oriented design as presented by Booch (1994). Four major steps involved in the object oriented design process are outlined.

- Identification of classes and objects at a given level of abstraction: In this step, key abstractions in the problem space are identified and labeled as potential classes and objects. It also serves the purpose of establishing the boundaries of the problem.

- Identify the semantics of these classes and objects: The meaning of the classes and objects identified in the previous step is established. This means defining the behaviour and attributes of each abstraction.

- Identify relationships between classes and objects: In this step, class and object interactions, such as patterns of inheritance among classes and pattens of visibility among objects and classes (what classes and objects should be able to "see" each other) are identified. The purpose is to formalise the conceptual as well as physical separation of concern among abstractions.

- Implementation of classes and objects: In this step, detailed internal views are constructed, including definitions of methods and their various behaviors.

One important aspect of these metrics is that they are directly aiming at the design of objects (or classes). The elements of the Chidamber and Kemerer metric are listed in Table 7.1.

Table 7.1: The metric suite (Chidamber and Kemerer, 1991)

Weighted Methods per Class (WMC)	Sums the complexity of individual methods.
Depth of Inheritance (DIT)	Asserts that design complexity increases with depth of the inheritance hierarchy since more classes and methods are involved.
Number of Children (NOC)	Number of immediate subclasses to a class. This number indicates of the potential influence a class has on the design.
Coupling between Objects (CBO)	The number of non-inheritance related couples between classes (use of methods and/or attributes).
Response For a Class (RFC)	The set of all methods available to an object. The response set of a class is the set of all methods that can be invoked in response to a message received by an object of that class
Lack of Cohesion in Methods (LCOM)	Amount of use methods make of the objects attributes. Identifies disparity between a class' methods and its data, indicating flaws in the design of the class.

The design of classes is declared to be central to the object oriented paradigm. Chidamber and Kemerer's metrics are designed to measure the complexity in the design of classes. The limitation is that dynamic behaviour of a system is not captured.

7.3.2 Object oriented design metrics

According to Lanza et al. (2005)it is necessary that an overview of an object oriented system include metrics that reflect on three main aspects, using the following metrics (both directly and in computed proportions):

1. *Size and complexity.* How big and complex is the system

   - NOP - number of packages
   - NOC - Number of Classes
   - NOM - Number of Operations[1]
   - LOC - Lines of Code
   - CYCLO - Cyclomatic Number, the sum of all McCabe's cyclomatic number for all operations

2. *Coupling.* Collaboration is at the core of object orientation.

   - CALLS - Number of Operation Calls. Distinct invocations, i.e. the sum of all user-defined operations.
   - FANOUT - Number of Called Classes. A sum of the FANOUT metric, defined as the number of classes referenced by a class.

3. *Inheritance.* Considered a major asset, inheritance must be measured and analysed.

   - ANDC - Average Number of Derived Classes, i.e. the average number of direct subclasses of a class (interfaces not counted).
   - ANH - Average Hierarchy Height. An average of the *Height of Inheritance Tree* (HIT) among the root classes defined in the system (interfaces not counted).

Whatever metrics used, they are not to be considered absolute measures of quality.

> *Metrics measure structural elements and as such they can reveal hidden symptoms. But there will always be a gap between the symptoms and the deep assessment that an expert of object oriented design can do using these symptoms.* (Lanza et al., 2005)

---

[1] User-defined operations, including methods or global functions.

7.3.3 Readability metric

In the small-scale context it is important to consider the skill to read and understand written code. Understandability and readability are interesting qualities of program code that has yet to be explored. Although there is a large body of literature on software measurement, no publications on measures for software readability could be found.

Basic syntactical elements must be easy to spot and easy to recognize, only then, one can establish relationships between the elements and form more abstract schema or chunks. In Börstler et al. (2007) we make an attempt to formulate a reading ease score for software (SRES).

SRES is based on a readability measure for ordinary text. By interpreting the lexemes of a programming language as syllables, its statements as words, and its units of abstraction as sentences, we could then argue that the smaller the average word length and the average sentence length, the easier it is to recognize relevant units of understanding, so-called "chunks". Since abstraction is a key programming concept, proper chunking is highly relevant for the understanding of programming examples.

A comparison of SRES with other significant measures is performed. The conclusions of the initial tests, is that SRES, along with the other measures, favours code with a high degree of decomposition. Decomposition is useful in managing complexity, and therefore important from a cognitive point of view. However, decomposition in itself is no guarantee for readability, the decomposition must capture meaningful abstractions in the problem domain. Furthermore, there are many important aspects of readability and understandability not covered by any measure, e.g. choice of names, commenting, and indentation,.

7.4 Summary

Object oriented-metrics are mainly focused on class, method and/or system-level analysis as shown by Purao and Vaishnavi (2003). Their survey shows that research-activities were intense during the mid-90's, but has decreased, and they regret the lack of consensus about definitions of quality characteristics.

In a small-scale context we would like to avoid overly complex examples, to reduce the cognitive load on the novice. Complexity of systems and classes is interesting but difficult to measure. The depth and width of inheritance hierarchies can be measured, and might be evaluated in terms of complexity. The number of classes, attributes and methods and how they interact could be used as indicators. Coupling and cohesion is also among the components mentioned in different metrics.

When dealing with small-scale problems it is important to keep the cohesion high, i.e. to make sure that an abstraction is focused and limited. This is well captured by *The Single Responsibility Principle*, see Section 4.2 and (Martin, 2003).

Chapter 8

He[d]uristics

Small-scale brings about restrictions on the design of examples not present in the design of large-scale object oriented systems. 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 particular about the context/problem domain that we supply for the examples.

The intention of the proposed He[d]uristics is to support the design of exemplary examples, with respect to the characteristics of object orientation, and in addressing novices' particular needs. For practical reasons it is important to avoid being too general or too detailed. Therefore, the proposed He[d]uristics are targeted towards general design characteristics, which means that more detailed practices, like keeping all attributes private, are not stated explicitly. The He[d]uristics are designed to be independent of a particular line of presentation (objects first/late, order of concepts, ..) and environment used.

The proposed He[d]uristics are:

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*

In this chapter the He[d]uristics are presented, each with a short description and motivation. Examples and discussions of consequences follow in Chapter 9. Finally, the proposed He[d]uristics are evaluated against established characteristics and practices in Chapter 10.

8.1   Model Reasonable Abstractions

This is maybe the most important He[d]uristic. Abstractions are at the heart of object orientation and we need to enforce the right approach to abstractions. So what is a reasonable abstraction?

In the small-scale context it has two implications.

- An abstraction has to be object oriented and act as a role-model for objects. Implementing the abstraction, we must not sacrifice essential object oriented characteristics for the purpose of saving space, in terms of number of classes, lines of code etc.

- It also 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 of 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.

To promote the understanding of objects, it is important to show the basic characteristics of an object: objects have identity, state and behaviour. Among other things, this implies that classes being mere data containers are not exemplary.

In view of the Single Responsibility Principle (SRP, Section 4.2), we can assume that in terms of responsibilities, small is beautiful. For the small-scale context this should imply few attributes and few methods. Furthermore, the limiting conditions of a small-scale example results in few lines of code, and keeping the abstraction focused with few collaborators means less passing of parameters. Encapsulation and information hiding must be emphasized.

Working with composition to have slightly more complex classes, it is handy to use compositions from real life as examples. But, with every day life examples it is important to explicitly discuss the difference between the model and the modelled. It is difficult for the inexperienced to accept that the model can have behaviour and responsibilities that a static, dead thing from our every day reality never would have.

Another common pifall is to model roles instead of classes. Whether a domain entity is a candidate for an abstraction or merely a role that some abstraction can take on, must be decided carefully.

8.2   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". It is however crucial to avoid artificial behaviour because it may distract the novice from understanding the basic concept of behaviour in an object. 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 thinking.

The consequences of demanding a reasonable behaviour makes some common habits improper. Printing for tracing is one example, this confuses the novice of how things are returned from methods and spoils the idea of having I/O separated from the functional parts of a system. Snippets of code is another. It does not promote object-thinking and leaves the novice with the extra cognitive burden of constructing a meaningful context for this code to work. Setters and getters connected to the implementational details (attributes) of a class can break the idea of encapsulation and information hiding (Section 5.7), and should therefore

be avoided as much as possible. This can be avoided by consistently discussing and exemplifying the separation of the implementation from the abstraction. Objects should provide services, not merely be wrappers of data.

8.3 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 for designing small-scale examples. It is crucial to define the responsibilities and services of an object separately from the internal representation and implementation of the attributes. Leaving the implementational details out from the design thinking will promote object thinking and make problem solving easier for the the novice.

A practical advice is given by Meyers (2004), anticipate *what clients might want to do* and *what clients might do incorrectly*. The interface/protocol of a class must be carefully designed, and be consistent with the problem domain. It should also be as complete as possible, even if some services would not be immediately requested. On the other hand, it is important not to burden clients with functionality they are not interested in. *The Single responsibility Principle* (SRP, Section 4.2) and *The Interface Segregation Principle* (DIP, section 4.6) both addresses this problem.

8.4 Favour Composition over Inheritance

Inheritance[1] is the concept that distinguishes object orientation from other paradigms, but it is considered difficult to learn and is therefore given a lot of attention. Introducing it early often results in examples that are too simple to show the strength of inheritance. Since novices still 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 composition is no less important to show this feature of object orientation. Being restrictive with inheritance is important to guide novices toward a proper use of inheritance. Polymorphism is powerful, and its strength can be demonstrated by interfaces and abstract classes as well as by inheritance. The distribution of responsibilities and the importance of proper protocols can be shown through the collaboration among objects.

Many classes in Java's API are implementing interfaces, and separating the introduction and use of behavioural (interfaces) and structural (inheritance) relationships is probably a good idea. Late introduction of inheritance makes it possible to show its benefits through more advanced examples. The Gang of Four-patterns (Chapter 6) strongly build on the recommendation *Favour composition over inheritance*. Skrien (2009) shows many examples of how this can lead to better designs.

8.5 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

---

[1]By inheritance we mean subclasses in Java (implementation inheritance) rather than interfaces in Java (interface inheritance).

clients view and the context makes the proposed design plausible, we may unintentionally contradict the general intention of an example. With this He[d]uristic we attempt to capture common pitfalls and to address some of the misconceptions described in section 2.5.

