Improving OO Example Programs

Marie Nordström and Jürgen Börstler, *Member, IEEE*

***Abstract*—When teaching object oriented programming, educators rely heavily on textbook examples. However, research shows that such examples are often of insufficient quality regarding their object-oriented characteristics. In this paper, we present a number of guidelines for designing or improving object oriented example programs for novices. Using actual textbook examples, we show how the guidelines can help in assessing and improving the quality of object oriented example programs.**

***Index Terms*—Object oriented programming, Example programs, Guidelines, Quality, Education.**

I. Introduction

EXAMPLE programs are a key resource for the teaching or learning of programming [1], [2]. It has been argued that object orientation is a "natural" way of thinking and therefore well suited for problem solving using program development. However, several studies question this claim [3]. Novices, for example, have more problems understanding a delegation-based control style, which is central to object-orientation, than a centralized one. This adds to the difficulties of teaching object orientation.

We have little scientific theory or evidence guiding us in how to introduce object orientation. However, there should be no doubt that we need to strive for examples that emphasize the general characteristics of object orientation. Research shows, though, that the overall quality of object oriented example programs in introductory textbooks is insufficient [4], [5]. The quality of common example programs, like the famous "`HelloWorld`" program, have been critically discussed for a long time [6] and there have been ongoing debates on the object-orientedness of these and similar examples [7]–[9]. However, all of these discussions have focused on superficial technicalities, rather than the inherent object oriented qualities (and suitability) of the examples.

In this paper, we present quality guidelines for object oriented example programs for novices that have been derived from key object oriented concepts and design principles [10]. We then show how these guidelines can be used to assess and improve typical examples from introductory programming texts. The reader should note that programming is not an exact science and there are varying opinions on and perceptions of quality. Improving an example is always easier than developing it in the first place. This paper suggests a critical attitude towards the object oriented quality of example programs. If we want students to write quality code, we have to give them "good" examples and insist on commonly agreed object oriented principles, guidelines, and rules even in small scale examples for novices.

Marie Nordström is with the Department of Computing Science, Umeå University, Umeå, Sweden. Corresponding e-mail: marie@cs.umu.se

Jürgen Börstler is with the Department of Computing Science, Umeå University, Umeå, Sweden (on leave) and with the School of Computing, Blekinge Institute of Technology, Karlskrona, Sweden. Corresponding e-mail: jurgen.borstler@bth.se

Manuscript received November X, 2010; revised month date, 2010.

II. Related Work

Textbooks are a major source for example programs and also work as a reference for how to solve specific problems.

Becker [11] reviewed 16 introductory object oriented programming texts and analyzed how objects and classes are introduced, how I/O is handled, and how they support software engineering concepts. His results show that it is difficult to find a text that meets every need, but that there are books that do a good job.

McConnell and Burhans [12] have examined the coverage of basic concepts in programming textbooks over time and observed a shift in the amount of coverage of various topics with each new programming paradigm. With object orientation they observe a decrease in the treatment of subprograms but also a decrease in basic programming constructs.

De Raadt et al. [13] examineed 49 textbooks used in Australia and New Zealand and found quite poor compliance with the ACM/IEEE curriculum guidelines. One explanation they offered for the poor compliance was that many texts are more focused on the syntactical details of a programming language than on conveying a more holistic view of programming as a problem solving tool.

In a multi-national study Lister et al. [14] concluded that few students are able to articulate the intent of code when asked to "think out loud" while taking a multiple-choice questionnaire.

Research also revealed that students hold a range of misconceptions about object orientation. Ragonis and Ben Ari [15] identified seven such misconceptions about program flow in object orientated programs. The same authors also present a comprehensive list of frequently observed difficulties and misconceptions of novices along with probable sources of these problems [16].

Holland et al. [17] discuss misconceptions concerning the concept of an object. They present a number of misconceptions that might relate to particular features of example programs and suggest guidelines for example construction to avoid these problems. Fleury [18] identifies a number of student constructed rules, where students generalize from valid models in erroneous ways.

Börstler et al. [4], [19] developed a checklist for evaluating the quality of object oriented examples for novices. Their checklist covers three aspects of example quality; technical quality, object-oriented quality, and didactic quality. Object oriented quality is captured by the following five quality factors: Reasonable Abstractions (O1), Reasonable State and Behaviour (O2), Reasonable Class Relationships (O3), Exemplary OO code (O4), Promotes "Object Thinking" (O5).

