Checklists for Grading Object-Oriented CS1 Programs: concepts and misconceptions

Kate Sanders
Mathematics and Computer Science Dept.
Rhode Island College
Providence, RI 02908 USA
ksanders@ric.edu

Lynda Thomas
Department of Computer Science
University of Wales, Aberystwyth
Ceredigion, SY23 3DB Wales, UK
ltt@aber.ac.uk

ABSTRACT
In this paper, we begin by considering object-oriented programming concepts and typical novice misconceptions, as identified in the literature. We then present the results of a close examination of student programs in an objects-first CS1 course, in which we find concrete evidence of students learning these concepts while also displaying some of these misconceptions. This leads to the development of two checklists that educators can use when designing or grading student programs.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education—computer science education

General Terms
Human Factors.

Keywords
CS1, object-oriented concepts, misconceptions, empirical research, assessment

1. INTRODUCTION
Teaching programming has always been challenging. There is an ongoing debate about whether object-oriented (OO) programming is more difficult to teach than procedural [6]. Whether or not it is inherently more difficult, it is currently harder because many instructors simply don't have as much experience with the OO approach and typical student problems.

In this paper, we hope to make teaching OO programming a little easier by offering checklists to help instructors quickly:

- verify that students are using the OO concepts typically taught in a CS1 course, and also
- diagnose some common student problems based on the programs they write.

(See Tables 1 and 2).

In Section 2, we discuss OO programming concepts and typical novice misconceptions as identified in the literature. Both misconceptions and concepts are shown in **bold** so that we can refer back to them later.

In Section 3, we describe the CS1 course that this research was based on, the programs, the concepts they covered (also in **bold**), the students, and the way we examined their submitted solutions to determine whether they appeared to understand the relevant concepts or exhibited the identified misconceptions.

Section 4 presents the results of this examination. We observed that the ways in which students exhibit misconceptions are more subtle than we expected and give several examples. Our results are summarized in two checklists for use in designing and grading programs.

Section 5 presents some general conclusions and future work.

2. RELATED WORK
Armstrong [1] surveys numerous papers on object-oriented development and compiles a list of object-oriented concepts. Those identified in more than half of the sources include **inheritance**, **object**, **class**, **encapsulation**, **method**, **message passing**, **polymorphism**, and **abstraction**.

Several empirical studies have looked at student understanding and misconceptions of object-oriented programming, based on student questions during labs [4], student designs for a simple object-oriented program [10,7], students' responses when asked to predict which of a set of programs would work [3], and extensive interviews [2]. In addition, Holland et al. [5] suggest a list based on their experience teaching CS1.

These studies examine different areas of object-oriented programming, but they draw a consistent picture. To summarize, they find the following misconceptions:

1. **errors in basic mechanics**
2. **instance/class conflation**: classes and instances are the same [5,4]
3. **problems with linking and interaction** [10]
4. **class /collection conflation** [10]: a class is a collection of instances, not an abstraction from its instances
5. **problems with abstraction** [7] **and hierarchies** (including inheritance, abstract methods, overriding, interfaces) [4]
6. **problems with modelling** [2,10]: failing to recognize that a class models something in the problem domain
7. **classes just text** [2]

8. **class/variable conflation**: classes are just wrappers for instance variables [5]
9. **identity/attribute confusion**: a variable that references an object is part of the identity of the object, like its name [5]
10. **belief that objects are simple records** for storing and retrieving data [5]
11. **problems with encapsulation** [3]
12. **work in methods is all done by assignment**, not by message passing [5]
13. **problems with accessors/mutators** [4]
14. **problems with constructors** [3, 4]

3. METHODOLOGY

3.1 Overview

In this project, we take an in-depth look at the programs written by the students in a single section of CS1. The class from which our data were gathered contained sixteen students, five women and eleven men. Five of the sixteen (four men and one woman) were CS majors. Five had had some previous programming background, including only one of the CS majors.

The course took an objects-first approach [9]. Classes, inheritance, and polymorphism were covered before such traditional topics as arithmetic and conditionals. There were five programming projects, all of which featured graphics (the first three using a simple graphics library provided with the text [9], the fourth and fifth using Java2D and Swing).