To show the idea of objects, we should strive for "many" objects present in the small-scale example. A common example often consists of only one or two references of a certain class, and objects are instantiated to show some of their functionality. But, in a "normal" situation there ought to be many instances of a class, because otherwise it seems like a questionable abstraction. Questionable since a class is a template for creating objects of the same type. This is not unproblematic, since the presence of many objects raises another problem: how should we handle many objects?

Another common pitfall is to use "one-of-a-kind" classes. If there is no need for more than one object of a certain class, it does not support the novice in understanding the conceptual difference between the object and the class.

It is also important to be explicit, i.e. using explicit objects whenever possible. Following the Law of Demeter (Section 4.7) is one way to make a design more explicit. Calling methods of explicit objects instead of calling the nameless object resulting from a 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. Implementing methods in the formal definition of another method, e.g., the signature, does not promote the understanding of objects and their behaviour. This is more suitable for the more experienced learners. Since tracing is an important part of learning to program, avoiding anonymous objects and classes, will decrease the cognitive load for a novice (Section 2.3).

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. Including the `main`-method in an abstraction means breaking the concept of abstraction and encapsulation. A class is the detailed description of an abstraction, used to instantiate objects from this abstraction. Having a `main`-method to be able to test the class, including the creation of an object of the class itself, does not lead to reasonable behaviour or abstractions. The method `main` is in itself 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.

8.6 Make Inheritance Reflect Structural Relationships

Inheritance is often over-emphasized and misused when introducing hierarchical structures early (see Section 8.4). 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, see Section 4.4) 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.

It is confusing for a novice that it is possible to instantiate both the the base class and the derived class, and it can not be considered exemplary to design structures this way. To eliminate this possible class/object conflation it is critical to adhere to one of the consequences of *The Dependency Inversion Principle* (DIP, see Section 4.5) and the basic idea of design patterns (see Section 6.2): *Never derive a class from a concrete class.*

Chapter 9

He[d]uristics in Practice

In this chapter, we demonstrate and discuss the intentions of the proposed He[d]uristics through examples. Different implications are discussed and suggestions for improving existing examples are made. This chapter should provide insights into the intended use of the proposed He[d]uristics and hopefully new insight and inspiration for the design of small-scale examples.

9.1 Model Reasonable Abstractions

Abstraction is at the heart of object orientation, and that makes this He[d]uristic one of the most important ones.

The ambition must be to make the abstraction *plausible*, both from a software perspective as well as from an educational perspective, all seen through the eyes of a novice.

One important implication of *Model Reasonable abstractions* is that context is critical to the abstraction. What differs good from bad is often in the details. Throwing an example up on the board is tempting, but making up examples on the fly is unfortunately hazardous, unless one possesses a set of carefully worked out examples. An obvious risk is to either violate the 'good' properties or realising the need for context and thereby making the example overly complex.

Illustrating smaller syntactical components it is common to use a complete application and place the example in the `main`-method. Listing 9.1 shows a simple example illustrating the for-loop.

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

Listing 9.1: Using a complete application illustrating syntactical elements.

This example is not contributing to the understanding of object orientation if we want to promote the idea that (almost) everything is about objects, and that they communicate through methods. `main` is a method not representing a behaviour of a certain type of object, and it is never explicitly called. So to avoid this potential confusion, *do not make main into the entire program.*

Decentralised responsibilities is a key component in object orientation. A reasonable abstraction should preferably have one responsibility. Therefore, *God classes must be avoided*, since large classes contradict the idea of focused abstractions (Riel, 1996).

One common mistake when designing small-scale examples is to model roles instead of classes. It is desirable that classes are made up of more than primitive attributes, and examples like the one in Figure 9.1 is not uncommon.
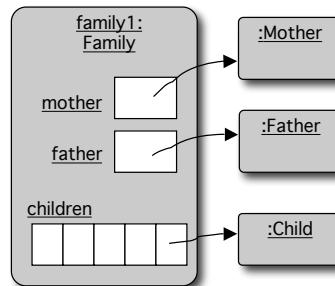
Figure 9.1: Classes that model roles (Riel, 1996).

The classes `Mother`, `Father` and `Child` are probably only persons having a certain logical interpretation in a given context. It is not unreasonable to assume that the same person can be mother in one family and child in another. Which yields the revised example in Figure 9.2.
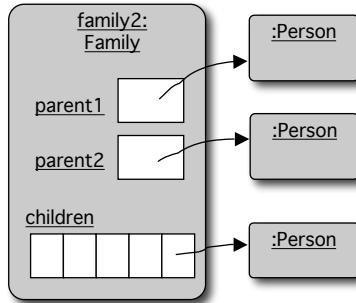
Figure 9.2: Objects with the same behaviour modelled by one class.

This design is closer to Dijkstra's definition of abstraction – *An abstraction is one thing that represents several real things equally well.* The important question is whether or not the distinction between Mother and Father is in behaviour, which would justify the use of different abstractions. Choosing the design of Figure 9.2 makes the class reusable and the abstraction more exemplary. Another frequent example of this, is students taking courses at a university department. Being a student is a role a person can take in one situation, while in another situation the same person might be a lecturer. This is especially true if the role is temporary. If the decision on what to model is based on the perspective of a client and situated in the problem domain, it becomes easier to *model classes, not roles.*

Deciding on abstractions must generally be made with *real objects*[1] in mind. A real object must be able to act as a role-model for object thinking, i.e. it must have identity, state and behaviour. Thinking in terms of responsibilities when choosing/designing the abstractions of a problem, might be one way of avoiding too much focus on implementational details. Christensen (2005) suggests to start the analysis and design with "There must be an object that is responsible for . . ."

The Single Responsibility Principle (Section 4.2) applied in a small-scale context should result in keeping classes, number of attributes, number of methods and number of parameters small, i.e. *keep things small.*

9.2 Model Reasonable Behaviour

The demand for objects to have reasonable behaviour is easy to agree on. We argue that what is reasonable is decided by the context. If the context is not supplied, the meaning of the concept behaviour is difficult to understand. One example of this, is how some specific construct in the language is illustrated. To demonstrate this, it is tempting and practical to show a few lines of code without context. The for-loop might be illustrated with the snippet in Listing 9.2.

---

[1]Note that "real objects" in this case is referring to objects showing the characteristics of object orientation, not real world object.

```
  int i = 0;

  for (int j=0; j<10; j++)
  {
    i = i+j;
  }
```

Listing 9.2: Small example without context.

But, just looking at the source code in Listing 9.2 does not aid the understanding of how and when things happen when the code is executed. What kind of object would have this behaviour and what would the corresponding responsibility be? It is even contradictory to how the concept illustrated, the for-loop, is supposed to be used 'for real'. It is not reasonable to have code that accumulates the sum of the integers 0..9. This is an artificial problem and would never appear in any programming problem, not even in the small-scale context. *Avoid using snippets*, since they do not promote the idea of behaviour.

Next it is tempting to use printing to show the values of the variables as the execution runs through the code, see Listing 9.3.

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

Listing 9.3: Printing for illustrating syntactical elements.

Seeing these lines of 'tracing' in almost every small example might lead the novice to believe that it actually is necessary to do the printing to 'get things done'. Furthermore, since the understanding of methods and their invocation heavily relies on the mathematical foundation of functions, the 'printing-effects' are even more dangerous when used in small methods, as in Listing 9.4.

```
private double x,y; //attributes
...
public double average()
{
  double temp = (x+y)/2;
  System.out.println( "Average : " + temp);
  return temp;
}//average
```

Listing 9.4: Printing for tracing in a method.

Now the novice is absolutely certain that printing is the only way to get results out of methods. However, external I/O is not a natural behaviour of objects. If anything, we should redefine the `toString`-method and retrieve the information by method call. Avoiding external I/O, means much more work to illustrate the behaviour of different language constructions, even on the very basic level. By rewriting the example, the notion of objects being autonomous entities providing services through the public interface of the class can be reinforced. It could also be argued that printing is just one way of displaying information and that is not the responsibility of the class to decide *what* the results are or *how* and *when* they are to be displayed. By adding another method, a client using these objects can, if needed, display some functionality of the method, see Listing 9.5.

To avoid this *do not use explicit printing for tracing* mixed up with the definition of an objects behaviour.

```
private double x,y; //attributes
...
public double average()
{
    double temp = (x+y)/2;
    return temp;
}//average

public String averageAsString()
{
  return "Average is : " + average();
}//averageAsString
```

Listing 9.5: Printing deferred to the client of the object.

Reasonable behaviour also conflicts with the general "textbook-rule" to have attributes private, and to use setters and getters for the attributes, which easily leads to dumb storage classes. It is important to *avoid classes consisting of only setters and getters* (and maybe `toString`). They tend to support the impression that classes are merely wrappers for values, and they are not exemplary of what should be considered behaviour. Behaviour should, preferably, be separated from the internal implementation of the abstraction, and emphasize the clients view.

In Listing 9.6, that happens to be one of the first user-written classes in a textbook, the class is not in any way different from having a simple variable of primitive type. Why bother all the overhead? In this case objects only seem to complicate things.

```java
public class Num
{
  private int value;

  public Num (int update)
  {
    value = update;
  }
  public void setValue (int update)
  {
    value = update;
  }
  public String toString ()
  {
    return value + "";
  }
}
```

Listing 9.6: A wrapper-class (Lewis and Loftus, 2007).

Another example of confusing behaviour, is the `Die`-class in Listing 9.7. There are several issues with this implementation of a die-abstraction, when it comes to reasonable behaviour. Is it reasonable to expect a newly created `Die`-object to have the face value 1; why is it not a random value? In method `roll`, the face value result of the roll is returned, even though there is a `getFaceValue`-method. *The Single Responsibility Principle* also applies to methods. Is this design chosen just to serve the testing, or is it the result of a carefully designed abstraction in a presented context?

```
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 void setFaceValue (int value)
  {
    faceValue = value;
  }
  public int getFaceValue()
  {
      return faceValue;
  }
  public String toString(){   ...   }
}
```

Listing 9.7: The `Die`-class (Lewis and Loftus, 2007).

