Practice What You Preach: Full Separation of Concerns in CS1/CS2

Hamzeh Roumani
Department of Computer Science and Engineering
York University
Toronto, Ontario, M3J 1P3, Canada
roumani@cs.yorku.ca

ABSTRACT
We argue that the failure to separate the concerns in CS1 is the leading cause of difficulty in teaching OOP in the first year. We show how the concerns can be detangled and present a detailed reorganization of contents for CS1/CS2 with CS1 exposing only the client view. We also report on our experience with this new pedagogy after three years of implementation at our institution.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Computer science education, Curriculum, Information Systems education.

General Terms
Design, Experimentation, Languages, Theory.

Keywords
Separation of concerns, component-based architecture client-view, encapsulation, API, object-based programming.

1. INTRODUCTION
Even though it has been over five years since many institutions moved their CS1 from Pascal (or a similar procedural language) to OOP, there is still a great deal of dissatisfaction with the move and with the high dropout rates that ensued. Generally speaking, students are finding OOP very complex and instructors are not pleased with the level of understanding attained at the end of CS2. The problem has undoubtedly several causes (rooted in the students, the instructors, the pedagogy, and OOP itself) but we believe that the main one stems from abandoning a long held principle in computer science: separation of concerns [3].

We discuss this principle in Section 2 and show that ignoring it does indeed make OOP overly complex. We therefore propose that the contents of CS1/CS2 be re-organized along the concern boundary, and we do so in sections 3 (for CS1) and 4 (for CS2). We also report in these two sections on our experience with this pedagogy after implementing it at our institution.

SIGCSE'06, March 1–5, 2006, Houston, Texas, USA.

This work was motivated by an earlier one [7], in which we alluded to the need for separating the concerns, and by the works of others [1,6], which ultimately lead to the same conclusion. The works of [4,5,9] call for a "components-first" pedagogy, which is somewhat similar to ours in premise, but different in the details. Specifically, our approach does not require any special framework or material to be acquired (we rely on the standard Java library); does not require a special specification language (we rely on the conventional, albeit less formal, Java API); and does not mix up the concerns in the same course (we spend the entire CS1 in the client view).

2. SEPARATION OF CONCERNS
Whether appearing as part of information hiding, as an outcome of encapsulation, or as a general abstraction tool, the main idea behind the separation of concerns principle is the recognition of two distinct roles: the client whose concern is the *what*, and the implementer whose concern is the *how*. The two concerns are disjoint except at the interface where some information (the API) is shared on a need-to-know basis.

As a pedagogy, this principle enables us to divide the space of knowledge into two regions with no dependency in between. Any concept in one region can be learned without knowing any of the concepts in the other region. Hence, by staging the topics so that the learning path does not cross regions, complexity is reduced.

Computer science educators preach this separation in courses on software design and software engineering, and they structure the curriculum around it in courses like networking and organization, but they take a cavalier attitude toward it in CS1/CS2. In fact, most CS1 textbooks serve to (unintentionally) blur the distinction between the client and the implementer concerns in the mind of the reader: they define formal parameters and arguments in the same sentence, cover `this` and `new` in the same section, and discuss `super` and polymorphism in the same chapter and often in an interleaved manner. A typical CS1 student is thus expected to learn these concepts (and similar concern-crossing ones such as arrays and collections) together—often in the same lecture. This learning path makes OOP overly complex and the end result is either high attrition rates or students who capture only the mechanism, i.e. the causal linkage between language constructs and behavior.

Moreover, these textbooks start by teaching the student how to write a class, not one with only a `main` method, but a full-blown instantiable class. Doing so from the very outset creates the impression that the implementer's role is superior to that of the client; and that in order to use something, one must first learn how it works. In our view, this will lead to students who will always be bottom-up thinkers, and who will likely tend to subordinate correctness, testing, and contracts to implementation.

The perceived complexity of OOP is often blamed on the objects-first pedagogy [2] but we see it as an entanglement along an orthogonal dimension: the concern. As such, we see no difference between introducing objects late or early (vis-à-vis complexity) as long as we use objects in one course and implement them in another. It may seem bizarre that a mere reordering of topics can reduce complexity but this is because very few people can shift abstraction levels, especially in CS1. Metaphorically speaking, we are trying to teach students about cars through a series of lectures each of which covers a single subject. In the lecture on steering, we talk about turn signals and the circuit connecting them to the flashing lights; about the steering wheel and how it causes the power steering fluid to amplify the torque before applying it on the axle. There is nothing inherently complex about any of these topics but if you have to think about spark plugs every time you accelerate, learning how to drive becomes complex. Detangling the concerns was not crucial in the Pascal days because the overall framework was simpler and amounts, in this metaphor, to replacing the car with a bike. In a bicycle, the absence of an encapsulating hood and the simplicity of what is "under the hood" make it reasonable to talk about pedaling and the rotation of the chain in the same sentence.

