Object Centered Design for Java: Teaching OOD in CS-1

Joel Adams **Jeremy Frens**
Department of Computer Science
Calvin College
Grand Rapids, MI 49546
{adams, jdfrens}@calvin.edu

Abstract

Object-centered design (OCD) is a methodology developed to help novice C++ programmers learn to design software. By adapting OCD for use with Java, we can reduce the number of phases in OCD from five to three, and introduce object-oriented design (OOD) in CS-1 instead of CS-2.

Categories & Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods.*

General Terms

Design, Languages.

Keywords

CS 1/2, Object-Oriented Issues, Software Engineering, Programming Languages and Paradigms, Pedagogy.

1 Introduction

Yogi Berra supposedly once said something like,

> *If you don't know where you're going,*
> *you might not get there.*

So it is with novice programmers trying to write software. Without a path to a solution, they may end up lost.

Teaching CS-1 students a software design methodology has drawn mixed reviews. Some claim that teaching design early is harmful [6]. However far more seem to be teaching some aspect of design, whether the focus is on teaching students to use design patterns [7][9], teaching them the consequences of bad algorithm design [10], or gradually introducing students to object-oriented design over CS-1 and CS-2 [1].

SIGCSE'03, February 19-23, 2003, Reno, Nevada, USA.

We are firm believers in the benefits of teaching object-oriented programming to our students. By doing so, our students learn to design software that is easier to code, easier to debug, and therefore less costly to maintain over an application's lifetime.

The gradual approach to object-oriented design described in [1] is called *object centered design* (OCD). OCD begins with a simple version of Booch's [4] approach to object-oriented design (OOD) that eliminates the "Where do I begin?" feeling that afflicts many novice programming students. Original OCD incorporates new programming concepts (e.g., classes and methods, inheritance) into the methodology over five graduated phases. By the final phase (in CS-2), students are using OOD.

The original version of OCD was developed in 1996 using the features of C++, a hybrid language. In this paper, we adapt OCD for use with Java in CS-1. The use of Java reduces the number of phases in OCD to three, providing students with a full introduction to OOD in CS-1.

2 Definitions

For clarity, we begin by defining our terminology.

When we use the phrase *object-oriented design*, we mean a software design methodology that

a. represents a problem's nontrivial objects using *classes*;
b. uses *inheritance* to consolidate attributes common to multiple classes into superclasses;
c. uses *polymorphism* to ensure that a message sent to an object elicits the appropriate behavior; and
d. produces an application that, without modification, can support *multiple user interfaces*.

When we use the phrase *object-oriented programming* (OOP), we mean the act of implementing an object-oriented design in some programming language. That is, teaching students to program in Java is no guarantee that we are teaching them object-oriented programming. As one wag has said,

> *I can write Fortran in any language!*

and Java is no exception. We are being dishonest with our students if we claim to be teaching them object-oriented programming and then teach them to design what are in essence Fortran programs expressed in Java syntax, as shown in Figure 1:

```
class Fahr2CelsiusConverter {
 public static void main(String [] args) {
   double fahr = Double.parseDouble(args[0]);
   double cels =  (fahr - 32.0) / 1.8;
   System.out.println(" --> " + fahr +
                      "F = " + cels + "C");
 }
}
```

Figure 1. A Fortran Program in Java Syntax

It is our objective that our students leave CS-1 with an understanding of object-oriented design and programming.

But how do we teach object-oriented design before students have learned about inheritance, polymorphism, and so on? The answer is that we cannot; instead we must begin where our students are and provide a design methodology that uses concepts with which they are familiar. Because it is a simplification of original OCD, we call this methodology **object centered design for Java**.

3 Object Centered Design for Java

3.1. *Phase I.* "Objects" Early.

Within the first week of CS-1, we teach our students to solve a problem using the following methodology:

a. Write a moderately detailed paragraph describing what the program is supposed to do, using words like *program*, *keyboard*, *screen*, etc.

b. Identify the nouns within your paragraph. These are the objects your program must declare. If there are any objects that cannot be directly represented using existing types, define a class to represent such objects.

c. Identify the verbs in your paragraph. These are your operations.

d. Apply the operations from step (c) to the objects from step (b) in a way that solves the problem.

Any student who can distinguish a noun from a verb can use this methodology to design their first program. This is important, as it effectively eliminates the *"Where do I begin?"* syndrome that many novice programmers experience when they are not taught how to design their programs.