A slight adjustment of this example makes the behaviour more consistent and less confusing to a novice, see Listing 9.8.

```
public class Die
{
...
  public Die()
  {
    faceValue = roll();
  }
  public void roll()
  {
    faceValue = (int)(Math.random() * MAX) + 1;
  }
  public int getFaceValue(){   ...   }
  public String toString(){   ...   }
}
```

Listing 9.8: The improved `Die`-class.

It is furthermore important to show that modelling objects in the problem domain, does not necessarily mean to copy the behaviour and characteristics of

91

the domain object. In a library system it would for example be reasonable for the borrower-object to be responsible for knowing any outstanding fees, despite the fact that this would never be the case in real life. We do not model the real world, we model a system solving a problem originated in the real world. Therefore we have to make an effort to aid novices in *separating the model from the modelled.*

9.3   Emphasize Client View

In small-scale examples, it is common with simple classes with only set- and get-methods handling the attributes of a class, see Listing 9.9.

```java
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;
  }
} //Card
```

Listing 9.9: Public access methods for attributes (Wick et al., 2004).

In this example it makes no sense declaring the attributes private, and it makes object orientation seemingly more complicated to use just because of the demand to keep attributes private. Wick et al. (2004) state that abstraction and encapsulation are the main principles of sound design, but this must not be contradicted by the example. *Object thinking* (West, 2004) helps us to think of objects from the problem domain and makes the argument for information hiding strong. Martin (2003) states that a model must be validated in connection to its clients. From an educational point of view it is therefore important to present a context for the small-scale examples. This has a pedagogical value in the sense that it will be possible to contrast different solutions with respect to their context. It will also show the need for comparing several solutions when deciding on an appropriate one for the problem at hand. This will illustrate that there is not a definite truth about 'right' and 'wrong'. It becomes important to emphasize the difference between attributes

and properties of a class, as well as giving a lot of attention to the interface/protocol of a class. Meyers (2004) concludes that the most important general interface design guideline is: "make interfaces easy to use correctly and hard to use incorrectly".

So what behaviour would be appropriate for the `Card` class? It is questionable if `Card`-objects ever should change state, this is not a reasonable behaviour from a clients point of view. There are 52 cards in a deck, and they never change suit or value. The interesting services is to be able to ask the object if it is of the same suit as another `Card`-object or whether it is higher/lower in value than another `Card`-object, and so on. If the context is to implement a simple poker card abstraction to be used in a computer administrated game, one suggestion could be Listing 9.10.

```java
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;
  }
  ...
}// Card
```

Listing 9.10: Examples of public access methods for responsibilities.

9.4   Favour Composition over Inheritance

Inheritance is is one of the most significant features of object orientation (Skrien, 2009). Properly used it increases the reusability of classes and minimizes the duplication of code. However, deriving a new class, from a class with only similar protocol is not good practice since it can cause unwanted consequences. This can make part of the inherited protocol inappropriate for the derived class. Implementing a stack through derivation from `Vector` is an example of this, see Listing 9.11.

```
public class Stack extends Vector
{
  public Stack() {}
  public Object push(Object item)
  {
    addElement(item);
    return item;
  }
    ...
}
```

Listing 9.11: `Stack` implemented with Inheritance (Kegel and Steimann, 2008).

The problem with this design is that there are a number of methods inherited from `Vector` (and its superclass `AbstractList`) that do not suit the behaviour of a stack, e.g. `insertElementAt` and `elementAt`. This means that `Stack` has a protocol that does not match the definition of a stack. This makes it possible for clients to access and manipulate `Stack`-objects in inappropriate ways; breaking the stack abstraction. By using composition and delegation instead of inheritance, the protocol of `Stack` specifies only legal requests and forwards them to the appropriate responsibilities of `Vector`, see Listing 9.12.

```
public class Stack
{
  protected Vector delegatee;
  public Stack()
  {
    delegatee = new Vector();
  }
  public Object push(Object item)
  {
    delegatee.addElement(item);
    return item;
  }
  public int size()
  {
    return delegatee.size();
  }
    ...
}
```

Listing 9.12: `Stack` implemented with Delegation (Kegel and Steimann, 2008).

There are a number of problem domains commonly used in introductory programming when introducing inheritance, like geometrical figures, genealogical relationships, and bank accounts. In the example of Figure 9.3, a class `Account` is responsible for managing the basics of a bank account: balance, owner and

creation date. To accommodate private customers a class `SavingsAccount` is derived from `Account` . It *is-an* `Account,` but also has the additional responsibilities connected to interest rates. When a new type of account is needed it is simple to derive a new class from `Account`.
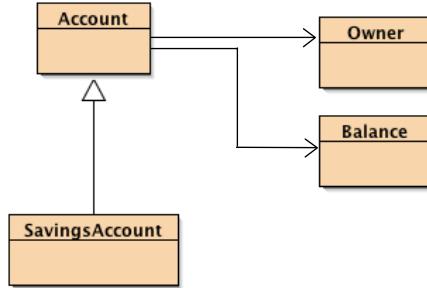
Figure 9.3: Hierarchy of accounts.

But in the small-scale context, this design might be too simple to be justifiable. If the class `SavingsAccount` adds restrictions to the behaviour of `Account`, e.g. restrictions on withdrawals, then this will be inconsistent with the expected behaviour of `Account` and *The Liskov Substitution Principle* is violated. In this case delegation is a better design. Using composition for this example, it would be a good idea to rename the `Account` class, calling it AccountI naming is too hasty and cause us to think about a design in fruitful way. The way we talk about objects is critical and we h about the vocabulary used (West, 2004) .

With the delegated design, an account of any kind can keep through the collaboration with an object designed for a limited responsibility, see Figure 9.4.
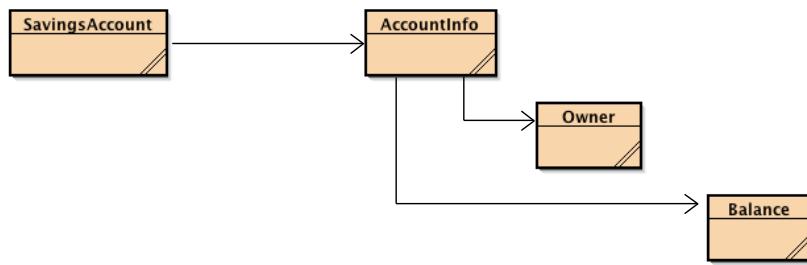
Figure 9.4: Composition of accounts.

So why would composition be preferable to inheritance? One reason is, that deriving from existing classes, not specifically designed for the intended hierarchy, is that parts of the inherited protocol might be inappropriate for the abstraction.

The general recommendation is to use delegation when the features of an existing class is wanted, but its protocol inappropriate for the new class. Then a client of the new class is offered services appropriate for the abstraction *(Eckel, 2002)*.

Another benefit from promoting delegation, is that composition is a strong argument for objects as autonomous and collaborating entities, and should be used to reinforce this object oriented characteristic.

How can we decide whether or not a small-scale example is suitable to illustrate inheritance? One recommendation is to think about the context/problem domain. Consider if is it going to be useful to upcast from the derived class to the base class? That is, is the particular example going to show the strength of late binding, or is about code sharing. Reuse of code is not by itself sufficient for inheritance.

For the small-scale situation it seem like a good recommendation to postpone the introduction of inheritance until it is possible to give an appropriate example.

9.5 Use Exemplary Objects Only

There are a number of details concerning small-scale examples to be aware of to prevent misconceptions or "bad" norm-building.

One common difficulty initially, is the confusion of class and object. To aid in differentiating between class and object, is to make sure that it is obvious that there are many objects instatiated from a single class. However, it can be difficult to *promote the idea of many objects*, because of the limited set of language constructs available. This can be done in different ways, e.g. using some kind of collection, depending on the order of concept introduction, but the important thing is to avoid single instances.

Even more problematic is the use of *one-of-a-kind classes*. Many textbooks contains examples that consist of single-object situations. One such example is the `RobberLanguageCryptographer`, see Listing 9.13.

```java
public class RobberLanguageCryptographer
{
  public RobberLanguageCryptographer()
  {
  }
  private 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)   {    ...    }
  ...
}
```

Listing 9.13: One-of-a-kind cryptographer.

In this example[2] the reason for more than one object might be difficult to understand, unless it is explicitly shown that different objects of the class 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 empty, and there are no attributes.

---

[2]Taken from a university website, given as solution for one of the assignments in a final CS1-exam.

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, 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, as in Listing 9.14.

```java
public class RobberLanguageCryptographer
{
  private char subst;
  public RobberLanguageCryptographer (char c)
  {
    if (!isConsonant(c))
      subst = c;
    else
      subst = 'o';
  }
  private boolean isConsonant(char c)   {    ...    }
  public String encrypt(String s)   {    ...    }
  public String decrypt(String s)   {    ...    }
  ...
}
```

Listing 9.14: Class instantiating several cryptographer-objects.

A similar one-of-a-kind issue can be raised with class `Random` (which is frequently usen in introductory textbooks) is shown i Listing 9.15.

```java
Random generator = new Random();
int i;

i = generator.nextInt();
System.out.println("A random int: " + i);
i = generator.nextInt(10);
System.out.println("An int in [0,9] : " + i);
```

Listing 9.15: Example of a class with no need for more than one object.

What is the need for the class `Random`? Is it ever necessary to have more than one object of this kind? There are reasons for having more than one random.number generator, e.g. to avoid having the same sequence of random numbers by using different seeds, but it is seldom productive to discuss this when an example is in need of a random number. Then the example is better replaced until the discussion on the generation of random numbers can be pursued. Another option is to use the static method `random` in `Math`, see Listing 9.16.

```
int i;
double d;

d = Math.random();
i = (int) (d*10);
System.out.println("A random double: " + d);
System.out.println("An int in[0,9]: " + i);
```

Listing 9.16: Avoiding one-of-a-kind objects introduces another problem.

This is probably a more adequate example, but uses a non-exemplary object oriented concept – a static method. This is confusing for novices since Math is the name of the class and no object is instantiated. In Java 1.5 this is however in some sense disguised by allowing import of the class and thereby making the call to static methods seem as the method is a part of general services built into the language. However, this might add to the confusion of a novice, since it is common to have difficulties separating what is actually part of the language and what is part of external libraries.