Results of a large scale study shows that the object oriented quality of example programs in common introductory text books is low [4], [5].

III. Educational Heuristics for OO Examples

Based on key concepts found in literature, and on key object oriented design principles used by the software developing community, a number of educational heuristics were developed and described in [10]. The intention of these heuristics is to support the design of exemplary examples, with respect to the characteristics of object orientation, and at the same time conceptually addressing the particular needs of novices. Based on the experiences with the design, evaluation, and use of an evaluation tool for object oriented examples for novices [4], [5], [19], further fine-tuning of these heuristics have resulted in five *Eduristics*.

The Eduristics are targeted towards general design characteristics, which means that more detailed practices, like keeping all attributes private, are not stated explicitly. Furthermore, they are designed to be independent of a particular pedagogic approach (objects first/late, order of concepts, ...), language, or environment.

In short, the Eduristics can be summarized as follows:

1) Model Reasonable Abstractions
   - Abstractions must be meaningful from a software perspective, but also plausible from a novice's point of view.
   - Do not put the entire application into `main` and isolate it from other application classes.
   - No God classes [20].
2) Model Reasonable Behaviour
   - Show objects changing state and behaviour depending on state.
   - Do not confuse the model with the modelled.
   - No classes with just setters/getters ("containers").
   - No code snippets.
   - No printing for tracing; use `toString` to communicate textual representations.
3) Emphasize Client View
   - Promote thinking in terms of services that are required from explicit clients.
   - Separate the internal representation from the external functionality.
4) Promote Composition
   - Emphasize the idea of collaborating objects; use non-trivial attributes to emphasise the distribution of responsibilities.
   - Do not use inheritance to model roles [21].
   - Inheritance should separate behaviour and demonstrate polymorphism.
5) Use Exemplary Objects Only
   - Promote "object thinking", i.e. objects are autonomous entities with clearly defined responsibilities.
   - Instantiate multiple objects of at least one class.
   - Do not model "one-of-a-kind" objects.
   - Make all objects/classes explicit: avoid anonymous classes and explain where objects that are not instantiated explicitly come from.
   - Make all relationships explicit: avoid message chains. Objects should only communicate with objects they know explicitly (Law of Demeter [22]).
   - Avoid shortcuts.

Using exemplary objects in particular means to support students in their generalizing from concrete examples to general properties. Emphasising improper role models might lead to erroneous generalisations or misconceptions. Many introductory (Java) textbooks use classes like `String` or `Math` to introduce the concepts of object and class. However, neither of them is a good role-model of a class. String-objects are immutable and can therefore not be shared, which means that they do not behave like "proper" objects. `Math` is just a container for methods and cannot even be instantiated.

We are well aware that real software features more non-exemplary than exemplary objects (see for example [**?**]), but from a teaching and learning point of view initial examples need to be prototypical for a concept [**?**], [**?**].

Our five Eduristics correspond well with the object oriented quality factors of the evaluation tool used in [4], [5]. We therefore argue that they help to increase the object oriented quality of examples for novices.

IV. A Closer Look at Examples

An important category of examples is the first user defined class (FUDC). For a novice, this example sets the stage for a typical class and must therefore be carefully chosen. Defining a small, simple, and easy to understand example, that exhibits high object oriented quality is a challenging task, in particular since novices only have a very small repertoire of concepts and syntactical constructs. As part of a larger study, a number of FUDCs from common introductory textbooks were evaluated and analysed in [4], [5]. The results of this analysis are discouraging; many examples score low in particular regarding their object oriented quality.

In the following sub-section, we use the Eduristics to examine a typical FUDC-example. This is followed by a more general discussion on common deficiencies in FUDCs. Alternative designs are proposed and finally we suggest a number of suitable abstractions to use in examples with appropriate contexts.

A. The `BMI`-example

Fig. 1 shows an example of a FUDC from a common textbook [24, chpt. 10].

The class is modeling Body Mass Index[1] objects. The class is presented as an attempt to make a previously introduced static method (see below) for calculating the BMI reusable.