In the first few weeks, we discuss types, expressions, primitive types, reference types (classes), and talk about operators and messages. In these early days, we assign simple problems that can be solved primarily using Java's primitive types or standard classes (e.g., `String`, `Integer`, `Double`). We provide our own `Keyboard` and `Screen` classes, to simplify Java's cumbersome I/O system.

Since there is no type available to represent the noun *program*, we instruct our students to follow the preceding methodology and build a class to represent their program. Classes are thus introduced early, though program classes are the only ones we build at the outset. At this point, we expect our students to write programs like that in Figure 2:

```
class Fahr2CelsiusConverter {
 public static void main(String [] args) {
   Keyboard kbd = new Keyboard();
   Screen scr = new Screen();
   scr.print("Enter a Fahr. Temperature: ");
   double fahr = kbd.readDouble();
   double cels =  (fahr - 32.0) / 1.8;
   scr.println(" --> " + fahr +
               "F = " + cels + "C");
 }
}
```

Figure 2. A First Program

Such a program is far from object-oriented in its design – in fact, it's not all that different from that of Figure 1 – but it does let us introduce several fundamental OO concepts, and we have started with concepts our students understand: *nouns* and *verbs*.

In this first phase, we deliberately blur the distinction between class instantiations and primitive type variables – we call both of these "objects". For the first few weeks of the semester this is convenient; later in the semester when students have built their own classes, this simplification is easily corrected.

We *do* assign problems besides temperature conversion. We merely use this problem here to illustrate our progression.

3.2. *Phase II.* Classes and Methods

In the fourth week of the semester, we begin teaching about methods. We start with simple formulaic class methods, motivated by the idea of making expressions reuseable. By posing problems whose solutions require the use of `if` statements and/or loops, we introduce control structures as a means of eliciting more complex method behavior.

When students have seen several examples of class methods, we make the jump to instance methods and variables. Default-value constructors, accessors, and converters are covered at this point as methods that make use of instance variables.

The problem of validating the arguments passed to an explicit-value constructor and/or mutator provides an excellent motivation for using the `if` statement.

To accommodate this new material, we revise our design methodology as follows:

a. Write a moderately detailed paragraph describing what the program is supposed to do, using words like program, keyboard, screen, etc.

b. Identify the nouns within your paragraph. These are the objects your program must declare. If there are any objects that cannot be directly represented using existing types, define a class to represent such objects.

c. Identify the verbs in your paragraph. These are your operations. *If an operation is not predefined,*
   1) *Define a method to perform that operation.*
   2) *Store it in the class responsible for the operation.*

d. Apply the operations from step (c) to the objects from step (b) in a way that solves the problem.

With this methodology and some guidance, our students can now create a class that represents a real-world entity (e.g., a temperature), rather than just a class that represents their program. Students have reached the stage known as *object-based* programming [3], which is one step closer to our goal of object-oriented programming. At this stage, we would expect our students to design a program like that in Figure 3:

```
class Fahr2CelsiusConverter {
 public static void main(String [] args) {
   Keyboard kbd = new Keyboard();
   Screen scr = new Screen();
   scr.print("Enter a Fahr. Temperature: ");
   Temperature fTemp = new Temperature();
   fTemp.read(kbd);
   Temperature cTemp = fTemp.toCelsius();
   fTemp.write(scr);
   scr.print(" == ");
   cTemp.write(scr);
   scr.println();
 }
}
```

Figure 3. An Object-Based Program

We would also expect them to build any classes on which such a program relies, such as the (fragmentary) class shown in Figure 4:

```
class Temperature {
 public Temperature() { ... }
 public Temperature(double mag, char scale)
 { ... }

 public double getMagnitude() { ... }
 public char getScale() { ... }

 public Temperature toCelsius() { ... }
 public Temperature toFahrenheit() { ... }
 public Temperature toKelvin() { ... }

 public void write(Screen scr) { ... }
 public void read(Keyboard kbd) { ... }

 public String toString() { ... }

 private double myMagnitude;
 private char   myScale;
}
```

Figure 4. A `Temperature` Class

During the latter part of this phase, we introduce problems that can be solved more conveniently using the remaining control structures (e.g., `switch` statement, `do` loop). Students are expected to use these in their assignments; thus the `toCelsius()` method listed in Figure 4 might be implemented something like what is shown in Figure 5:

```
public Temperature toCelsius() {
 Temperature r = null;
 double m = myMagnitude;
 switch (myScale) {
  case 'C':
    r = new Temperature(m, 'C');
    break;
  case 'F':
    r = new Temperature((m-32.0)/1.8, 'C');
    break;
  case 'K':
    r = new Temperature(m-273.15, 'C');
    break;
  default:
    System.err.println("toCelsius(): " +
      "myScale is invalid: " + myScale);
 }
 return r;
}
```

Figure 5. The `toCelsius()` Method

During this time, we also introduce arrays and files, and incorporate these into student assignments.