Aside from the topic order and the extensive use of graphics, the course design was fairly typical. There were three hours of lecture/discussion per week, plus a one-hour weekly lab, quizzes, a final exam, and the five programming assignments. All the programs were based on subjects previously covered in lectures and lab and there were also examples of relevant code for all programs in the text.

Altogether, the students submitted 71 programs, totalling several hundred pages of code. We read and re-read these, looking for significant features. We then turned again to the literature for a list of key concepts and misconceptions, developed and refined a list of features that we thought might indicate understanding (or lack thereof), and read the programs yet again.

3.2 The programs

The students had the opportunity to demonstrate understanding of all the concepts (and most of the misconceptions) listed above in their programs. The terms in **bold** in this subsection represent the essential concepts that were covered in each program.

Program 1 required the students to use the textbook's graphics library to create a simple cartoon. Each cartoon was required to include three instances of a "Cartoon Critter" class with at least five parts. This assignment has four main goals. The students were to: demonstrate understanding of the **basic mechanics** of defining classes, instance variables, and methods; define and instantiate a composite class (the `CartoonCritter`), show understanding of the **containment (has-a) hierarchy;** show an ability to work with **constructors** and **parameters** by giving the `CartoonCritter` constructor parameters that allowed different instances of the Critter to vary (if only in location); and show an understanding of **the class/instance** distinction by instantiating three different Critter objects.

Program 2 was required to display a "tour guide" and some other objects on the screen, such that when the user clicked on one of the other objects, the tour guide would display information about the object that was clicked on. There were to be two different classes of object that behaved slightly differently when clicked on. In addition to the OO requirements of Program 1, students were to create **objects that interact** with each other (by sending messages and making use of the return values from those messages), and create a small **inheritance (is-a) hierarchy** of classes, where instances of the subclasses would behave **polymorphically**.

Program 3 required the students to display an object with at least six parts, plus a row of five "buttons" (colored ellipses). When one of the parts of the object was clicked on, it should display a black outline. Then when one of the buttons was clicked on, the currently selected shape should change to the color of the button. Two new concepts were key to this program: Students needed to (1) create a very simple invisible object (i.e., an object that doesn't correspond to anything visible on the screen) to hold the currently selected shape; and (2) create **use-a relationships** by passing this peer object as a parameter to each of the other objects that needs to communicate with it. Thus students were to demonstrate a deeper understanding of **interacting objects** and **abstraction** by adapting and using a very simple **design pattern**, the Holder pattern. This pattern, introduced in the text [9], consists of one object that manages a particular piece of data (generally one instance variable plus accessor/mutator methods) and two or more objects that use that piece of data.

Program 4 required the students to switch to Java's graphics libraries to code a subset of the Tetris game. The Tetris shapes were simply supposed to appear at the top of the window, fall to the bottom, and disappear. In this program there was an opportunity to define an **inheritance hierarchy** (by defining a `Piece` class and extending for different shapes). Whether or not the students chose to do so (with no prompt), gave us some sense of how well they had integrated inheritance into their mental model.

Program 5 introduced no new concepts. It was intended to be a capstone experience for the semester, pulling together the concepts previously studied into a somewhat longer program requiring more **abstraction**. It gave students the choice of two very simple games.

4. RESULTS

The students all showed some understanding of the key concepts we wanted to teach: **inheritance**, **object**, **class**, **encapsulation**, **methods**, **message passing**, **polymorphism**, and **abstraction** [1]. All students created multiple instances of some classes; created objects that interacted; used containment and inheritance hierarchies; and some used invisible objects and design patterns.

Most of them also showed evidence of some misconceptions, however. In Section 2 we identified the essential misconceptions that a reading of the literature might lead us to expect in student programs. We now look at each of these misconceptions in turn.

4.1 Basic mechanics

Unlike Garner et al. [4] we did not see much evidence of problems with basic mechanics. All but one of the students submitted a working program for the first assignment that approximated what was required; the remaining student (Student 4) submitted a working, satisfactory program for the second assignment. This knowledge may have been somewhat 'fragile,' however [8]. As programs got harder, fewer of them compiled, and students made elementary mistakes such as leaving code outside of methods (e.g., Student 2, Program 5). On the other hand, these errors may have been due to lack of time rather than lack of understanding.