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 demosntrate how to use objects of the class. Listing 9.17 shows a common example of this.

```
public class Person
{
  private ...;
  ...
  public Person(...)   {   ...   }//constructor
  public void someMethod(...)  {   ...   }

  public static void main(string[] args)
  {
    Person p = new Person(...);
    p.someMethod(...);
  }
} //class Person
```

Listing 9.17: Attempt to avoid extra classes.

In this example, 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 instantiating the objects discussed. *Isolate main* to keep the boundaries of objects clear, see Listing 9.18.

```
public class Test
{
  public static void main(string[] args)
  {
    Person mother = new Person(...);
    mother.someMethod(...);
    Person father = new Person(...);
    father.someMethod(...);
  }
} //class Test
```

Listing 9.18: Keep clients in separate classes.

This example still has to be put into context, because the class `Test` is not a real client of objects of class `Person`, merely substituting a client temporarily for testing purposes. The problematic use of main can be avoided by using an IDE that supports working with objects in isolation, i.e. instantiating objects and invocating methos without explicit testprograms, e.g. BlueJ (BlueJ).

9.6 Make Inheritance Reflect Structural Relationships

It is challenging to find small-scale examples that illustrates inheritance, because of the many demands on inheritance (Chapter 4). A common small-scale example for inheritance is the design of the geometrical shapes rectangles and squares. The rectangle is defined as shown in Listing 9.19

```
public class Rectangle
{
  private double height, width;
  public void setHeight(double h)
  {
    height=h;
  }
  public void setWidth(double w)
  {
    width=w;
  }
}//Rectangle
```

Listing 9.19: A `Rectangle` class.

From a mathematical point of view it might be possible to say that a square is 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 Figure 9.5 and Listing 4.2.
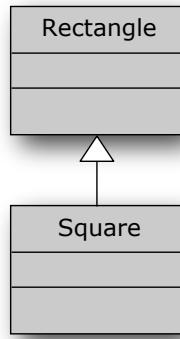
Figure 9.5: Square seen as a more specialised version of a rectangle.

Looking more closely at this example, it is not unreasonable to think that the designer of the method `setWidth`, assumed that setting the width of a rectangle leaves the height unaltered. This is not true for the behaviour of a `Square`, and the design is violating *The Liskov Substitution Principle* as discussed in Section 4.4. This principle states that descendants must not violate assumed characteristics of the base class and that anything expected from the base class must be true for the derived class. Adhering to the design advice to "only derive a class from an abstract class" 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. In (Martin, 2003) this is formulated as *Derived methods should expect no more and provide no less than its inherited method.* 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. As Martin (2003) states it: *Behaviorally, a Square is not a Rectangle! And it is behavior that software is really all about.* Booch (1994) formulates a description of inheritance (class structure) based on the definition of hierarchy: *Hierarchy is a ranking or ordering of abstractions.* Furthermore, it might be argued that mathematically squares and rectangles do not change size, they are "constants", and therefore no really good examples. An extensive discussion of this particular example is given in (Skrien, 2009).

If there is no structural relationship between `Square` and `Rectangle`, the recommendation is to use compomposition instead. Defining a class `QuadFig` that has the responsibility to know its two sides, and to able to evaluate its area and to answer whether it is a square or not, makes this example more general, and the common characteristics of the two shapes are in focus (Listing 9.20).

```
public class QuadFig
{
  private double side1, side2;

  public QuadFig(double s1, double s2)
  {
    side1 = s1;
    side2 = s2;
  }
  public double getSide1(){   ...   }
  public double getSide2(){   ...   }

  public double getArea()
  {
    return side1*side2;
  }
  public boolean isSquare(){   ...   }
}
```

Listing 9.20: Squares and rectangles are all quadrangles.

This example is no longer illustrating inheritance, but it might be used in connection with the design in Figure9.5 and Listing **??** to discuss what inheritance is and how and when it should be used (and not).

Another issue is that *inheritance should separate behaviour* (Riel, 1996), which can be difficult within the small-scale context. If the behaviour of a class depends on its state, and the state is tested explicitly to decide on the bahaviour, this can be a proper situation to use inheritance. The class `Ball` in Figure 9.6 has a behaviour that depends on the value of its attribute colour. The context could be that these balls are part of some kind of game, where blue balls explode, green balls jump and red balls are eaten.
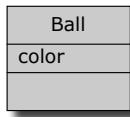

Figure 9.6: Major difference in behaviour depending on the value of an attribute (Riel, 1996).

In this context, `Ball` can have the responsibility of knowing its colour and performing an action, `act`, and this behaviour is then defined by the derived classes (Figure 9.7).

Declaring the class `Ball` and the method `act()` abstract, forces the novice to realise the need to supply implementations for the behaviour in the derived classes. A slighly larger example is given in Figure 9.8. In this example a "Database of Multimedia Entertainment, DoME" is being designed.

In this design, the attributes and behaviour declared in the base-class `Item`, is general and concerns the entries in the database. The derived classes are suffi-

102
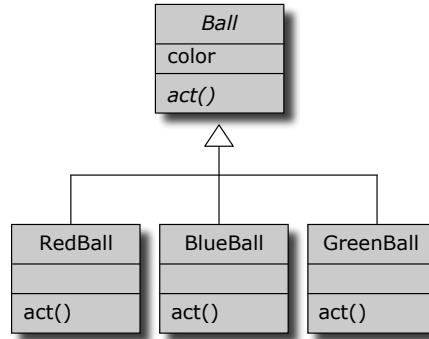
Figure 9.7: Inheritance used to separate behaviour among siblings. Slightly adapted from(Riel, 1996).
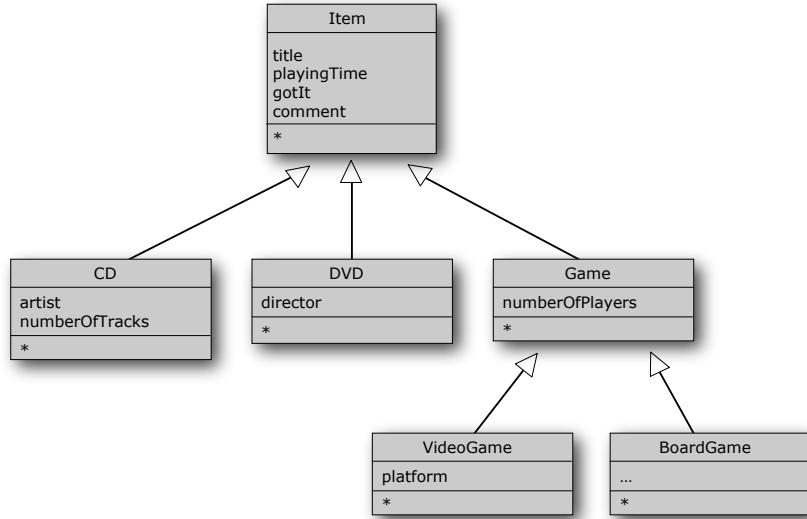
Figure 9.8: The DoME hierarchy. (Barnes and Kölling, 2009)

ciently different, compared to each other, to justify the added responsibilities and behaviour.

The Dependency-Inversion Principle is critical to inheritance, and some practical advice for upholding it is given in (Martin, 2003):

- No variable should hold a reference to a concrete class.

- No class should derive from a concrete class.

- No method should override an implemented method of its base class.

Objects need to be instantiated eventually, but these advice will support the educator designing small-scale examples on inheritance, and the novices' understanding and use of inheritance.

9.7 Summary

The proposed He[d]uristics and some of their implications are summarized below.

1. *Model Reasonable Abstractions*
Plausible both from a software perspective and also from a novice perspective.
Do not make main into the entire program.
Real objects (with identity, state and behaviour), implies, e.g., no stateless or behaviourless classes (containers).
Small is beautiful (in terms of classes, methods and parameters) implies, e.g., no God classes.
Model classes not roles.

2. *Model Reasonable Behaviour*
Separate the model from the modelled.
Avoid setter/getters, particularly for attributes.
No snippets.
No printing for tracing.

3. *Emphasize Client View*
Promote thinking in outside expectations.
Separate the internal representation from the external functionality.

4. *Favour Composition over Inheritance*
How to know which to choose, to avoid the overuse of inheritance.
Emphasize the idea of collaborating objects.
Delay the introduction of inheritance.

5. *Use Exemplary Objects Only*
Always promote the idea of many objects.
No one-of–a-kind classes.
Be explicit, do not use static, anonymous classes, and keep the Law of Demeter in mind.
Separate `main` from abstractions.

6. *Make Inheritance Reflect Structural Relationships*
Important to distinguish between external and internal is-a relationships.
Behaviour must guide the design of hierarchies.
Inheritance should separate behaviour.

Some of these characteristics/qualities could be placed under more than one heading.

The use of real objects (with identity, state and behaviour), is on one hand a question of abstraction, and on the other the promotion of exemplary objects. In this work *Use Exemplary Objects Only* is more intended to focus on the actual presentation of small-scale examples while *Model Reasonable Abstractions* is aiming at a proper mindset for object oriented design.

Chapter 10

Validation

10.1 Introduction

The aim of this work is to propose a number of He[d]uristics that will aid in upholding object oriented characteristics when designing small-scale examples. They should be few enough to keep in mind without effort, but comprehensive enough to cover basic object oriented characteristics.

In the following sections we evaluate the He[d]uristics presented in Chapter 8 and 9against the advice and concepts discussed in Chapters 2–7.

In Section 10.2, we discuss the collections of advice[1] surveyed (Chapter 4–6) and how the proposed He[d]uristics upholds them.

In Section 10.3, we define a set of object oriented concepts collected from the findings of Chapters 2–3, and then we evaluate the He[d]uristics with respect to this set.

Finally in Section 10.4, we investigate whether or not the He[d]uristics can contribute to the surveyed cognitive difficulties discussed in Sections 2.4–2.6.

10.2 He[d]uristics vs. Advice

The collections of advice from in Chapters 4 - 6 are on different levels of detail. In many cases they overlap and address similar features of object orientation. A classification the collections of advice in terms of level of detail is shown in Figure 10.1.