3. CS1: THE CLIENT VIEW

In CS1 we adopt the client view throughout. This means we only write main programs that use existing components. The main program consists of a `main` method and nothing else. In particular, it is essential that it does not contain other `static` methods (or else it degenerates into a procedural program). The size of the `main` method ranges from a few lines of code near the beginning of the course to about 40 near the end, i.e. the entire main class fits on one printed page. The `main` method can declare variables of any type (but not arrays); instantiate component classes and use their fields and methods; and employ control structures (selection, loops, and exception handling). The components can be selected from the standard library of the language or from any other source that provides an API.

3.1 CS1 Topics

We cover the topics shown in Figure 1 with each taking about a week. We use Java and adopt an objects-first paradigm with objects introduced fairly early (static features in Week 3 and non-static ones in Week 4). The topics may look like the ones found in most textbooks but they are not: they capture only the client's perspective. The topic in Week 4, for example, includes reading the API, locating the constructor, creating an instance, and then using its feature to solve the problem at hand. Similarly, the topic of Week 8 tells us how to identify aggregate classes from their APIs, not by looking at their implementations.

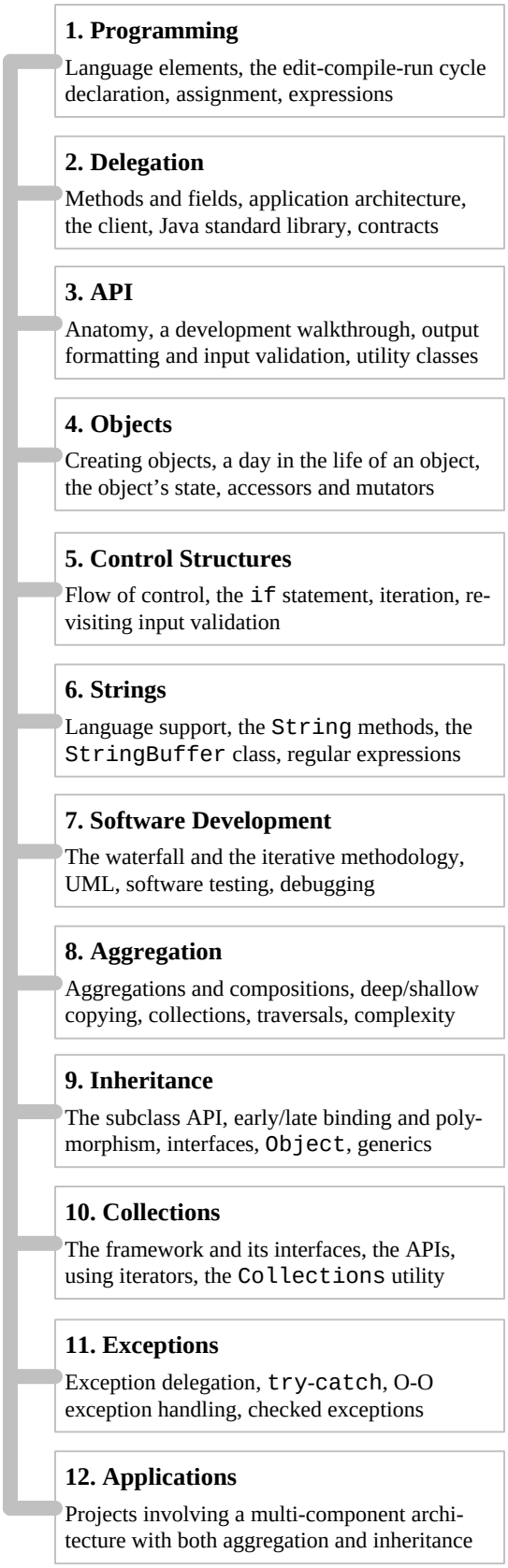

Figure 1. The weekly topics of CS1.

The figure does not include topics from areas such as SP (social and professional issues) since they are concern-neutral. The ordering of the shown topics and the emphasis placed on each were influenced by the textbook that we used [8] but one can shift the emphasis or introduce objects a bit earlier or later as needed. The selection of topics to include, however, is dictated by the requirement of main-only classes. Hence, topics such as recursion (which requires writing a method other than `main`) and callback (which requires writing an instantiable class and registering it as a listener) simply do not belong in a client-first CS1. Similarly, when unit testing is discussed in Week 7, it is necessarily black-box testing since we have no access to the code within components. In other words, by letting the client draw the line between what can and cannot be covered in CS1, we obtain a natural selection of topics in CS1/CS2 guaranteed to separate the concerns.

