Design Guidelines for the Lab Component of Objects-First CS1

Hamzeh Roumani
Department of Computer Science
York University
Toronto, Ontario, Canada
roumani@cs.yorku.ca

Abstract

We adopt the view that CS1 labs are not programming assignments, and that they should not be used for grading students or assessing their level of understanding. Instead, we think of them as teaching instruments that complement lectures by teaching the same material but in an exploratory fashion. But for labs to play this pedagogical role, certain conditions must be met in terms of how they are written and the complexity they expose. In this paper we present guidelines for designing the labs and for the Java packages that must accompany them, with special emphasis on software engineering. Our own experience with implementing these guidelines, together with a few samples, is included.

1 Introduction

Labs have long been associated with CS1 but there doesn't seem to be a consensus as to their format or to the pedagogical role they play. Historically (as the name implies) they were meant to be done in the laboratory, but with the increase in enrolments and the inability of institutions to cope, take-home labs became common. This shift, from an environment where time and collaboration were fully controlled, to one with no controls at all, has naturally led departments to reduce the weight associated with labs from 20-30% of the overall course grade to less than 10%, and this has, in turn, led economically-minded students not to do labs. In fact, this lack of control has led some departments, esp. those offering distance education, to drop labs altogether. It is also not clear whether labs should be viewed as evaluation tools [2] (like assignment projects but controlled) or teaching instruments [5] (like lectures but explorative).

SIGCSE'02, February 27– March 3, 2002, Covington, Kentucky, USA.

It is our belief that, if properly designed, labs constitute an integral component of CS1 and that, in fact, the knowledge they impart cannot be conveyed as effectively in lecture. Many students approach CS1 the same way they approach, say, Math1 or Physics1: "as long as we understand the concepts presented in lecture and read the corresponding chapter in the text, we are doing fine". It takes 4-5 weeks before these students sit at a computer, but by then, it is too late. Labs enable us to create early rapport between student and computer; thereby exploiting the fact that CS is the only science whose ontological reality is so readily accessible. On the conceptual level, labs are unique in that they can uncover misconceptions and cognitive models the student comes to CS1 with. As argued in [5], labs adopt a constructivist approach that enables student to confront any mental models they have, and build new ones. Moreover (and this is based on informal surveys we conducted over two years), it seems that the younger generation finds the Labs' explorative approach more appealing than the analytic one often adopted in lecture ("because it is *fun* to recognize patterns in repeated observations, but it is *boring* to learn and apply an abstract concept").

In this paper, we present a summary of our experience with labs after designing, implementing, and refining them for three years (since our CS1 switched to Java). Section 2 presents a number of design principles that can be viewed as guidelines for designing labs so that they fulfill their pedagogical role and complement lectures. One of the principles calls for a specialized set of library classes, and these are discussed in section 3. Section 4 implements the principles by specifying the structure of each lab, while section 5 provides concrete samples from selected labs.

Our work is restricted to introductory courses adopting the *Objects-First* approach. Using the classification scheme of the latest Computing Curriculum report, CC2001 [3], this would be $CS101_O$ (the first in the new three-course sequence) or to the traditional $CS111_O$ (the first of two introductory courses).

2 Design Principles

- *Labs are not to be thought of as evaluation tools*. Students should see them as educational instruments that complement the coverage in lecture and in the textbook. To that end, students must be allowed to discuss the tasks among each other and/or seek help from the TA, and instructors are encouraged to draw examples from the Labs in their lectures and answer lab-related questions. Nevertheless, some mechanism must be in place to entice students to do the labs within their scheduled weeks (and to check off the names of those who did).

- *The Labs must be portable and self-paced.* Students can do them in a campus lab or on any home PC that can run the Java SDK. This implies that all needed Java packages installed on the Departmental system must be available for download (perhaps as one jar file). A generous time limit must be allotted for the completion of a lab; one that leaves ample time for students to consult the textbook or seek help, and that allows advanced students to ponder about the posed ideas and complete the optional tasks.

- *Labs must be explorative in nature*. Unlike lectures, which typically follow an analytic approach, labs must develop skills and processes for discovering behaviors. While designing labs, it is helpful to think of Computer Science as a natural Science and to think of labs as its phenomenology, or, to let the name "Lab" live on, experiments. Once students achieve a high comfort level with experimenting, they should never have to ask: "Can you assign an `int` to a `float`?" but rather, will find it more natural to write a tiny program to check. They also will fully relate to black-box testing, software specification, and software validation.

