OBJECT-CENTERED DESIGN

A FIVE-PHASE INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING IN CS1-2

Joel C. Adams
Department of Mathematics and Computer Science
Calvin College
Grand Rapids, MI 49546
adams@calvin.edu

ABSTRACT

With Pascal waning in popularity as the CS1 language of choice, many colleges and universities are considering the adoption of C++ (an imperative and object-oriented hybrid language) as its replacement. An important issue that must be addressed in making such a change is the question of what software design methodology should be taught to CS1 students. Two common answers are (i) continue teaching structured design in CS1 and switch to object-oriented design in CS2; or (ii) teach object-oriented design from the outset in CS1. We believe that both of these approaches have significant drawbacks. To avoid these drawbacks, this paper describes a graduated approach to object-oriented design that we call *object-centered design.* The approach introduces students to object-oriented design by the end of CS2 without an abrupt paradigm shift, and without requiring an early introduction of inheritance.

INTRODUCTION

It is said that the phrase

May you live in interesting times.

is an old Jewish curse. The Computer Science Education literature of recent years indicates that these are "interesting times" to be teaching CS1-2. Established practices are being challenged (Pattis93, Roberts95), new approaches are being tried (Astrachan95, Tucker94), object-oriented design is being embraced (Decker&Hirshfield93, Reek95), and there would appear to be a significant movement away from Pascal to C++ (Decker&Hirschfield94, Hitz&Hudec95, Reid93, Wick95).

Unlike Smalltalk (a *pure* object-oriented language), C++ is a *hybrid* language, combining the imperative features of its parent C with the class construct of Simula. That is, where one must employ object-oriented design in Smalltalk, doing so is *optional* in C++. This means that colleges and universities switching (or considering a switch) to C++ must make a decision as to *when* object-oriented design (OOD) is taught. Options include:

1. Teach *OOD early*, at the beginning of CS1.
2. Teach *OOD late*, deferring it until CS2 (or later).
3. Adopt some intermediate approach between 1 and 2.

This paper argues in favor of option 3, and presents one such approach.

A PLEA FOR PRECISE SPEECH

Let us first define what object-oriented design is, and in so doing, define what it is not. (Stroustrup92) identifies and defines the following approaches to software design:

- *Procedural*: Decide which procedures you want; use the best algorithms you can find.
- *Modular*: Decide which modules you want; partition the program so that the data is hidden in modules (sic).
- *Data Abstraction*: Decide which types you want; provide a full set of operations for each type.
- *Object Oriented*: Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance.

These definitions imply that *classes and inheritance are the defining characteristics of object-oriented design*. That is, simply switching from Pascal to C++ does not make CS1 object-oriented, since the hybrid nature of C++ permits any of the above approaches to be taught. If we are unwilling to teach CS1 students to design their software with classes and inheritance, then we should not describe what we are teaching them as object-oriented design. To that end, this paper describes object-*centered* design — an approach that focuses on objects (without inheritance) at the outset, and expands gradually, culminating in object-oriented design.

THE PROBLEMS WITH OOD-EARLY AND OOD-LATE

The OOD-early and OOD-late approaches have different drawbacks.

OOD-Early. The preceding definition of object-oriented design creates a problem for those wishing to teach OOD *at the outset* of CS1, since a proper understanding of inheritance requires a firm understanding of classes and function members (methods); a proper understanding of classes requires a firm understanding of types; and a proper understanding of function members requires a firm understanding of functions. Few CS1 students understand types and functions at the outset of CS1, much less classes or inheritance. These dependencies are shown in Figure 1:

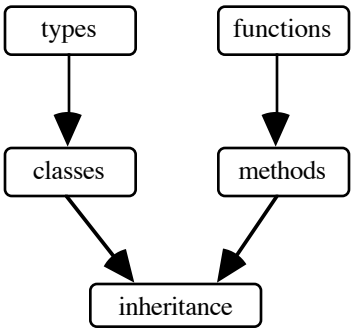

Figure 1
Dependencies Between OOD Components

Put differently, CS1 novices do not have the cognitive framework to grasp the concepts underlying object-oriented design, because they have no experience dealing with types and functions, much less classes, function members or inheritance.

This is not to say that teaching students to *use* classes, function members and inheritance early in CS1 is impossible; but teaching them object-oriented design without experience using classes and inheritance is problematic.

OOD-Late. The OOD-late approach is also problematic in CS1, since it means that students must later "unlearn" the techniques they were taught in CS1 when they are expected to learn OOD (Guzdial95). The difficulty of this "paradigm shift" for students should not be underestimated; and if OOD is what we want our students to eventually use, it is hard to justify teaching them structured design at the outset, even if doing so makes life easier for the CS1 faculty (Decker&Hirschfield94).