[1]"Body Mass Index (BMI) is a number calculated from a person's weight and height. For adults 20 years old and older, BMI is interpreted using standard weight status categories that are the same for all ages and for both men and women. For children and teens, on the other hand, the interpretation of BMI is both age- and sex-specific" http://www.cdc.gov/healthyweight/assessing/bmi/adult_bmi/index.html

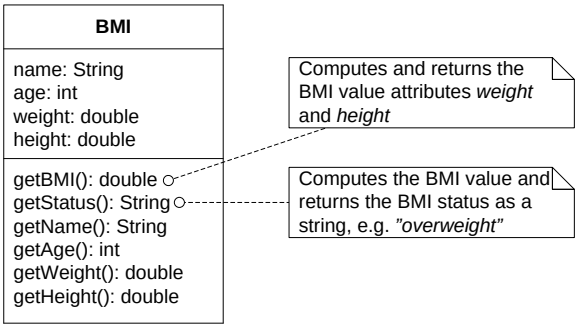

Fig. 1. UML class diagram for `BMI`.

```
public static double getBMI(
    double weight, double height);
```

The example text argues that if there is a need to associate the weight and height with a person's name and birth date, the ideal way to couple them, would be to create an object that holds them all. According to the text, this example demonstrates the advantages of the object-oriented paradigm over the procedural paradigm. Using our Eduristics, we would evaluate the object oriented qualities of this example in the following way:

1) *Model Reasonable Abstractions*: Abstraction is a mechanism to cope with complexity. A good abstraction focuses on the essential properties of a phenomenon while ignoring irrelevant details. BMI is basically a value or value/weight status pair and the `BMI`-class could be said to focus on these essential properties. However, it is difficult to understand why `name` and `age` should be properties (state) of a `BMI`-object. In the present case, the attributes `name` and `age` are never even used, although BMI's for children and adults are calculated differently. It is therefore unclear what `BMI` is actually modelling.
2) *Model Reasonable Behaviour*: The `BMI`-class is just a "wrapper" for some immutable values set on instantiation. The state of an `BMI`-object never changes, although it would seem reasonable that at least `age` and `weight` should be mutable. Nevertheless, the BMI-value and its corresponding status are recalculated for every call of the methods `getBMI()` and `getStatus()`. This is highly unreasonable behaviour, not only from the class' interface point of view, but also from an implementation point of view.
3) *Emphasize Client View*: `BMI`-objects do not offer any significant services. The only thing clients can do is to group a set of immutable values and ask `BMI`-objects for "their own values":

```
BMI bmi = new BMI(...);
double bmiValue = bmi.getBMI();
String bmiStatus = bmi.getStatus();
```

Martin argues that a model can only be meaningfully validated in terms of its clients [27]. In the `BMI` case it is, however, difficult to imagine a client.

5) *Use Exemplary Objects Only*: `BMI`-objects do neither have mutable state nor meaningful behaviour. Although the accompanying test program instantiates two `BMI`-objects that does not help much to emphasise the difference between objects and classes, since the only thing done with the objects is to get their BMI-values and print them to `System.out`. The example does not promote object thinking; the objects are no autonomous entities with clearly defined responsibilities.

This example also illustrates the importance of naming. Judging from the attributes of the `BMI`-class, it seems more adequate to name this class something like `PersonalData`, which is responsible for keeping track of the weight-history for an individual. Then calculating the BMI would be a smaller task to be performed for some supplied service. However, naming is also dependent on the context. The name of a class must adequately describe a phenomenon from the problem domain's point of view. The name should also give proper associations of what to expect from objects of this type.

B. Common example deficiencies

Simple "structures", like `Clock`, `Card`, `GradeBook` etc. are commonly used as early examples in introductory programming textbooks. They are characterised by a number of attributes of basic types and `String`, and methods to set and get those attributes individually. See Fig. 2 for typical designs, taken almost straight out of common introductory Java textbooks. Formally, all examples in Fig. 2 follow the fundamental notion of a class as encapsulating data and methods that operate on that data. However, we would argue that a method does not really operate on the data, if the only thing it does is to assign or retrieve the value of an instance variable.

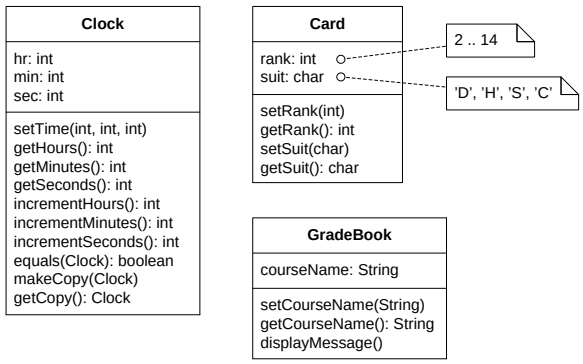

Fig. 2. UML class diagram for common container classes.