---

[1]The word "advice" is used to represent any of the principles, patterns, rules, heuristics and guidelines presented in chapters 4-6.
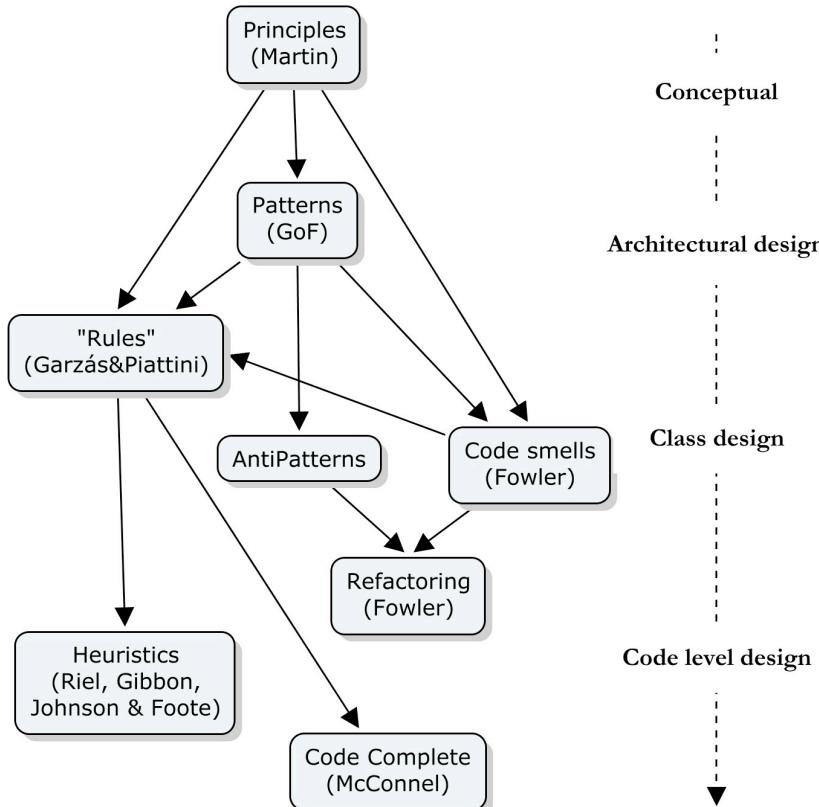
Figure 10.1: Classification of advice

Patterns and *Principles* are general advice on how to design reusable and maintainable object oriented units. *Rules*, *AntiPatterns* and *Code Smells* are all designed for existing code and attempt to identify problematic situations and constructions, and make suggestions how to deal with them. The different *Heuristics* (Riel, Gibbon, Johnson and Foote, and McConnel) are detailed implementational advice. The number of advice in a single collection rises, as the level of detail increases.

Typical examples of advice for each of the levels of detail is presented below.

Conceptual: *The single responsibility principle* (Martin, 2003), see Section 4.2. Each responsibility should be a separate class. A class should have one, and only one, reason to change. This means keeping the class small and focused. The interface should be simple and clearly state the services provided by the class.

Architectural design: *A class collaborates with many others* (Garzás and Piattini, 2007b), see Section 5.7 and Table D.3. Extensive collaboration may be an indication of too many responsibilities.

Class design: *Large class smell* (Fowler et al., 1999), see Section 6.6. A large class

is an indication of a class trying to do too much, and possible indications of this are many attributes and/or methods.

Code design: *Limit the number of attributes per class* (Gibbon and Higgins, 1996)*,* see Section 5.4 . Some advice in this category is even more detailed and explicitly states the number of parameters, lines of code, methods in a class etc.

The categories are not sharp, e.g. *Class design* has more to do with implementation/code, while *Architectural design* is more on design issues.

It is crucial to uphold as many of the general principles and heuristics as possible, since teaching "bad" strategies that have to be unlearned later, must be considered to be a less pedagogical path. The problem is to avoid burdening novices with too many explicitly stated general rules.

In general, all advice in Chapters 4 - 6 can be captured by the high-level conceptual principles of Chapter 4. Apart from the principles of Chapter 4, one of the two main principles of patterns from Section 6.2 have been included in this evaluation, *Favour object composition over class inheritance.* The other basic idea of patterns, *Program to an interface, not an implementation* is similar to *The Dependency Inversion Principle* (DIP).

- SRP – The Single Responsibility Principle

- OCP – The Open Closed Principle

- LSP – The Liskov Substitution Principle

- DIP – The Dependency Inversion Principle

- ISP – The Interface Segregation Principle

- LoD – Law of Demeter

- Favour object composition over class inheritance.

The granularity of Principles and He[d]uristics is different, and the evaluation is therefore an interpretation of whether a suggested He[d]uristics supports a particular principle or not.

The results of the evaluation in Table 10.1 shows that all of the principles are covered to some extent.

Table 10.1: Relationship between Principles and He[d]uristics.

Principle	He[d]uristic					
	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
SRP – The Single Responsibility Principle	X	X	X		X	
OCP – The Open Closed Principle						X
LSP – The Liskov Substitution Principle	X					X
DIP – The Dependency Inversion Principle		X				X
ISP – The Interface Segregation Principle	X	X	X			
LoD – Law of Demeter		X				
Favour object composition over class inheritance				X	X	

The He[d]uristics are only slightly addressing the principles *OCP* and *LoD*. This could be expected, since the He[d]uristics are focusing on educational strategies, and do not specifically address these kinds of principles. *The Open Closed Principle* has to be enforced primarily through proper use of inheritance. *The Law of Demeter* can be upheld by careful application of the He[d]uristic *Use Exemplary Objects Only.*

10.3   He[d]uristics vs. Concepts

There is no commonly agreed upon set of concepts characterising object orientation and there is no agreed upon vocabulary. Sometimes different words are used to represent the same conceptual idea. Looking for object oriented concepts in all kinds of advice (Chapters 4 - 6) and related work (Chapters 2 and 3), we selected the concepts shown in Figure 10.2 as the minimal set of most frequent and emphasized concepts in the literature.

There are further concepts frequently mentioned in the literature that have been excluded from this selection.

The concepts *Method* and *Message*, is covered by the concept *Communication*. We have made this choice because they are closely related and the words are used to indicate communication, even though message is on a more conceptual level than method. In the small-scale context it is initially less crucial to make this
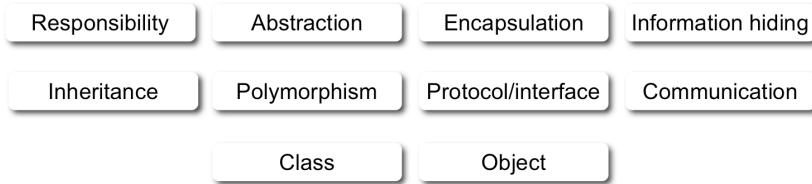
Figure 10.2: Minimal set of object oriented concepts

distinction.

Coupling and *Cohesion* are to a large extent dependent on both the abstraction and on the design of the class and are therefore not explicitly included in the set of concepts, since we consider them covered by the advice presented in Section 8.2.

Reuse is made possible through a number of practices in object orientation; e.g. carefully designed abstractions, collaboration among classes, composition and proper use of inheritance.

Table 10.2 summarizes how the essential object oriented concepts are covered by our He[d]uristics.

Table 10.2: Relationship between Concepts and He[d]uristics

He[d]uristic

Concept	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
Responsibility	X	X		X	X	
Abstraction	X		X		X	X
Encapsulation	X	X		X	X	X
Information hiding			X		X	
Inheritance	X					X
Polymorphism						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

In general, the concepts of Figure 10.2 are well covered by the He[d]uristics.

Polymorphism is the concept least explicitly focused. The only He[d]uristic that applies explicitly is *Make Inheritance Reflect Structural Relationships.* On the other hand is polymorphism covered by several of the principles in Section 10.2, e.g. *The Liskov Substitution Principle* and *The Dependency Inversion Principle.*

10.4 Addressing Misconceptions and Difficulties

A number of studies and attempts to classify different problems and common misunderstandings among object oriented novices have been reviewed in Section 2.4 - 2.6. They are important indicators of conceptual difficulties. To assist educators in upholding object oriented qualities in small-scale examples, it is important to see whether or not common conceptual difficulties can be addressed using the proposed He[d]uristics. Therefore, the proposed He[d]uristics are evaluated against a number of common difficulties.

The learning difficulties investigated in relation to the proposed He[d]uristics, are listed below.

Misconceptions (Section 2.5)

1. Avoiding object/variable conflation.
2. Objects are not simple records
3. Work in methods is not all done by assignment
4. Object/class conflation
5. Identity/attribute confusion.
6. Conflation of textual representation of objects and references

Difficulties (Section 2.5)

1. Object state
2. Method invocation
3. Parameters
4. Return values
5. Input instructions
6. Constructors
7. The overall picture of execution

Harmful Examples (Section 2.6)

1. Examples that are too abstract
2. Examples that are too complex
3. Concepts applied inconsistently
4. Examples undermining the concept introduced

Table 10.3: Coverage of Misconceptions in He[d]uristics

He[d]uristic

Misconceptions, Difficulties and Harmful examples	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
M - Avoiding object/variable conflation.				X		
M - Objects are not simple records	X	X	X	X	X	X
M - Work in methods is not all done by assignment	X	X	X		X	
M - Object/class conflation	X	X	X	X	X	
M - Identity/attribute confusion.	X				X	
M - Conflation of textual representation of objects and references					X	
Diff 1 – Object state	X	X			X	
Diff 2 – Method invocation		X	X			
Diff 3 – Parameters						
Diff 4 – Return values		X			X	
Diff 5 – Input instructions						
Diff 6 – Constructors	X					
Diff 7 – The overall picture of execution						
Harmful - Examples that are too abstract	X		X			
Harmful - Examples that are too complex			X			
Harmful - Concepts applied inconsistently						
Harmful - Examples undermining the concept introduced	X	X		X	X	X

The investigation of to what extent the proposed He[d]uristics can be of assistance in avoiding common difficulties is presented in Table 10.3.

In general, the proposed He[d]uristics focus on object oriented aspects of examples, even if *Favour Composition over Inheritance* and *Use Exemplary Objects Only* are emphasizing the need to be particular about details to avoid being counterproductive.

Some of these difficulties is not covered by the proposed He[d]uristics.

Diff 3-Parameters and *Diff 5-Input instructions* are general programing difficulties and does not concern object orientation in particular.

Diff 7-The overall picture of execution can not be resolved by better examples,

it requires a conceptual model of execution.