4.2 Instance/class conflation

Program 1 was designed to get students thinking about the difference between class and instance [5]. Surprisingly, even though we found no evidence of difficulty distinguishing between library classes and their instances, we observed several problems distinguishing between user-defined classes and instances of those classes. Students 2 and 8, for example, defined separate classes for Program 1 and instantiated each one once, instead of instantiating the same class three times (e.g., `Head`, `Head_1`, and `Head_2`; `Body`, `Body_1`, and `Body_2`). The classes have the same fields and methods; they differ only in their property values.

Another student (S15) created three very different classes for Program 1 and instantiated each one once. This design could have been explained as an artistic decision rather than a misconception (although it did conflict with the assignment specifications). The code's comments, however, show an identification of classes and instances, for example: "Definition of the `SweetBunny` class … Also known as `_black` in `BunnyApp.`"

In Program 3, the problem of instance/class confusion was displayed in a more sophisticated way. The students had now learned about inheritance, and some of them over-used it. They were required to create six buttons (i.e., clickable `Ellipses`) that each corresponded to a different colour. Instead of using a single class and creating different instances for each colour, Student 1 (for example) defined a class `ColorButton` with subclasses `OrangeButton`, `CyanButton`, etc. It seems that students were **confusing subclass with instances**.

4.3 Problems with linking and interaction

As noted above, all but one of the students showed the ability to successfully **link** multiple classes together into a compiling program in Program 1 [10]. There was evidence that they didn't completely understand this process, however: all but three of the programs passed parameters into constructors that were not then used.

All 13 of the students who submitted Program 2 gave evidence of understanding how objects interact in simple ways with effective use of method calls and parameters. These programs contained from 3 to 9 different classes.

Program 3 required objects to communicate in a more sophisticated way. 13 of the 16 programs used parameters in simple situations correctly, and nine of those used the return values of methods. But only two students understood the Holder pattern (with its **use-a relationship**) well enough to get their programs to work properly. Four students had near misses. They created the shapes correctly, for example:

```
   Tree tree = new Tree(10,150,_cursor);
```

but never linked up the `_cursor` in the `Tree` constructor.

In Program 4 (the minimal Tetris game) 11 of 14 students passed the basic board into the shapes as a peer object and then used methods on that board, thus demonstrating an understanding of the 'use-a' relationship.

As the semester went on, they occasionally referenced nonexistent classes, but that seems to have been due to the increased complexity of the programs, rather than misconceptions about linking.

4.4 Class/collection conflation

One of the programs showed evidence of class/collection conflation [10]. Program 2 by Student 8 contained two classes, `Team` and `Team1`. The two class definitions were very similar. Upon examination, it became clear that the instances of `Team` all belonged to one United States' baseball league (the American League East), and the instances of `Team1` all belonged to another (the American League West).

4.5 Problems with hierarchies and abstraction

Abstraction is a fundamental concept of programming. Data abstraction is fundamental to the OO paradigm [1]. And it has been widely claimed that object-oriented programming is difficult because of the amount of abstraction it requires [6].

In addition to the abstraction inherent in the idea of classes and objects, and discussed above, OO programming requires the understanding of both containment and inheritance hierarchies. In Program 1, all but one of the programs submitted contained at least two classes and used some form of **containment hierarchy**.

Students 5 and 12 showed continuing problems with the **containment hierarchy**. In Program 1, Student 12 put all three hats in one class, all three bodies in another, etc., and Student 5 put all his or her code in a single class. In Program 4, they each instantiated several squares, enough to make two or more Tetris pieces, in a single class.

Students created **inheritance** hierarchies by extending library shapes like `Ellipse` and `Rectangle`. In Program 2, for example, Student 5 created two classes, `Player` and `Pitcher`, that both extended `Ellipse` but responded differently to user input. They also defined their own inheritance hierarchies. Student 9 defined a `Ghost` class for Program 2 with two subclasses, `LazyGhost` and `ScaryGhost`.