All three abstraction in Fig. 2 are reasonable. They model intuitive entities that can be easily placed in some context, but we would still argue that they lack certain object oriented qualities. Primarily, they do not exhibit reasonable behaviour (Eduristic 2), and they do not emphasize client view (Eduristic 3).

However, it is possible to improve examples like the ones above without too much effort. The `Clock`-example might be turned into a `StopWatch` (see Fig. 3), or a `ClockDisplay` (see Fig. 4) that both add reasonable behaviour beyond just set- and get-methods. The `ClockDisplay`-example goes even

a step further and shows non-trivial collaboration between objects without making the example overly complex.

The `Card`-class could be a central class of some card game, but this would require substantial extensions, both in size and complexity. One would at least need classes for rank and suit to complete the actual card abstraction (see [26] for an excellent discussion on when to make a type). Furthermore, additional classes for deck and hand objects might be needed to do something meaningful with `Card`-objects.

A `GradeBook`-object basically is a container for student-grade pairs. Reasonable behaviour could for example be added by methods for the calculation of average scores etc. However, it would still basically be a container, but now with a "bloated" interface/protocol that violates the Single Responsibility Principle [27].

C. Alternative clock-designs

To use the idea of a clock, and still make it an exemplary example, we have found two appealing designs. The first example is the timer, or the stopwatch, see Fig. 3.

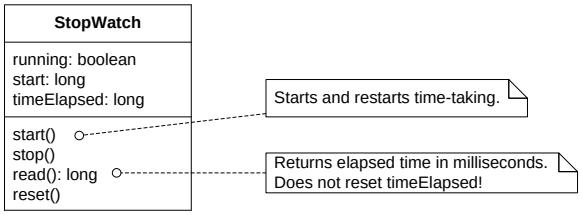

Fig. 3. Class diagram for `StopWatch` [28].

In this design, the actual representation of time is delegated to a system supplied service. The state is easy to understand, either the timer is running, or not. Time is not updated continuously by the object, but calculated when needed (method `read`). The possibilities to accumulate time, and to reset the timer provides clients with the typical features one would expect from a timer or stop watch. This makes `StopWatch` an intuitive and useful abstraction with reasonable behaviour.

Another example of a simple and elegant design, is the `ClockDisplay`-example, shown in Fig. 4.

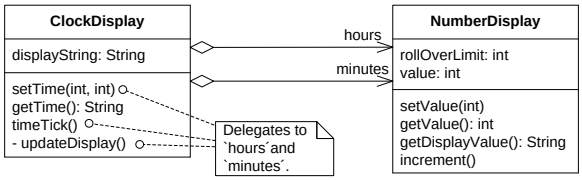

Fig. 4. Class diagram for `ClockDisplay` and `NumberDisplay` [29].

This example even illustrates collaborating objects, without adding much to the overall complexity of the example. Using instance variables of class-type early on makes it easier for novices to understand that instance variables don't need to be of basic types. Although this is a common novice misconception [17], we found that early examples almost exclusively feature instance variables of basic types.

D. Further Examples

In this section we give a short list of suggestions for suitable early examples.

- `Die`: A simple, intuitive abstraction that can be easily placed in meaningful contexts and it is easy to imagine applications using several Die-objects. Rolling a die is a meaningful non-trivial behaviour. Die-objects could even be instantiated with different numbers of faces to illustrate that quite different objects can be instantiated.
- `BankAccount`: Can be a very good example, if used with care. A bank account is a simple and intuitive abstraction and bank account-objects have simple but non-trivial behaviour. It is also fairly easy to imagine meaningful contexts and clients. However, educators must be aware that specializations of bank accounts are localized phenomena and students might have a hard time to figure out the meanings of checking and saving accounts.
- `ClickCounter`: A very simple abstraction modelling a manual counter that increases a value on every click. Has many features in common with the `StopWatch` shown in Fig. 3, but is even simpler.
- `TrafficLight`: Slightly more complicated abstraction that illustrates state dependent behaviour.