Harmful-Concepts applied inconsistently general problem, not particular for object orientation, but our He[d]uristics help to provide a more consistent picture of object orientation.

Slightly different problems are addressed by *Principles for teaching novices* (Section 2.4) and *Student constructed rules* (Section 2.5).

Principles for teaching novices (Section 2.4)

1. Reveal the programming process, in order to ease and promote the learning of programming.
2. Teach skills, and not just knowledge, in order to promote the learning of programming.
3. Present concepts at the appropriate level of abstraction.
4. Order material so as to minimize the introduction of terms or topics without explanation
5. Use unambiguous, clear, and precise terminology

Student constructed rules (Section 2.5)

1. The Java compiler can distinguish between same-named methods only if they have differences in their parameter lists.
2. The only purpose of invoking a constructor is to initialize the instance variables of an object
3. Numbers or numeric constants are the only appropriate actual parameters corresponding to integer formal parameters.
4. The dot operator can only be applied to methods.

The Novice-teaching principles are general advice for instructional design rather than addressing specific object oriented qualities of the examples, and they have no correspondence in the proposed He[d]uristics.

The Student Constructed Rules illustrate the need to anticipate and discuss common misunderstandings/misconceptions when introducing a concept or the semantical implications of syntactical element. These problems are not addressed by the proposed He[d]uristics.

Chapter 11

Conclusions and Future Work

There seems to be a trade-off between object oriented principles and 'good' examples when it comes to the specific needs of an educational situation. Small examples are burdened by large overhead in terms of justifying exemplary objects and their behaviour. Size is definitely important! If novices are to comprehend an example easily, it must be restricted in size, both in terms of lines of code, but also in terms of complexity. Trying to apply established object oriented principles to examples and exercises in small-scale examples might seem difficult at first. There will be causes for violation of even basic principles. However, having them in mind, interpreted in the educational situation, should make us think twice and hopefully avoid some of the pitfalls.

The level of abstraction in object orientation adds difficulty to the effort of both the mediator of knowledge and the novice, compared to learning problem solving and programming through the imperative paradigm.

The number of He[d]uristics has been difficult to decide upon. Too few would probably mean that they would be on such a general level that they would be hard to use, and to many would also be impractical.

It has been astonishing to realise how immature object orientation is, in terms of agreed upon terminology and definitions of characteristics, even within the educational community.

Often small-scale examples are given too little attention, sometimes as a result of too superficial understanding of how details may contradict the ideas being conveyed and perhaps sometimes out of ignorance.

To evaluate and further develop the suggested He[d]uristics we are planning some empirical studies.

- Surveys and interviews with educators, both at universities and in upper secondary schools, to investigate their descriptions of what they consider most critical in object orientation and how they teach it.

- Testing the proposed He[d]uristics on groups of educators to get feedback on usability.

Based on these studies we revise the proposed He[d]uristics to reflect the experiences resulting from the analysis of the the empirical material.

Other interesting research questions:

- Is it possible to formulate metrics for small-scale situations? How can these metrics be evaluated? If there is such a metric it must be based on qualities of both object oriented principles and educational values. One way of collecting more educational background material is to find out how lecturers value the difficulties of object orientation and how they chose to address them. This would help determine common problems and common choices.

- How would the suggested He[d]uristics influence the example evaluating tool discussed in Börstler et al. (2008)?

Bibliography

Example of McCabes Cyclomatic Complexity. `http://www.vidbob.com/qa-info/control-flow-graphing.html`, Webpage last visited 2008-11-19.

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

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

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

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

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

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

Barnes, D. J. and Kölling, M. (2009). *Objects First with Java: A Practical Introduction Using BlueJ: International Edition, 4/E*. Pearson Higher Education, 4th edition.

Bashar Molla, M. K. (2005). An overview of object oriented design heuristics. Master's thesis, Department of Computer Science, Umeå University, Sweden.

Beck, K. and Cunningham, W. (1989). A laboratory for teaching object oriented thinking. In *Conference proceedings on Object-oriented programming systems, languages and applications, OOPSLA'89*, pages 1–6, New Orleans, Louisiana, United States. ACM.

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

Bennedsen, J. and Caspersen, M. E. (2004). Programming in context – a model-first approach to cs1. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education,*, pages 226–230, St. Louis, Missouri, USA. ACM.

Bennedsen, J. and Caspersen, M. E. (2006). Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bull.*, 38(2):39–43.

Bennedsen, J. and Caspersen, M. E. (2008). *Model-Driven Programming*, pages 116–129. Springer-Verlag, Berlin, Heidelberg.

Bergin, J. (2007). Building graphical user interfaces with the mvc pattern. `http://csis.pace.edu/~bergin/mvc/mvcgui.html`, Webpage Last visited 2009-01-02.

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 2008-06-25.

Bock, D. (2008). The paperboy, the wallet, and the law of demeter. `http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf` Webpage Last visited 2008-12-28.

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

Börstler, J. (2004). Object-oriented analysis and design through scenario role-play. Technical report, Dept. of Computing Science, Umeå University, Umeå, Sweden.

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., 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-108.09, Dept. of Computing Science, Umeå University, Umeå, Sweden.

Boustedt, J., Eckerdal, A., McCartney, R., Moströmk, J.-E., Ratcliffe, M., Sanders, K., and Zander, C. (2007). Threshold concepts in computer science: do they exist and are they useful? *SIGCSE Bull.*, 39(1):504–508.

Brown, W. J., Malveau, R., McCormick, H., and Mowbray, T. (1998). *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley.

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. (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.

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.

Christensen, H. B. (2005). Implications of perspective in teaching objects first and object design. In *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education*, pages 94–98.

Christensen, K., Fitsos, G. P., and Smith, C. P. (1981). A perspective on software science. *IBM Systems Journal*, 20(4).

Clark, R., Nguyen, F., and Sweller, J. (2006). *Efficiency in Learning, Evidence-Based Guidelines to Manage Cognitive Load*. Wiley & Sons.

Cooper, J. W. (2000). *Java Design Patterns – A Tutorial*.

DemeterW3. Demeter webpage. `http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/object-formulation.html`, Webpage last visited 2008-07-30.

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

Dijkstra, E. W. Definition of abstraction (personal conversation with David Parnas).

Dijkstra, E. W. (1983). The fruits of misunderstanding. Unpublished work, transcript at `http://www.acm.org/education/curricula/ComputerScienceCurriculumUpdate2008.pdf`.

Dijkstra, E. W. (1999). Computing science: Achievements and challenges. Unpublished work, transcript at `http://www.cs.utexas.edu/users/EWD/ewd12xx/EWD1284.PDF`.

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.

Eckel, B. (2002). *Thinking in Java, 3rd ed.* Prentice Hall Professional Technical Reference.

Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K., and Zander, C. (2006). Putting threshold concepts into context in computer science education. *SIGCSE Bull.*, 38(3):103–107.

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.

Fleury, A. E. (2001). Encapsulation and reuse as viewed by java students. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 189–193.

Fowler, M. (2003). Patterns. *IEEE Software*, 20(2):56–57.

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. (2007a). Improving the teaching of object-oriented design knowledge. *SIGCSE Bull.*, 39(4):108–112.

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

Gibbon, C. (1997). *Heuristics for Object-Oriented Design.* PhD thesis, University of Nottingham.

Gibbon, C. A. Object-oriented design heuristics: a working document, internal report (missing thesis ref.). Personal com. July 2006 with Dr. Gibbon and prof. Higgins concerning missing Thesis ref.

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.

Gil, J. 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. ACM.

Goldman, K. J. (2004). A concepts-first introduction to computer science. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 432–436, New York, NY, USA. ACM.

Gries, D. (2007). Teaching java –with oo first. Key Note at PPPJ 2007. `http://www.cs.cornell.edu/gries/programlive/OOfirst.pdf`, Webpage Last visited 2008-12-02.

Gries, D. (2008). Teaching oo to beginners. key note at oopsla 2008. `http://www.cs.cornell.edu/gries/programlive/oopsla.key`, Webpage Last visited 2008-12-02.

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

Gupta, S. (2008). Designing abstraction. `http://javaboutique.internet.com/tutorials/JavaOO/OCP.html`, Webpage last visited 2008-12-19.

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.

Hewson, P. W. (1981). A conceptual change approach to learning science. *International Journal of Science Education*, 3(4):383–396.

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.

Horstmann, C. S. (2004). *Object-oriented design & patterns.* Wiley New York.

Hunt, A. Dont repeat yourself (-the wikiwikiweb). `http://c2.com/cgi/wiki?WikiWikiWeb/`, Webpage Last visited 2009-01-11.

Hvam, L., Riis, J., and Hansen, B. L. (2003). Crc cards for product modelling. *Computers in Industry*, 50(1):57–70.

JHawk Webpage. Jhawk java code metrics. `http://www.virtualmachinery.com/jhawkmetrics.htm`, Webpage Last visited 2009-01-11.

Jimenez, E. (2006). Antipatterns. `http://www.antipatterns.com/EdJs_Paper/Antipatterns.html`, Webpage last visited 2008-10-17.

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

JPie. JPie webpage. `http://jpie.cse.wustl.edu/`, Webpage last visited 2008-12-12.

Karahasanovic, A., Levine, A. K., and Thomas, R. (2007). Comprehension strategies and difficulties in maintaining object-oriented systems: An explorative stud. *Journal of Systems and Software, Evaluation and Assessment in Software Engineering - EASE06*, 80(9):1541–1559.

Kay, A. (1990). User interface: A personal view. In Laurel, B. and Mountford, S., editors, *The Art of Human-Computer Interface Design*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Kay, A. C. (1996). The early history of smalltalk. pages 511–598.

Kegel, H. and Steimann, F. (2008). Systematically refactoring inheritance to delegation in java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 431–440, New York, NY, USA. ACM.

Kotzé, P., Renaud, K., and van Biljon, J. (2008). Don't do this - pitfalls in using anti-patterns in teaching human-computer interaction principles. *Comput. Educ.*, 50(3):979–1008.

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.

Lewis, J. and Loftus, W. (2007). *Java Software Solutions*. Addison-Wesley, 5th edition.

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., 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.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR.

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

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.

Mertz, A., Slough, W., and Cleave, N. V. (2008). Using the acm java libraries in cs 1. *J. Comput. Small Coll.*, 24(1):16–26.

