Heuristics for Designing Object-Oriented Examples for Novices

MARIE NORDSTRÖM and JÜRGEN BÖRSTLER
Umeå University

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

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

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

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

General Terms: Principles, Guidelines, Examples

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

1. INTRODUCTION

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

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

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

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

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

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

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

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

2. RELATED WORK

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

McConnell and Burhans [2002] examined how the coverage of basic concepts in programming textbooks has changed over time. They conclude that "there has been a trend of decreasing coverage for basic programming and subprogram concepts as other more current material has been added". In general, it seems as we are focusing more on syntactical issues than on problem solving. This view is also supported by De Raadt et al. [2005], who examined 49 textbooks used in Australia and New Zealand according their compliance with the ACM/IEEE curriculum recommendations.

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

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

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

3. ESSENTIAL CONCEPTS AND PRINCIPLES OF OBJECT ORIENTATION

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

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

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

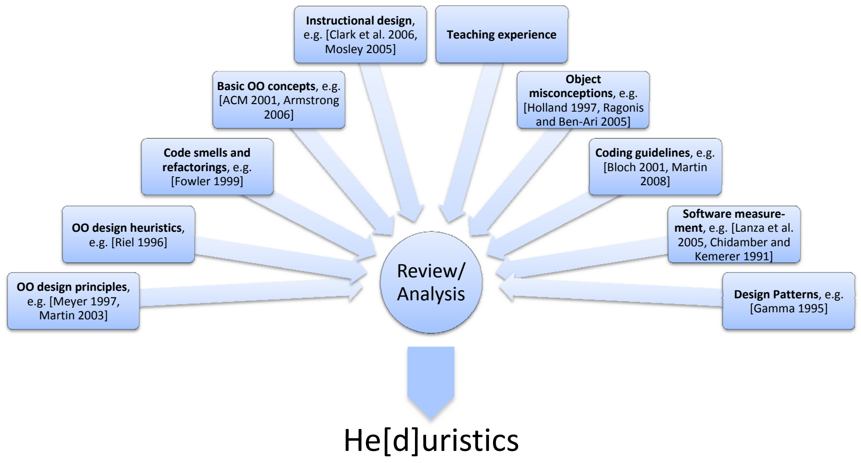

Instructional design, e.g. [Clark et al. 2006, Mosley 2005]
Teaching experience
Basic OO concepts, e.g. [ACM 2001, Armstrong 2006]
Object misconceptions, e.g. [Holland 1997, Ragonis and Ben-Ari 2005]
Code smells and refactorings, e.g. [Fowler 1999]
Coding guidelines, e.g. [Bloch 2001, Martin 2008]
OO design heuristics, e.g. [Riel 1996]
Software measurement, e.g. [Lanza et al. 2005, Chidamber and Kemerer 1991]
OO design principles, e.g. [Meyer 1997, Martin 2003]
Design Patterns, e.g. [Gamma 1995]
Review/ Analysis
He[d]uristics

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

code in return.

4. HE[D]URISTICS

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

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

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

4.1 Model Reasonable Abstractions

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

An abstraction should also be plausible, both from a software perspective as well as from an educational perspective. In particular, it must be plausible seen through the eyes of a novice. Concept formation is driven by cognitive economy and inference [Rosch 1999]. A good classification provides a maximum amount of information about (the properties of) a particular instance with the least cognitive effort. Concept formation is also influenced by the knowledge of and experience with the things that are categorized. A novice's concept formation will therefore be very different from an experienced software developer or domain expert. Which properties are perceived as meaningful or not can therefore be very different [Rosch 1999].