3.2 Possible Applications

In this subsection we list applications that our CS1 students can write at the end of each of the weeks shown in Figure 1. The list is not meant to be prescriptive in any way; we present it here merely to further clarify the client-view approach and to demonstrate that it is indeed possible to build non-trivial applications by writing only a `main` method.

1. **Primitive Types**
   Given a code fragment, determine the type and value of the expressions in it.

2. **Apply Integer Arithmetic**
   Invoke the `currentTimeMillis` method of the `System` class and compute an approximation to today's date from it.

3. **A Project**
   Write a program that reads the altitude of a satellite and outputs its period in hours, minutes, and seconds. Involves the `Math` class and `printf`.

4. **Explore an API**
   Read the API of the `Random` class in `java.util` and use it to create an instance and invoke the `nextInt` method on it several times. Why is this method overloaded?

5. **Infinite Series**
   Write a program that computes the sum, but with alternating signs, of the reciprocals of the odd numbers. Show that this sum converges to pi/4.

6. **Symmetric-Key Cryptography**
   Write a program that reads a string of letters and outputs it encrypted (done through an alphabet shift using a Vigenere-style keyword). Involves a single loop and the `String` class.

7. **HTML Scraping**
   Write a program that outputs the current temperature (or get a live stock quote) by querying a site. Involves the `URL` class, `Scanner`, and `StringTokenizer`.

8. **Working with Dates**
   Write a program to determine if the relationship between the `Calendar` and `Date` classes is a composition or not. Both classes are in `java.util`.

9. **Object Serialization**
   Read the API of the `ObjectOutputStream` class and use it to store a `Calendar` instance in a file. In a different main program, retrieve the instance and output its date.

10. **Working with Collections**
    Write a program that reads distinct integers and outputs their median. Do this first with and then without invoking the sort method of `Collections`.

11. **Socket Programming**
    Write two programs (preferably on different computers) that allow their users to chat. Uses the `Socket` and `ServerSocket` classes of `java.net`.

3.3 Our Experience

We moved our CS1 to Java using the conventional (i.e. mixed-concern) objects-first pedagogy back in 1999. We then shifted to the client-only approach presented herein in the fall of 2003. The course has been offered six times since then by six different instructors. All six instructors had taught the course before the shift, and hence, were in position to compare before/after results. The following points capture the main findings:

- The "initial shock" has been reduced. Whereas the old approach used to quickly alienate a large percentage of the students and lead them to drop in the first six weeks, the new one seems to engage almost all students early.
- The early dropout rate (occurring before the final exam) was cut at least in half. This result is the same regardless of who is teaching the course.
- The presentation of key concepts becomes sharply focused in the client-only view. In order to determine how a method behaves when its precondition is not met, or whether a method returns a deep or a shallow copy, you must write a client and experiment; you cannot peek at the implementation.
- The extension mechanism in Java makes it very easy for students to use instructor-created components. By bundling the components in a `jar` file and asking the students to store it in their `ext` directory, the components become accessible like any class in the standard library.

4. CS2: THE IMPLEMENTER VIEW

The topics we cover in CS2 are shown in Figure 2. The choice of topics for CS2 depends of course on whether the two-course or the three-course introductory sequence [2] is used. Nevertheless, the theme is the same: implementing components and covering implementer-only topics such as GUI and callback, recursion and sorting, and selected data structures.

There has been a significant difference in how students perceive the material of CS2 since CS1 was moved to the client view. For one thing, they are already familiar with the terminology and the fundamental OO concepts. Moreover, they are comfortable with the language constructs; can read APIs and use them; and, perhaps most importantly, can confidently deal with the compiler and the runtime error messages. With these concepts and skills in place, learning how to implement a class given its API appears as natural next step.