The students did not always use inheritance correctly. In some cases, as noted above in Section 4.2, subclasses should have been instances of a single class.

Finally, students frequently failed to use inheritance when they could have done so. In Program 4, for example, although the various Tetris piece classes shared code, the students defined completely separate classes for each. Because it's easy to cut and paste from one class to another, there's no immediate incentive to factor out code.

Another form of abstraction involves the use of **interfaces**, which might be considered 'abstraction by role'. Students used interfaces in Programs 3, 4 and 5 that were similar to those they had seen in the lectures, the labs, or the text, but did not factor out method signatures into new interfaces of their own.

Students seem to be comfortable using 'recipes' or abstractions of *small* pieces of code. For example, they all consistently used the recipe we provided for class definitions: instance variables first, then a constructor, then other methods, with the `main` method (if any) at the end.

The students had more trouble with design patterns. Many succeeded in using the Composite Pattern, but less than half came close to success with the Holder Pattern in Program 3 (see Section 4.4).

4.6 Failures in modelling

Eckerdal and Thuné's students experienced classes and objects at different levels, from the lowest level as pieces of code, up to models of the real world [2].

In Program 4 (a very simple Tetris program) the students in our sample used a fish-tank program they had seen in lab as a starting point. 10 programs (of 14 submitted) included remnants of the code for the fish tank. These students appear to be experiencing code as no more than text that is merely copied and pasted rather than something with real-world meaning.

Other students gave evidence of Eckerdal and Thuné's next level: they created objects that functioned within a program, but didn't correspond to the domain. For example, Student 2 in Program 1 defined a `Body` class that contained a `_body` instance variable of type `Ellipse` (as well as arms and legs). Student 3 in Program 1 defined a `Leg` class as part of his or her `Critter` that itself contained two `Leg` instance variables.

Other students, as noted above, modelled classes successfully but failed to model the inheritance relationship among them.

4.7 Misconceptions we *didn't* find

There was little evidence in these programs of **class/variable conflation**, **identity/attribute conflation**, or **objects-only data records**. Students did not define classes with only one instance variable, unless this was appropriate. The nature of the programs made identity/attribute conflation more or less impossible, and using graphics so immediately and extensively made it unlikely that students would assume objects were only data records.

We found no evidence of problems in **understanding accessor/mutators**. Fleury [3] noted that students had problems with **encapsulation**, being reluctant to give the same method names to methods on different classes. We did not observe this.

We found no evidence of **work in methods being exclusively done by assignment**. Starting in Program 1, students wrote methods that sent messages to other objects (e.g., setting their size or location). Where needed, these methods also involved computation.

Finally, we found no evidence of **problems with constructors**. As noted above, the students followed our recipe for class definitions, which included a constructor, and consistently used constructors where appropriate.

5. CONCLUSIONS AND FUTURE WORK

Looking so closely at students' programs has been a fascinating and sometimes sobering experience for us. We realise that frequently when grading student programs we are so rushed for time that we look at superficials: does it compile, does it fulfil the test cases, are there some comments? We hope that we note when there is something really wrong with the design.

What is evidenced in these programs, however, is that working programs can contain subtle errors that suggest a serious misconception. For instance, the variations that students showed of the class-instance-conflation misconception were a real surprise to us.

We summarize our results in two checklists (Tables 1 and 2) of things instructors can look for when grading CS1 programs. We hope that these checklists will help instructors both in designing assignments that test particular concepts and misconceptions and in grading those assignments (without spending hours reading and re-reading each program, as we did!)

We plan to build on this work by examining programs from other classes and other institutions. Moreover, having identified these symptoms, we plan to design experiments that will further investigate student misconceptions.

6. REFERENCES

[1] D. J. Armstrong. The Quarks of Object-Oriented Development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] A. Eckerdal and M. Thuné. Novice Java Programmers' Conceptions of 'object' and 'class', and Variation Theory. *ITICSE-05*, pages 89–93, 2005.

[3] A. E. Fleury. Programming in Java: student-constructed rules. *SIGCSE-00*, pages 197–201, 2000.