Meyer, B. (1997). *Object-oriented Software Construction 2/E.* Prentice Hall.

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.

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

Mosley, P. (2005). A taxonomy for learning object technology. *J. Comput. Small Coll.*, 20(3):204–216.

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.

Or-Bach, R. and Lavy, I. (2004). Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bull.*, 36(2):82–86.

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. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

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

Piaget, J. (1962). The stages of the intellectual development of the child. *Bulletin of the Menninger Clinic*, 26(May):120–128.

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.

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

121

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.

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

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

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-oriented modeling and design*. Prentice-Hall, Inc.

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

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.

The Free Dictionary (2008). `http://www.thefreedictionary.com/heuristic`, Webpage last visited: 2008-11-21.

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

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

White, G. and Sivitanides, M. (2005). Cognitive differences between procedural programming and object oriented programming. *Inf. Technol. and Management*, 6(4):333–350.

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.

Wirth, N. (2002). Computing science education: the road not taken. In *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education*, pages 1–3, New York, NY, USA. ACM.

List of Figures

List of Tables

Listing

Appendix A

Riel's heuristics

Classes and Objects: The Building Blocks of the Object Oriented Paradigm

02.01 All data should be hidden within its class

02.02. Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

02.03. Minimize the number of messages in the protocol of a class (protocol of a class means the set of messages to which an instance of the class can respond)

02.04. Implement a minimal public interface that all classes understand

02.05. Do not put implementation details such as common-code private functions into the public interface of a class

02.06. Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using.

02.07. Classes should only exhibit nil or export coupling with other classes, that is, a class should only use operations in the public interface of another class or have nothing to do with that class.

02.08. A class should capture one and only one key abstraction

02.09. Keep related data and behaviour in one place.

02.10. Spin off non-related information into another class (that is, non-communicating behaviour)

02.11. Be sure the abstractions that you model are classes and not simply the roles objects play

Topologies of Action-Oriented Versus Object-Oriented Applications

03.01 Distribute system intelligence horizontally as uniformly as possible, that is the top level classes in a design should share the work uniformly.

03.02 Do not create god classes or god objects in your system. Be very suspicious of a class whose name contains DRIVER, MANGER, SYSTEM, SUBSYS-TEM, etc.

03.03 Beware of classes that have many accessor methods defined in their interface. Having many implies that related data and behaviour are not being kept in one place.

03.04 Beware of classes that have too much non-communicating behaviour, that is, methods that operate on a proper subset of the data members of a class. God classes often exhibit a great deal of non-communicating behaviour.

03.05 In applications that consist of an object oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

03.06 Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behaviour in one place.)

03.07 Eliminate irrelevant classes from your design.

03.08 Eliminate classes that are outside the system.

03.09 Do not turn an operation into a class. Be suspicious of any class whose name is a verb or is derived from a verb, especially those which have only one piece of meaningful behaviour. Ask if that piece of meaningful behaviour needs to be migrated to some existing or undiscovered class.

03.10 Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Relationships Between Classes and Objects

04.01. Minimize the number of classes with which another class collaborates.

04.02 Minimize the number of message sends between a class and its collaborator.

04.03 Minimize the amount of collaboration between a class and its collaborator, that is, the number of different messages sent

04.04 Minimize fanout in a class, that is the product of the number of messages defined by the class and the messages they send

04.05 If a class contains objects of another class, then the containing class should be sending messages to the contained objects, that is, the containment relationship should always imply a <uses> relationship.

04.06 Most of the methods defined in a class should be using most of the data members most of the time.

04.07 Classes should not contain more objects than a developer can fit in his or her short-term memory. A favourite value for this number is six.

04.08 Distribute system intelligence vertically down narrow and deep containment hierarchies

04.09 When implementing semantic constraints, it is best to implement them in terms of the class definition. Often this will lead to a proliferation of classes, in which case, the constraint must be implemented in the behaviour of the class - usually but not necessarily, in the constructor.

04.10 When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down in a containment hierarchy as the domain allows.

04.11 The semantic information on which a constraint is based is best placed in a central, third party object when that information is volatile.

04.12 The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.

04.13 A class must know what it contains, but should never know who contains it.

04.14 Objects that share lexical scope - those contained in the same containing class - should not have <uses> relationship between them.

Inheritance Relationship

05.01. Inheritance should be used only to model a specialization hierarchy

05.02 Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes

05.03 All data in a base class should be private; do not use protected data.

05.04 In theory, inheritance hierarchies should be deep - the deeper the better

05.05 In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short term memory. A popular value for this is six.

05.06 All abstract classes must be base classes.

05.07 All base classes must be abstract classes.

05.08 Factor the commonality of data, behaviour, and/or interface, as high as possible in the inheritance hierarchy.

05.09 If two or more classes share only common data (no common behaviour), then that common data should be placed in a class that will be contained by each sharing class

05.10 If two or more classes have common data and behaviour (that is, methods), then those classes should each inherit from a common base class that captures those data and methods.

05.11 If two or more classes only share common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.

05.12 Explicit case analysis on the type of an object is usually an error, the designer should use polymorphism in most of these cases.

05.13 Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy, where each value of the attribute is transformed into a derived class.

05.14 Do not model the dynamic semantics of a class through the use of an inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at run time.

05.15 Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

05.16 If you think you need to create new classes at run time, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a new class.

05.17 It should be illegal for a derived class to override a base class method with a NOP method, that is, a method that does nothing.

05.18 Do not confuse optional containment with the need for inheritance. Modeling optional containment with inheritance will lead to a proliferation of classes.

05.19 When building an inheritance hierarchy, try to construct reusable frameworks rather than reusable components.

Multiple Inheritance

06.01 If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise.

06.02 When ever there is inheritance in an object oriented design ask yourself two questions: 1) Am I a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of me?

06.03 Whenever you have found a multiple inheritance relationship in a object oriented design be sure that no base class is actually a derived class of another base class, i.e. accidental multiple inheritance.

Association Relationship

07.01 When given a choice in an object oriented design between a containment relationship and an association relationship, choose the containment relationship.

Class-Specific Data and Behaviour

08.01 Do not use global data or functions to perform bookkeeping information on the objects of a class, class variables or methods should be used instead.

Physical object oriented Design

09.01 Object oriented designers should never allow physical design criteria to corrupt their logical designs. However, very often physical design criteria's used in the decision making process at logical design time.

09.02 Do not change the state of an object without going through its public interface.

Appendix B

Gang of four patterns

Creational Patterns

Factory Pattern Returns an instance of one of several similar classes depending on the data provided to it. Uses inheritance to achieve this.

Abstract Factory Pattern One level of abstraction higher than the factory pattern. Used to return one of several related classes of objects, each of which can return several different objects on request. The Abstract Factory is a factory object that returns one of several factories.

Singleton Pattern Ensures that there can be one and only one instance of a class. Usually solved by embedding a static variable inside the class that is set on the first instance and checked for each time the constructor is entered.

Builder Pattern Separates the construction of a complex object from its representation, so that several different representations can be created depending on the needs of the program.

Prototype Pattern Starts with an initialised and instantiated class and copies or clones it to make new instances rather than creating new instances. Particularly useful when creating new instances are expensive.

Structural Patterns

Adapter Pattern used to change the interface of one class to that of another one.

Bridge Pattern Intended to keep the interface to your client program constant while allowing you to change the actual kind of class you display or use. You can then change the interface and the underlying class separately.

Composite Pattern Components that are individual objects and also can be collection of objects. A recursive definition used e.g. in Java's graphical components.

Decorator Pattern A class that surrounds a given class, adds new capabilities to it, and passes all the unchanged methods to the underlying class.

Facade Pattern groups a complex object hierarchy and provides a new, simpler interface to access those data.

Flyweight Pattern provides a way to limit the proliferation of small, similar class instances by moving some of the class data outside the class and passing it in during various execution methods

Proxy Pattern provides a simple place-holder class for a more complex class which is expensive to instantiate.

Behavioural Patterns

Chain of Responsibility Pattern allows an even further decoupling between classes, by passing a request between classes until it is recognized.

Command Pattern provides a simple way to separate execution of a command from the interface environment that produced it, and

Interpreter Pattern provides a definition of how to include language elements in a program.

Iterator Pattern ormalizes the way we move through a list of data within a class.

Mediator Pattern defines how communication between classes can be simplified by using another class to keep all classes from having to know about each other.

Memento Pattern Without violating encapsulation, capture and externalise an object's internal state so that the object can be restored to this state later.

Observer Pattern defines the way a number of classes can be notified of a change.

State Pattern provides a memory for a class's instance variables.

Strategy Pattern used to encapsulate algorithms to be used by a context. The client needs to be aware of the different strategies.

Template Pattern provides an abstract definition of an algorithm.

Visitor Pattern adds function to a class.