Further suitable examples can be derived from situations where we—out of convenience—usually just use variables of some base type. For example to model monetary values, it is very common to use just a variable of floating point type. However, money arithmetic works slightly different than floating point arithmetic. Furthermore, things will become complicated in case one needs to handle different currencies. Fowler [26] presents an interesting discussion on this subject and recommends to define a type (class) whenever some entity needs special behaviour in its operations that a primitive type do not have.

V. Conclusion

The object oriented quality of examples for novices is often insufficient. To prevent students from making premature erroneous generalisations, we find it important to always be faithful to the object-oriented paradigm. Since students will use examples as role models, they must not contradict the concepts and rules we intend them to pick up. In this paper we have described how the design of examples can be enhanced by the use of educational heuristics. Small details can often make a big differences when it comes to object oriented quality. When dealing with examples for novices, it is important to respect that every example serves two purposes. On one hand it demonstrates a particular concept or feature of the paradigm or the language. On the other hand it is also a stepping stone in conveying the general idea of the paradigm, and this must be given appropriate attention.

An important ingredient of an example is its context. A good abstraction focuses on the essential characteristics of some object from a particular point of view [27], [30]. Without this point of view, it can be difficult to make sense of an abstraction. The context, or cover story [4], provides meaning and motivation and supports novices in understanding of the

source code: What problem is being addressed and what is the reason for this particular solution?

REFERENCES

[1] J. Anderson, R. Farrell, and R. Sauers, "Learning to program in LISP," *Cognitive Science*, vol. 8, no. 2, pp. 87–129, 1984.

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

[3] M. Guzdial, "Paving the way for computational thinking," *Commun. ACM*, vol. 51, no. 8, pp. 25–27, 2008.

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

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

[6] R. Westfall, "'hello, world' considered harmful," *Communications of the ACM*, vol. 44, no. 10, pp. 129–130, 2001.

[7] CACM, "Hello, world gets mixed greetings," *Communications of the ACM*, vol. 45, no. 2, pp. 11–15, 2002.

[8] M. H. Dodani, "Hello world! goodbye skills!" *Journal of Object Technology*, vol. 2, no. 1, pp. 23–28, 2003.

[9] CACM Forum, "For programmers, objects are not the only tools," *Communications of the ACM*, vol. 48, no. 4, pp. 11–12, 2005.

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

[11] B. Becker, "Pedagogies for cs1: A survey of java textbooks," 2002.

[12] J. J. McConnell and D. T. Burhans, "The evolution of CS1 textbooks," in *Proceedings FIE'02*, 2002, pp. T4G–1–T4G–6.

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

[14] R. Lister, E. S. Adams, S. Fitzgerald, W. Fone, J. Hame, M. Lindholm, R. McCartney, J.-E. Moström, K. Sanders, O. Seppälä, B. Simon, and L. Thomas, "A multi-national study of reading and tracing skills in novice programmers," *SIGCSE Bull.*, vol. 36, no. 4, pp. 119–150, 2004.

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

[16] ——, "A long-term investigation of the comprehension of OOP concepts by novices," *Computer Science Education*, vol. 15, no. 3, pp. 203–221, 2005.

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

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

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

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

[21] M. Fowler, "Dealing with roles," in *Proceedings of the 4th Pattern Languages of Programming Conference (PLoP)*, 1997.

[22] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol. 6, no. 5, pp. 38–48, 1989.

[23] M. Nordström and J. Börstler, "Heuristics for designing object-oriented examples for novices," *TOCE (to be submitted)*, vol. ??, no. ??, p. ??, 2010.

[24] Y. D. Liang, *Introduction to Java programming : comprehensive version*, 8th ed. Upper Saddle River, N.J.: Pearson Education, 2009.

[25] J. Parsons and Y. Wand, "Choosing classes in conceptual modeling," *Communications of the ACM*, vol. 40, no. 6, pp. 63–69, 1997.

[26] M. Fowler, "When to make a type," *IEEE Software*, vol. 20, no. 1, pp. 12–13, 2003.

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

[28] D. A. Bailey and D. W. Bailey, *Java Elements–Principles of Programming in Java*. McGraw Hill, 2000.

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

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

Marie Nordström is finishing her PhD in Computer Science didactics, after many years of teaching introductory programming to both CS majors and minors.

Jürgen Börstler is a professor of Computing Science with many years of experience in object oriented analysis and design and a strong interest in computer science education research.