[4] S. Garner, P. Haden, and A. Robins. My Program Is Correct But It Doesn't Run: a preliminary investigation of novice programmers' problems. *ACE-05*, pages 173–180, 2005.

[5] S. Holland, R. Griffiths, and M. Woodman. Avoiding object misconceptions. *SIGCSE-97*, pages 131–134, 1997.

[6] R. Lister, A. Berglund, T. Clear, J. Bergin, K. Garvin-Doxas, B. Hanks, L. Hitchner, A. Luxton-Reilly, K. Sanders, C. Shulte, and J. Whalley. Research Perspectives on the Objects-Early Debate. *SIGCSE Bulletin* (2006).

[7] R. Or-Bach and I. Lavy. Cognitive activities of abstraction in object orientation: an empirical study. *SIGCSE Bulletin*, 36(2): 82–86, 2004.

[8] Perkins, D. and Martin, F. (1986) Fragile Knowledge and Neglected Strategies in Novice Programmers. In Soloway, E. and Iyengar, S. (Eds) *Empirical Studies of Programmers*. Ablex, NJ, USA. pp. 213-229.

[9] K. Sanders and A. van Dam. *Object-Oriented Programming in Java*. Addison-Wesley, Boston, 2006.

[10] B. Thomasson, M. Ratcliffe, and L. Thomas. Identifying novice difficulties in object-oriented design. *ITICSE*-06, 2006.

ACKNOWLEDGMENTS

Thanks to Robert McCartney, to Ann Moskol, to NSF for their support of this project under grant DUE-0410546, and most of all, to our students.

Some things to look for	Suggests understanding of:
√ Program compiles	Basic mechanics
√ Constructors defined and used	Constructors
√ Multiple instances of same class	Object and class
√ Variables/methods with same name in different classes	Encapsulation
√ Multiple classes defined in program √ Composite object (object with parts) defined √ Object passed as parameter to constructor (peer object) √ Peer object assigned to instance variable √ Methods other than constructors defined √ Message sent to part / peer object √ Methods' return values used	Linking; message passing; methods
√ Library classes extended √ User-defined classes extended √ Shared properties/methods factored into superclass	Inheritance
√ Single library class has multiple subclasses that define the same method differently √ Single user-defined class has multiple subclasses that define the same method differently	Polymorphism
√ Classes correspond to visible objects in domain √ Some class corresponds to invisible (conceptual) object	Modelling
√ Methods have parameters √ Method parameters used √ Inheritance hierarchies defined and used √ Simple recipes used √ Design patterns used	Abstraction

Table 1: Indications that a student understands basic OO concepts

Some things to look for	Suggests misconception:
√ Classes identical except for class names √ Classes identical except for property values that could be set in constructors √ Classes identical except for very minor changes √ Classes rarely/never instantiated more than once √ Superclass/subclass used instead of class/instance	Instance/class conflation (Holland)
√ All code in a single class √ Code in single class instead of composite class and parts √ Classes defined but not linked in √ Work in methods exclusively done by assignment √ Objects passed as parameters but not used	Problems with linking and interaction (Thomasson et al.)
√ Class whose instances all model elements of some collection	Class / collection conflation (Thomasson et al.)
√ Classes identical except for property values that could be set in constructors √ Duplicate code not factored into superclass √ Duplicate method signatures in different classes not defined as interface	Problems with abstraction (Or-Bach & Lavy)
√ Classes joined that should be separate or vice versa √ Classes do not correspond to objects in domain √ Failure to use inheritance to model hierarchical domain	Problems with modelling (Thomasson et al.; Eckerdal & Thuné)
√ Failure to delete irrelevant code when adapting	Classes just text (Eckerdal & Thuné)
√ Variables with names that are really values of attributes	Identity/attribute conflation (Holland et al.)
√ No classes with methods other than constructors or accessor/mutators (or `main`)	Objects are only data records (Holland et al.)
√ Methods / variables in different classes always have different names	Problems with encapsulation (Fleury)

Table 2: Indications that a student has one of the OO misconceptions found in the literature