Beyond the questionable pedagogy of teaching our students material that they will later be required to unlearn, the OOD-late approach adds significantly to the conceptual difficulty of CS2, which is already (at least in our curriculum) one of the courses students find most difficult.

AN INTERMEDIATE APPROACH

Because of these drawbacks, we believe that an intermediate approach is to be preferred. In developing this approach, our guidelines were that the approach should:

A. generalize naturally to OOD, to avoid the "paradigm shift" of the OOD-late approach; and
B. be teachable from the beginning of the CS1 course.

Because of the above-mentioned problems associated with OOD-early, that approach is ruled out by guideline B; but guideline A suggests that perhaps a simplified version of OOD might work. We have combined these observations into a design methodology consisting of five graduated phases, culminating in OOD. For lack of a better phrase, we call this graduated methodology *object-centered design*, which is loosely based on what is commonly called the Booch approach to OOD (Booch91). By initially teaching students to associate the term *object* with each noun in a problem description, and the term *operation* with each verb, students can apply this methodology at the outset of CS1, satisfying guideline B. By focusing on objects at the outset and gradually generalizing toward OOD, this approach also satisfied guideline A.

CS1: PHASES 1-3

Phase I. *Objects Early*. At the outset of CS1, we teach our students to design software using the following methodology:

1. Identify the objects within your problem.
2. Identify the operations on the objects listed in 1 that are needed to solve the problem.
3. Order the operations in 2 into a problem solution.
4. Implement your solution.

In this approach, *objects need not be class instantiations*. By examining relatively simple problems at the outset of CS1, the objects can be represented using the C++ basic types: integers, reals, characters, and booleans. That is, we deliberately constrain the problem-space at the outset in order to introduce the students to object-centered design using objects of the fundamental types and the operations that can be applied to objects of those types. Doing so gives our students experience successfully applying this software design to problems, building confidence.

The only non-fundamental type objects introduced during this phase are C++ standard library objects such as `cin` and `cout` for interactive I/O. We use our discussion of these objects to introduce the terms *class* and *class object*, and use them from this point on to begin building the student's vocabulary.

Phase II. *Functions Early*. As the means by which an object's behavior is implemented, it is important for students to grasp the central role that functions play in OOD. For those not familiar with object-oriented programming, consider the main function from a Turbo C++ Windows application shown in Figure 2:

```
#include "SampApp.h"

int
PASCAL WinMain(HANDLE NewEnv, HANDLE OldEnv,
               LPSTR CmdLn, int CmdShow)
{
  SampleApp MyApp("Example", NewEnv, OldEnv,
                             CmdLn, CmdShow);
  MyApp.Run();
  return MyApp.Status;
}
```

Figure 2
A Turbo C++ Windows Application Main Function

This main function contains no control structures, but two function calls: one to the `MyApp` constructor which creates the application, and one to the `MyApp` function member `Run()`. All control structures (such as the event-handling loop) and other details are hidden within the `MyApp` function members.

This style might seem unusual to those used to structured programming, but its structure is completely typical of a Windows application. Further, it illustrates the roles functions and control structures play in such applications. Where control structures often dominate the main function in a structured design, functions typically dominate the main function of an object-oriented design: *control structures simply provide the means of achieving more sophisticated function behaviors*.

Since functions are the means of implementing behavior in OOD, we introduce them early in CS1 (weeks 2 and 3). We introduce functions in our discussion of the operations on the fundamental types by discussing header files and libraries of functions that the students can also use as operations on those types, such as those declared in stdlib.h, ctype.h, and math.h. Students thus learn about using libraries, calling functions and passing arguments as a natural extension of expressions.

In addition to "normal" functions, we use some of the `istream` *function members* (e.g., `cin.get()`) to introduce that concept and terminology, again laying the groundwork for later expansion.

We next teach our students to write simple parameterized formula-style functions, such as that shown in Figure 3:

```
double FeetToMeters(double Feet)
{
  return Feet * 0.3048;
}
```

Figure 3
A Simple Formula-style Function

This provides an early introduction to the basics of function-writing, especially parameters and return values.

Once our students have seen how to write parameterized functions, we expand the design methodology as follows:

1. Identify the objects within your problem.
2. Identify the operations on the objects listed in 1 that are needed to solve the problem. *If an operation is not provided, write a function to implement it.*
3. Order the operations in 2 into a problem solution.
4. Implement your solution.

As before, we have our students practice on problems whose objects can be modeled with the provided types, but for which some of the operations must be implemented by the students.