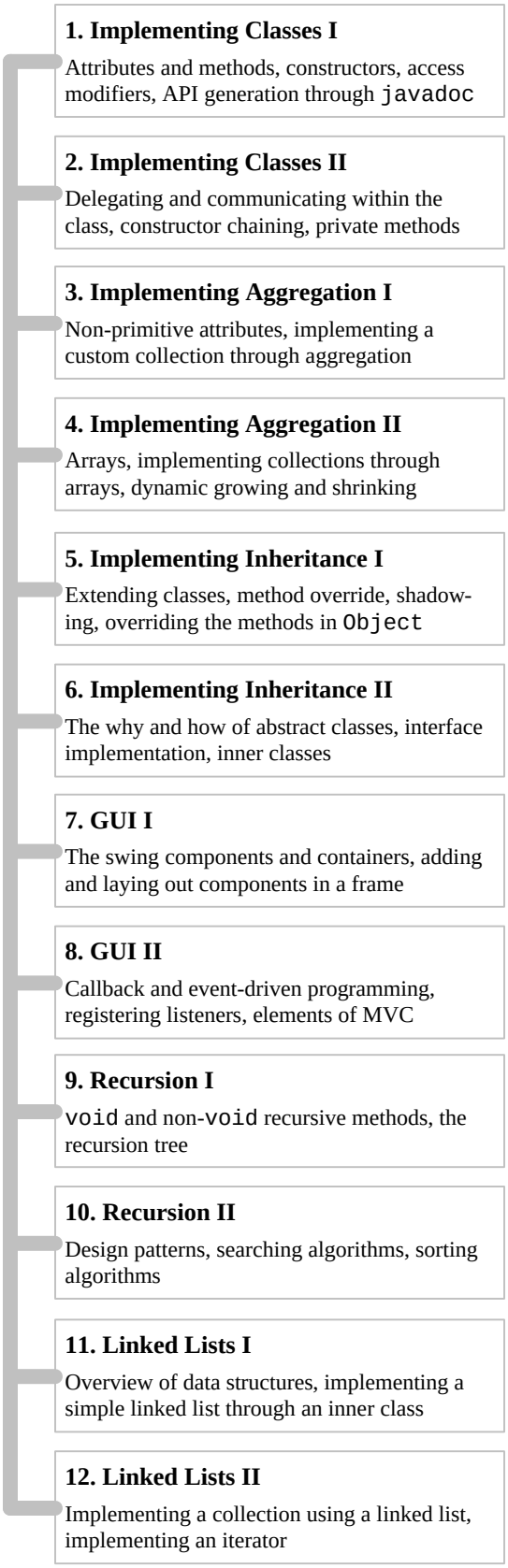

Figure 2. The weekly topics of CS2.

A second observed difference is that students continued to think like clients even after they learned how to implement. When we ask them to build an appointment book, for example, their first reaction is to extend or aggregate a `Map`, and use the methods in the `Calendar` class, rather than use arrays.

5. CONCLUSION

We have presented a new pedagogy for OOP in which objects are used in CS1 and implemented in CS2 (and CS3 if present). At first glance, it may seem wasteful (or shallow) to spend the entire CS1 playing the client role but we have shown that one can cover most of the key concepts and build elaborate applications without exposing any implementation. That one can teach CS1 like this without building a repertoire of components or a specialized IDE is possible thanks to the availability of numerous powerful components in the standard libraries of today's O-O languages.

We argued that this pedagogy makes CS1 seem easier and allows students to build interesting applications quickly. And having used components throughout CS1, students become ready, and even eager, to look "under the hood" in CS2. We also presented anecdotal evidence that supports these arguments.

6. REFERENCES

[1] Bucci, P., Long, T. and Weide, B. Do We Really Teach Abstraction? *In Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2001, 26-30.

[2] *Computing Curricula 2001, Final Report.* Joint IEEE-ACM Task Force. http:// www.sigcse.org/cc2001.

[3] Dijkstra, E. Note EWD447 (1974) reproduced in *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, Berlin, 1982. ISBN 0–387–90652–5.

[4] Howe, E., Thornton, M., and Weide, B. Components-First Approaches to CS1/CS2: Principles and Practice. *In Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 2004, 291-295.

[5] Koenig, A. and Moo, B. Rethinking How to Teach C++, Part 1: Goals and Principles. *Journal of Object Oriented Programming 13*, 7 (2000), 44-47

[6] Long, T., Weide, B., Bucci, P. and Sitaraman, M. Client View First: An Exodus From Implementation-Biased Teaching. *In Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, ACM Press, 1999, 136-140.

[7] Roumani, H. Design guidelines for the lab component of objects-first CS1. *SIGCSE*2002, 222-226.

[8] Roumani, H. *Java By Abstraction: A Client-View Approach.* Pearson Education Canada, Addison-Wesley, Toronto, Ont., 2006. ISBN 0-321-22689-5. URL: http://vig.pearsoned.ca /catalog/academic/product/0,1144,0321226895,00.html

[9] Sitaraman, M., Long, T., Weide, B., Harner, E. and Wang, L. A Formal Approach to Component-Based Software Engineering: Education and Evaluation. *In Proceedings of the ICSE International Conference on Software Engineering*, IEEE, 2001, 601-609.