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

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

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

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

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

Listing 1. Application illustrating a for-loop.

```
public class Ex
{
```

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

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

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

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

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

  public int getRank()
  {
    return rank;
  }

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

  public char getSuit()
  {
    return suit;
  }

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

4.2 Model Reasonable Behaviour

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

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

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

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

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

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

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

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

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

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

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

4.3 Emphasize Client View

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

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

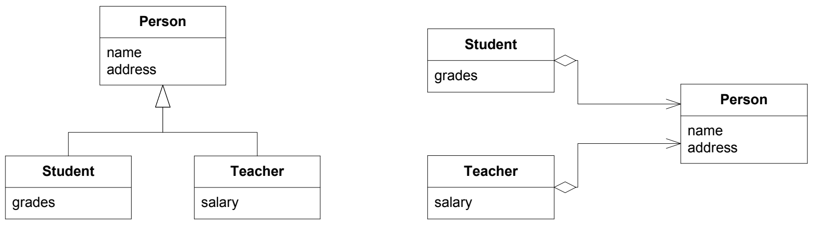
The methods provided by a class must make immediate sense to a novice or a good "cover story" must be provided that motivates object behaviour.

4.4 Favour Composition over Inheritance

Inheritance is the concept that distinguishes object orientation from other paradigms. It is therefore given significant coverage in introductory object oriented programming. Introducing inheritance early typically leads to very simple examples, since novices only have mastered a very limited repertoire of concepts and syntactical constructs. Early examples are therefore usually too simple to show the real power of inheritance. Furthermore, inheritance is often used to exemplify code reuse, which generally is considered a problematic motivation for inheritance [Armstrong and Mitchell 1994] and may lead to typing problems [Liskov and Wing 1994], for example when implementing a stack by inheriting from a vector or a queue.

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

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



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

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

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

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

4.5 Use Exemplary Objects Only

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

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

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

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

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

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

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

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

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

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

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

4.6 Make Inheritance Reflect Structural Relationships

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

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

Listing 5. A `Rectangle` class.

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

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

  public void setHeight(double h)
  {
    height=h;
  }

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

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

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

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

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

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

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

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

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

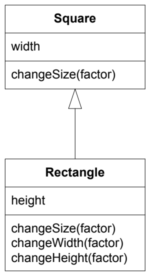


Fig. 3. `Rectangle` inheriting from `Square`.

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

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

5. DISCUSSION

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

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

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

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

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

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

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

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

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

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

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

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

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

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

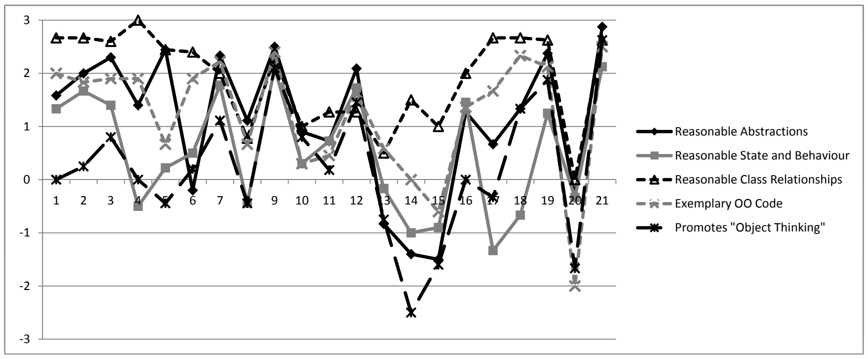


Fig. 4. Average object oriented quality scores for the 21 examples in [Börstler et al. 2010] (range: +3(extremely poor) .. -3(excellent)).

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

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

Listing 7. First User Defined Class: `GradeBook`

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

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

  // constructor initializes courseName
  public GradeBook( String name )
  {
    courseName = name;       // initialize courseName
  }
  // method to set the course name
  public void setCourseName( String name )
  {
    courseName = name;        // store the course name
  }

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

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

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

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

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

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

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

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

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

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

Listing 8. First User Defined Class: `Die`

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

  public Die()
  {
    faceValue = 1;
  }

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

  public int getFaceValue()
  {
    return faceValue;
  }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

  public int getFaceValue()
  {
    return faceValue;
  }

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

6. CONCLUSIONS

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

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

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

REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Received October 2010; ...