Appendix C

Smells and associated refactorings

Associated with each code smell presented in (Fowler et al., 1999) is given a set of refactoring element to aid in the process of reorganising the code in a systematic way. (from (Fowler et al., 1999), back cover)

Smells	Common Refactorings
Alternative Classes with Different Interfaces	Rename Method, Move Method
Comments	Extract Method, Introduce Assertion ;
Data Class	Move Method, Encapsulate Field, Encapsulate Collection
Data Clump	Extract Class, Introduce Parameter Object, Preserve Whole Object
Divergent Change	Extract Class
Duplicated Code	Extract Method, Extract Class, Pull Up Method, Form Template Method
Feature Envy	Move Method, Move Field, Extract Method
Inappropriate Intimacy	Move Method, Move Field, Change Bidirectional Association to Unidirectional, Replace Inheritance with Delegation, Hide Delegate

Smells	Common Refactorings
Incomplete Library Class	Introduce Foreign Method, Introduce Local Extension Large Class Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Large Class	Extract Class, Extract Subclass, Extract Interface, Replace Data Value with Object
Lazy Class	Inline Class, Collapse Hierarchy
Long Method	Extract Method, Replace Temp with Query, Replace Method with Method Object, Decompose Conditional
Long Parameter List	Replace Parameter with Method, Introduce Parameter Object, Preserve Whole Object
Message Chains	Hide Delegate
Middle Man	Remove Middle Man, Inline Method, Replace Delegation with Inheritance
Parallel Inheritance Hierarchies	Move Method, Move Field
Primitive Obsession	Replace Data Value with Object, Extract Class, Introduce Parameter Object, Replace Array with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/ Strategy
Refused Bequest	Replace Inheritance with Delegation
Shotgun Surgery	Move Method, Move Field, Inline Class
Speculative Generality	Collapse Hierarchy, Inline Class, Remove Parameter, Rename Method
Switch Statements	Replace Conditional with Polymorphism, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Parameter with Explicit Methods, Introduce Null Object

Smells	Common Refactorings
Temporary Field	Extract Class, Introduce Null Object

Appendix D

Tables

Table D.1: Frequencies of object oriented concepts (Armstrong, 2006)

Concept	Count	Percentage
Inheritance	71	81%
Objects	69	78%
Class	62	71%
Encapsulation	55	63%
Method	50	57%
Message passing	49	56%
Polymorphism	47	53%
Abstraction	45	51%
Instantiation	31	35%
Attribute	29	33%
Information hiding	28	32%
Dynamic binding	13	15%
Relationship	12	14%
Interaction	10	12%
Class hierarchy	9	10%
Abstract data type	7	8%
Object-identity independence	6	7%
Collaboration	5	6%
Aggregation	4	5%
Object model	4	5%
Reuse	3	3%
Cohesion, Coupling, Graphical, Persistence	2	2%
Composition, Concurrency, Dynamic model, Extensibility, Framework, Genericity, Identifying objects, Modularization, Naturalness, Safe referencing, Typing, Virtual procedures, Visibility	1	1%

Table D.2: The MeTHOODS heuristics catalogue (Grotehen, 2001)

1	A class in a containment hierarchy should only depend on its child classes
2	Every attribute should be hidden within its class
3	Avoid dependencies of objectbase classes on their clients
4	A class should capture one, and only one key abstraction with all its information and its entire behavior
5	Do not create unnecessary classes to model roles
6	Avoid pure accessor operations.
7	Avoid additional relationships of base classes to their derived classes
8	Avoid classes with properties implying redundancies
9	Avoid multivalued dependencies
10	Whenever possible, convert associations and uses relationships in the strongest containment relationship.
11	Avoid contained objects that can concurrently be modified.
12	All properties of the basetype must be usable in objects of its subtypes in every location in that a basetype object is expected
13	Common properties of objects should be defined in a single location
14	Soft classes should not be base classes
15	Do not misuse inheritance for sharing attributes
16	The overloading should only define differences to the overloaded operation
17	Avoid full parallel overloading in siblings
18	Avoid case analysis on properties of objects
19	Prefer typing by attribute to typing by inheritance
20	An operation should only use classes of attributes of its class, classes of its parameters or classes of locally created objects
21	Avoid mirror fragments in class structures
22	Dependencies in inheritance hierarchies should not go from higher to lower levels.
23	Hard fragments should not depend on soft fragments
24	Avoid direct recursive associations

Table D.3: Object oriented design rules(Garzás and Piattini, 2007b)

Rule	Intended design characteristics
Dependencies of concrete classes	Dependencies should be on abstractions. Clients do not need to know the implemented class only it's services.
An object behaves differently according to its state	Different behaviour should be separated into different classes.
A class hierarchy has many levels	Inheritance is static (can not be changed during run-time). Too many levels makes maintenance difficult. Consider compositions instead of inheritance.
Something is used very little or not used at all	Simplifying the design and increasing maintainability.
A super class knows one of its sub-classes	This is a no-no! The use of polymorphism is inhibited and the design-solution counterproductive.
A class collaborates with many others	One ambition in OO is to keep coupling low and to model abstractions with "single" responsibilities. Makes the design sensitive to changes.
A change in an interface has an impact on many clients	It is better to have many specific interfaces than a single general-purpose one. Classes should be reusable. Martin(Martin 2003) defines this in the Interface Segregation Principle (ISP)
There is no abstraction between an interface and its implementation	To avoid duplicated code a majority of the methods of the interface should be implemented in the abstract class. However, it is important not to force derived classes to redefine implemented methods.
A super-class is concrete	The idea of inheritance is to make the descriptions of an abstraction more and more specialised. Having an instantiable super-class assigns two roles to the class, on one hand defining a common "part" for other classes and on the other hand being an entity by itself.
A service has a lot of parameters	A long parameter list is a "bad smell" (Fowler et al 1999) and indicates passing a large amount of data. Either pass data as objects or reconstruct for many services with few parameters.
A class is large	If a class is large it might indicate a violation of the Single Responsibility Principle (SRP) (Martin 2003). This means that the class is taking on to much responsibilities or services.
Elements of the user interface are within domain entities	The reason for this rule is to separate the core of a system from its presentation. This is stated in the early pattern Model-Controller-View.
A class uses more things from another class than from itself	Also known as the bad smell Feature envy (Fowler et al 1999). Cohesion is important in object orientation and is the notion of how fit the abstraction modelled is. The solution might be to relocate the service to another class.
A class rejects something it has inherited	Basically this is inheritance gone bad. Inheritance should be based on structural relationships, with specialisation increasing down the hierarchy. It is absolutely necessary for any object belonging to the hierarchy to deliver the expected services promised by any part in its line of the hierarchy.
If attributes of a class are public or protected	Encapsulation is an important part of object orientation and one of its strengths. The idea is for a class to offer services not being a container of values. The client should not be bothered by details of how the internal representation of the abstraction is implemented.

Table D.4: Effective Rules for Java(Bloch, 2001)

Creating and Destroying Objects	1	Consider Providing Static Factory Methods Instead of Constructors
	2	Enforce the Singleton Property with a Private Constructor
	3	Enforce Noninstantiability with a Private Constructor
	4	Avoid Creating Duplicate Objects
	5	Eliminate Obsolete Object References
	6	Avoid Finalizers
Methods Common to All Objects	7	Obey the General Contract when Overriding Equals
	8	Always Override HashCode When You Override Equals
	9	Always Override toString
	10	Override Clone Judiciously
	11	Consider Implementing Comparable
Classes and Interfaces	12	Minimize the Accessibility of Classes and Members
	13	Favor Immutability
	14	Favor Composition Over Inheritance
	15	Design and Document for Inheritance or Else Prohibit It
	16	Prefer Interfaces to Abstract Classes
	17	Use Interfaces Only to Define Types
	18	Favor Static Member Classes Over Non-Static
Substitutes for C Constructs	19	Replace Structures with Classes
	20	Replace Unions with Class Hierarchies
	21	Replace Enums with Classes
	22	Replace Function Pointers with Classes and Interfaces
Methods	23	Check Parameters for Validity
	24	Make Defensive Copies when Needed
	25	Design Method Signatures Carefully
	26	Use Overloading Judiciously
	27	Return Zero-Length Arrays, Not Nulls
	28	Write Doc Comments for All Exposed API Elements
General Programming	29	Minimize the Scope of Local Variables
	30	Know and Use the Libraries
	31	Avoid Float and Double if Exact Answers are Required
	32	Avoid Strings where Other Types are More Appropriate
	33	Beware the Performance of String Concatenation
	34	Refer to Objects by their Interfaces
	35	Prefer Interfaces to Reflection
	36	Use Native Methods Judiciously
	37	Optimize Judiciously
	38	Adhere to Generally Accepted Naming Conventions
Exceptions	39	Use Exceptions Only for Exceptional Conditions
	40	Use Checked Exceptions for Recoverable Conditions and runtime Exceptions for Programming Errors
	41	Avoid Unnecessary Use of Checked Exceptions
	42	Favor the Use of Standard Exceptions
	43	Throw Exceptions Appropriate to the Abstraction
	44	Document All Exceptions Thrown by Each Method
	45	Include Failure-Capture Information in Detail Messages
	46	Strive for Failure Atomicity
	47	Don't Ignore Exceptions
Threads	48	Synchronize Access to Shared Mutable Data
	49	Avoid Excessive Synchronization
	50	Never Invoke Wait Outside a Loop
	51	Don't Depend on the Thread Scheduler
	52	Document Thread-Safety
	53	Avoid Thread Groups
Serialization	54	Implement Serializable Judiciously
	55	Consider Using a Custom Serialized Form
	56	Write ReadObject Methods Defensively
	57	Provide a ReadResolve Method when Necessary

Table D.5: Micro Patterns(Gil and Maman, 2005)

Main Category	Pattern	Short description	Additional Category
Degenerate classes			
Degenerate state and behaviour	Designator	An interface with absolutely no members.	
	Taxonomy	An empty interface extending another interface.	
	Joiner	An empty interface joining two or more superinterfaces	
	Pool	A class, which declares only static final fields, but no methods.	
Degenerate behaviour	Function Pointer	A class with a single public instance method, but with no fields.	
	Function Object	A class with a single public instance method, and at least one instance field.	
	Cobol Like	A class with a single static method, but no instance members	
Degenerate state	Stateless	A class with no fields, other than static final ones.	
	Common State	A class in which all fields are static.	
	Immutable	A class with several instance fields, which are assigned exactly once, during instance construction.	
Controlled creation	Restricted Creation	A class with no public constructors, and at least one static field of the same type as the class.	
	Sampler	A class with one or more public constructors, and at least one static.	
Containment			
Wrappers	Box	A class which has exactly one, mutable, instance field of the same type as the class.	
	Compound Box	A class with exactly one non-primitive instance field.	
	Canopy	A class with exactly one instance field that it assigned exactly once, during instance creation.	Degenerate State
Data Managers	Record	A class in which all fields are public, no declared methods.	Degenerate Behaviour
	Data Managers	A class where all methods are either setters or getters.	
	Sink	A class whose methods do not propagate calls to any other class.	
Inheritance			
Base classes	Outline	A class where at least two methods invoke an abstract method on "this"	Degenerate State
	Trait	An abstract class, which has no state.	
	State Machine	An interface whose methods accept no parameters.	Degenerate State and Behaviour
	Pure Type	A class with only abstract methods, and no static members, and no fields.	
	Augmented Type	Only abstract methods and three or more static final fields of the same type	
	Pseudo Class	A class that can be rewritten as an interface: no concrete methods, only static fields.	
Inheritors	Implementor	A concrete class, where all the methods override inherited abstract methods.	
	Overrider	A class in which all methods override inherited, non-abstract methods.	
	Extender	A class that extends the inherited protocol, without overriding any methods.	