We have found that if we explain to students that wrapping a formula within a function makes that formula *easier to reuse*, they seem to have a greater appreciation for functions. This early introduction also reduces the amount of new material that must be mastered when we introduce reference parameters.

We then continue this theme of reusability by introducing header files and separate compilation, to allow students to write functions that can be used by multiple programs. As before, this is easy to motivate, since the *reusability* is something that will save the students work.

Once the students have a firm grasp of functions and separate compilation, we then introduce the control structures as a means of implementing increasingly sophisticated function behaviors. Reference parameters receive a similar treatment. *Students thus learn to apply these concepts in order to elicit a desired behavior from a function*, laying the groundwork for the introduction of function members.

For some projects, we provide off-the-shelf classes (e.g., `RandomInt`, `string`) that enable students to solve a more interesting problem than would be possible otherwise. Students use these classes as "black boxes" but are exposed to class-related concepts and terminology, such as *construction* and *destruction* of an object. By doing so, students gain working definitions of these terms, so that fewer new concepts must be introduced when students begin to design their own classes.

Phase III. *Building Classes*. By week 8 in the course, we are ready to begin building upon the groundwork laid earlier. To introduce the idea of programmer-defined types and operations, we use with a simple enumeration (e.g., `Weekday`), which we store in a header file and for which we implement I/O operations.

In week 9, students begin to build their own classes. We model this for them by designing and implementing a small class (e.g., `Temperature`), to show the syntax behind the ideas of classes, member functions and constructors, which they have been using for some time.

We then expand the design methodology again:

1. Identify the objects within your problem.
1a. If an object cannot be modeled using a provided type, build a class that allows such an object to be modeled.
1b. List the desired behaviors for each class object, and implement each behavior as a function member.
2. Identify the operations on the objects listed in 1 that are needed to solve the problem. If an operation is not provided, write a function to implement it.
3. Order the operations in 2 into a problem solution.
4. Implement your solution.

While this expanded approach requires students to make a significant transition, our graduated methodology makes it less difficult for them (than a structured-to-OOD transition), since they are already thinking in terms of objects and operations. By spreading the OOD ideas across five phases, students have to master fewer new ideas in any given phase.

In the final weeks of CS1, we provide an off-the-shelf `vector` template, introducing the concept and use of parameterized classes. Our students thus leave CS1 knowing how to create classes and function members, and how to use off-the-shelf templates.

CS2: PHASES IV & V

We begin CS2 with a one-to-two week review, to reinforce the final topics from CS1, and to bring transfer and AP students to the same place as students that have completed our CS1 course.

Phase IV. *Container Classes*. This phase takes several weeks, beginning with an introduction to pointers, followed by the introduction of the basic data structures at a conceptual level. Pointers are then used to implement several of the data structures, which are "parameterized" using the `typedef` mechanism. We then teach students to replace the `typedef` statement with the necessary syntax to convert the data structure into a template. We then (again) expand the software design methodology:

1. Identify the objects within your problem.
1a. If an object cannot be modeled using a provided type, build a class that allows such an object to be modeled. *If a class is for storing objects, make it a template.*
1b. List the desired behaviors for each class object, and implement each behavior as a function member.
2. Identify the operations on the objects listed in 1 that are needed to solve the problem. If an operation is not provided, write a function to implement it.
3. Order the operations in 2 into a problem solution.
4. Implement your solution.

As before, we select problems for the students to solve whose objects provide practice using this expanded methodology.

Phase V. *Object-oriented Design*. In the last few weeks of CS2, we reach the culmination of the CS1-2 sequence, and introduce the topics of inheritance and polymorphism. In so doing, we make the final transition in our design methodology, and introduce the students to OOD. As usual, we start with an example, and then once the students have seen inheritance in use, we expand their design methodology a final time:

1. Identify the objects within your problem.
1a. If an object cannot be modeled using a provided type, build a class that allows such an object to be modeled. If a class is for storing objects, make it a template. *If two classes have something significant in common, isolate their commonality in a base class.*
1b. List the desired behaviors for each class object, and implement each behavior as a function member.
2. Identify the operations on the objects listed in 1 that are needed to solve the problem. If an operation is not provided, write a function to implement that operation.
3. Order the operations in 2 into a problem solution.
4. Implement your solution.

The remainder of the course provides the students with practice using the new ideas of inheritance and polymorphism.