3.3. *Phase III.* Inheritance

In week ten or eleven, we introduce inheritance as a means of relating two classes using the *is-a* relationship, and emphasize that a class B is-a specialization of class A if and only if all messages that can be sent to an instance of A can be appropriately sent to an instance of B. We use simple UML class-structure diagrams like that shown in Figure 6 to help students visualize such relationships:

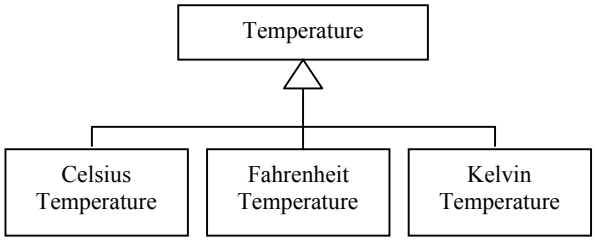

Figure 6. Introducing Inheritance

Discussing what happens when an inherited method is redefined by a subclass leads naturally to a discussion of polymorphism.

For example, we highlight how polymorphism can be used to eliminate `if` and `switch` statements whose sole purpose is to distinguish between peer objects. To illustrate, compare the `toCelsius()` methods in Figure 7 with that shown in Figure 5:

```
abstract class Temperature {
 ...
 public double getMagnitude() {
  return myMagnitude;
 }

 abstract public char getScale();
 abstract public Temperature toCelsius();
 ...
 private double myMagnitude;
}
//-------------------------------------
class FahrenheitTemperature
      extends Temperature {
 ...
 public Temperature toCelsius() {
  return new CelsiusTemperature(
        (this.getMagnitude()-32.0)/1.8 );
 }

 public char getScale() {
  return 'F';
 }
}
//-------------------------------------
class CelsiusTemperature
      extends Temperature {

 public CelsiusTemperature(double m) {
  super(m);
 }
 ...
 public Temperature toCelsius() {
  return new CelsiusTemperature(
        this.getMagnitude() );
 }

 public char getScale() {
  return 'C';
 }
 ...
}
```

Figure 7. A Temperature Class Hierarchy

Using examples like this, students see how investing time in one's design can simplify the methods one must write.

To encourage our students to think in this manner, we modify our design methodology one last time:

a. Write a moderately detailed paragraph describing what the program is supposed to do, using words like program, keyboard, screen, etc.
b. Identify the nouns within your paragraph. These are the objects your program must declare.
   1) If there are any objects that cannot be directly represented using existing types, define a class to represent such objects.
   2) *If two or more classes have common attributes, consolidate those attributes into a superclass, and derive those classes from the superclass.*
c. Identify the verbs in your paragraph. These are your operations. If an operation is not predefined,
   1) Define a method to perform that operation.
   2) Store it in the class responsible for the operation.
   3) *Where necessary, have subclasses override inherited definitions.*
   4) *If a class is responsible for an operation but can not implement it, declare an abstract method.*
d. Apply the operations from step (c) to the objects from step (b) in a way that solves the problem.

This phase combines phases III and V from original OCD. Original OCD delayed polymorphism until after the introduction of pointers in CS-2, because polymorphism requires pointers. Java makes it possible to introduce polymorphism in CS-1 because – unlike C++ – no new syntax is required to use pointers in Java.

Using this methodology, we give our students an assignment that requires them to build a class hierarchy like that in Figure 7, and write a program that uses their hierarchy to solve a problem, like that in Figure 8:

```
class Fahr2CelsiusConverter {
 public static void main(String [] args) {
   Keyboard kbd = new Keyboard();
   Screen scr = new Screen();
   scr.print("Enter a Fahr. temperature: ");
   Temperature fTemp =
                new FahrenheitTemperature();
   fTemp.read(kbd);
   Temperature cTemp = fTemp.toCelsius();
   fTemp.write(scr);
   scr.print(" == ");
   cTemp.write(scr);
   scr.println();
 }
}
```

Figure 8. An Object-Oriented Temperature Converter

Once students have experience implementing an object-oriented design, advanced topics like graphical user interfaces, multithreading, or networking can be introduced during the final few weeks of the semester.

By the end of CS-1, the students are using object-oriented design. While they are not yet masters of it, our students leave CS-1 with a working knowledge of OOP.