OBSERVATIONS AND CONCLUSIONS

We have described a software design methodology that, starting with ideas ascertainable by novice programmers, builds gradually and naturally, culminating in the object-oriented design methodology. Just as a person learns to swim by gradually gaining expertise in the water (as opposed to being thrown in the deep end), we believe that the easiest way to learn OOD is by gradually gaining expertise in the use of classes, inheritance, and the classes in a class library. We believe that spreading this process over the CS1-2 sequence solves one of the problems facing colleges and universities switching to C++. However, additional issues remain, some of which we discuss next.

One issue concerns the adoption of the Standard Template Library (STL) for inclusion in the C++ language standard by ANSI Committee X3J16 . With this adoption, decisions must be made regarding the integration of STL into the CS1-2 syllabus. As indicated by its name, STL provides templates for many of the standard data structures. Just as the calculator raises questions of how arithmetic is best studied , STL raises questions of how data structures are best studied. In a *top-down approach*, students could be taught to use STL early (possibly in CS1) and then study the implementation of objects from the library in CS2. In a *bottom-up approach*, students could be taught about pointers and how to implement linked structures in the traditional way, and then learn about STL once they have mastered these topics. Which (if either) approach is "better" remains to be seen, as implementations become available.

A related issue is the extensive use of *iterator classes* in STL. Use of these classes involves a significant shift in the way data structures are processed, and adds yet another new topic to a crowded syllabus. Perhaps more problematic is that they represent another new idea that faculty must master in order to teach their students to fully utilize STL.

Another STL-related issue is the role of exceptions and exception-handling, and the extent to which they must/should be addressed in the CS1-2 sequence. STL makes extensive use of exceptions, and so it would seem that some treatment is needed in order to use it effectively.

Finally, consider that as a small, simple language, Pascal made it possible to cover all of the language features and give an in-depth introduction to data structures in two courses. In our experience, the much greater size of C++ makes this impractical. Either some of the language features, or some of the course topics (or both) must be jettisoned. We have chosen a middle road, deciding to ignore some significant language features (e.g., multiple inheritance, exceptions) and leave in-depth coverage of some data structures (e.g., trees) for a later course. One colleague has suggested that perhaps a CS1-2-3 course sequence is needed. Perhaps colleagues from the Ada community could provide insight into how they resolve this issue in their introductory sequence.

Those teaching CS1-2 live in interesting times indeed. Before the times become any less interesting, these and other new issues facing teachers of CS1-2 must be resolved. We look forward with anticipation to their resolution in the years ahead.

REFERENCES

(Atrachan95) Owen Astrachan, *AAA and CS1: The Applied Apprenticeship Approach to CS 1*, SIGCSE Bulletin, 27(1): 1-5, March, 1995.

(Booch91) Grady Booch, Object Oriented Design, Benjamin/Cummings, 1991.

(Decker&Hirschfield93) Rick Decker and Stuart Hirschfield, *Top Down Teaching: Object-Oriented Programming in CS1*, SIGCSE Bulletin, 25(1): 270-273, March, 1993.

(Decker&Hirschfield94) Rick Decker and Stuart Hirschfield, *The Top 10 Reasons Why Object-Oriented Programming Can't be Taught in CS1*, SIGCSE Bulletin, 26(1): 51-5, March, 1994.

(Guzdial95) Mark Guzdial, *Centralized Mindset: A Student Problem with Object-Oriented Programming*, SIGCSE Bulletin, 27(1): 182-185, March, 1995.

(Hitz&Hudec95) Martin Hitz and Marcus Hudec, *Modula 2 versus C++ as a First Programming Language - Some Empirical Results*, SIGCSE Bulletin, 27(1): 317-321, March, 1995.

(Pattis93) Richard E. Pattis, *The "Procedures Early" Approach in CS 1: A Heresy*, SIGCSE Bulletin, 25(1): 122-6, March, 1993.

(Reek95) Margaret M. Reek, *A Top-down Approach to Teaching Programming*, SIGCSE Bulletin, 27(1): 6-9, March, 1995.

(Reid93) Richard J. Reid, *The Object Oriented Paradigm in CS 1*, SIGCSE Bulletin, 25(1): 265-9, March, 1993.

(Roberts95) Eric S. Roberts, *Loop Exits and Structured Programming: Reopening the Debate*, SIGCSE Bulletin, 27(1): 268-72, March, 1995.

(Stroustrup95) Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Publishing Company, 1992, pp. 14-22.

(Tucker94) Allen B. Tucker, *New Directions in the Introductory Computer Science Curriculum*, SIGCSE Bulletin, 26(1): 11-15, March, 1994.

(Wick95) Michael Wick, *On Using C++ and Object-Orientation in CS1*: The Message is Still More Important that the Medium, SIGCSE Bulletin, 27(1): 322-6, March, 1995.