4 Observations

4.1. Original OCD vs. OCD for Java

The original object-centered design methodology [1] (with C++) required five phases spread over CS-1 and CS-2. Object-centered design for Java requires just three phases, all of which can be covered in CS-1. This is primarily because Java is a "more pure" object-oriented language that is closer in philosophy to Smalltalk than C++. Like Smalltalk and unlike C++:

- *All Java subprograms are methods*, meaning they must be associated with a class; thus classes must be introduced before subprograms can be covered.
- *Java reference-type variables are implicitly pointers, and Java methods are by default polymorphic*, so no new syntax is required to teach polymorphism.

These and other differences make it possible (using Java) to introduce programming novices to OOD in a single course. Providing students with an understanding of inheritance and polymorphism in CS-1 means those topics need not be taught from scratch in CS-2. This permits CS-2 to focus on data structures, where the OOP concepts introduced in CS-1 can be reused and reinforced.

Java also makes introducing exceptions easier, but this is orthogonal to OCD and beyond the scope of this paper.

4.2. Anecdotal Benefits of Using OCD with Java

We have not conducted a scientific study of the benefits of using object-centered design with Java. However we have noticed that in teaching students to use this methodology, the number of students who visit during office hours and say, "I don't know where to start!" has dwindled to one or two per semester, usually on the first or second assignment.

Interestingly, such students are usually lost because they are not using the methodology. (Perhaps they were absent the day it was introduced.) Once we walk the student through its use and show how the solution to a problem comes from following the methodology's steps, the student leaves and does not need further help. It helps to have a textbook that uses the methodology consistently [2].

4.3. Phase IV

Phase IV of the original OCD introduced students to C++ templates in order to build container classes. In Java, container classes store references to `Objects`, using inheritance instead of syntax. Consequently, containers can be easily incorporated into the new third phase, by training students to use the broadest possible base class.

Accessing an object from such a container requires us to introduce students to type-casting, but this is much simpler than learning to build C++ templates.

It does appear that some template-like feature will be added to Java as proposed in Java Specification Request 014 [5][8]. When this happens, we will reintroduce Phase IV in CS-2. However container classes are orthogonal to OOD (i.e., phase IV was a convenience, rather than a necessity in original OCD). Thus, teaching students to build parameterized container classes in CS-2 will not diminish our ability to introduce them to OOD in CS-1.

5 Conclusion

Object-centered design for Java is a software design methodology that, starting with ideas that are ascertainable by novice programmers, builds gradually and naturally as object-oriented topics are introduced, until it culminates in object-oriented design. OCD for Java makes it possible to introduce students to OOD in CS-1. It also seems to eliminate the "I don't know where to start" syndrome that can afflict novice programmers.

When a student's goal is to write software that solves a problem, object-centered design for Java is a methodology that helps them see where they are going, and thus helps them "get there."

References

[1] Adams, J., Object-Centered Design: A Five Phase Introduction to Object-Oriented Programming in CS1-2, *Proceedings of the 27nd SIGCSE* (Mar 1996), ACM Press, pp 78-82.

[2] Adams, J., Nyhoff, J., Nyhoff, L., *Java An Introduction to Computing*, Prentice Hall, 2001.

[3] Astrachan, O., Using Classes Early, An Object-Based Approach to Using C++ in Introductory Courses, *Proceedings of the 29th SIGCSE* (Feb 1998), ACM Press, pp 383.

[4] Booch, G., Object Oriented Development, *IEEE Transactions on Software Engineering*, 12(2) Feb 1986, pp 211-221.

[5] Bracha, Gilad et al. Java Specification Request 014: Add Generic Types of the Java Programming Language. http://jcp.org/jsr/detail/014.jsp.

[6] Buck, D. and Stucki, D., Design Early Considered Harmful: Graduated Exposure to Complexity and Structure Based on Levels of Cognitive Development, *Proceedings of the 31st SIGCSE* (Feb 2000), ACM Press, pp 75-79.

[7] Proulx, V., Programming Patterns and Design Patterns in the Introductory Computer Science Course, *Proceedings of the 31st SIGCSE* (Feb 2000), ACM Press, pp 80-84.

[8] Sun Microsystems. Prototype for JSR014: Adding Generics to the Java Programming Language. http://developer.java.sun.com/developer/earlyAccess/adding_generics/, requires free registration.

[9] Wallingford, E., Toward a First Course Based on Design Patterns, *Proceedings of the 27th SIGCSE* (Feb, 1996), ACM Press, pp. 27-31.

[10] Wolz, U., Teaching Design and Project Management with Lego RCX Robots, *Proceedings of the 32nd SIGCSE* (Feb 2001), ACM Press, pp 95-99.