- *Labs on object-based programming must be set in an abstraction that is credible and consistent.* Labs in the first part of the course concentrate on class usage (write a main method that uses a given API) while the remaining ones cover writing classes. It is critical that these earlier labs are not viewed as inferior to the later ones, and hence, they must not explain things by "as we shall see later when we look under the hood". Instead, they should be viewed as teaching how to confront complexity by extracting facts that are implementation-independent. It has been widely recognized [1,7] that without a clear separation between user and implementer, the student does not benefit from abstraction (as a tool to learn programming), and does not see abstraction as a tool (for dealing with complexity in general). But for abstraction (the *cover story* as [1] puts it) to work, we must adopt standard models for presenting classes and for visualizing their instances. Implementing and enforcing these models can be achieved at various levels, but no matter how implemented or enforced, some sort of a plausible mental picture must emerge (or else there are too many holes in the story). We achieved this, as shown below, by providing a cleanly designed set of inter-related classes.

- *Only one method (`main`) should appear in the class that the student writes in an object-based lab.* A typical object-based lab involves writing a `main` method that uses existing classes to perform input, validate, compute, and then output. But even in simple programs, a task may need to be repeated (e.g. output two numbers with a thousand separator) and one is tempted to place the task's code in a (`static`) method so that `main` can re-use it. Such an approach, which traces the historical evolution of procedural abstraction, teaches bad habits (state is held only by the caller; methods have no side effects; the returns of a method is determined solely by its parameters) and prevents the picture of an object as an intelligent agent with state, from emerging. We therefore must design these labs so this situation does not arise, or so that it is handled without adding `static` methods to the class.

3 The Supplied Packages

We have found that supplying our own set of classes (bundled in one `jar` file as two packages) allows us to control the degree of complexity students see, and the level of consistency in naming and documenting. For example, by consistently naming accessors and mutators (using `get` and `set`), students can easily discern the object's attributes and determine which, among them, are read-only. And by ensuring that all classes have `toString`, `equals`, and `clone` methods, students feel confident of their early observations. We adopted `javadoc`-style API throughout. The minimum needed features are:

- **Simple I/O:** This can be text (i.e. console) based or GUI. For the former, the class should provide static methods for reading and writing primitive types and strings. For the latter, Swing's modal dialogs can be wrapped in static methods overloaded for every type. See [4] for a survey of I/O packages and for a concrete implementation of a simple GUI I/O.
- **File/URL I/O:** Two classes (one for reading and one for writing) are needed. The constructor takes the file name (or the URL in case of reading) as parameter and there are (non-static) methods for reading / writing.
- **Assertion services:** The objective here is to expose CS1 students to design-by-contract constructs [6], which provide a unique view of programs as a collec-

tion of contracts rather than a continuum of statements. How the assertion is expressed (as a simple boolean or a predicate), how is it enforced (not at all or via an exception), and the types of supported assertions (pre, post, invariants, etc.) can vary significantly (and so will the complexity). We have chosen (what we see as) a middle ground: the method:

```
static void assert(boolean, String)
```

terminates the program (by throwing an exception) if the `boolean` is false, and displays the passed string along with a stack trace. This can be used to express preconditions, checks, and loop invariants; and can be adapted for post-conditions; but admittedly, is limited to boolean conditions and does not get automatically incorporated into the API documentation. A number of more elaborate approaches are available [9,10] (see [10] for a survey of published work) but it remains to be seen whether the degree of formality they require is suitable for CS1.

- A number of cleanly designed classes that contain `static` and not-static fields and methods, overloaded features, composition, and inheritance. A number of deliberate (but clearly documented) S/E violations can be incorporated (a public field that should be private, an accessor returning an object instead of its clone, a mutator setting a field without validation, etc.).

4 Implementation

We assigned 8 labs per term (4 months), with one week to complete each. This is the most we could in a term because at least four additional weeks are needed for supplementary material (such as Unix, tools; and writing assignment reports) and for two assignment projects (these are individual pieces of work that are completely separate from the labs).

Each lab is self-contained, depending only on preceding ones, and introducing any new materials it needs via explorations. This relieved us from having to maintain precise synchronization with lecture, which is difficult in large, multi-section courses. The first 4 labs are object-based; the remaining 4 are object-oriented. Each lab has three sections all focusing on one topic:

- **Explorative tasks:** Each task asks the student to look for some feature in a given API, write a code fragment that uses (or implements) the feature, add debugging I/O, and then predict the output and verify by actually running the fragment. This way, students learn by observing and, in addition, get into the habit of testing incrementally as they program. The tasks alternate between ones that introduce new topics by asking the student to do something and observe, and ones that probe and challenge the student's mental framework by asking for an explanation.
- **Exercises:** These are similar to the explorative tasks but involve less handholding. Rather than pinpointing the needed method, for example, the student has to search for it. Furthermore, exercises in the object-based labs are drawn from a different set of classes in order to ensure that students are able to read and comprehend an API regardless of context. For example, if strings and file I/O were visited in the explorative part, the exercises would look at `String-Tokenizer` and `Random`.
- **Checking:** This task involves solving a specific problem. In the object-based labs, the student is asked to write a main method that accomplishes a stated task and generates output having a given format (a few sample runs of the sought program are given). In the object-oriented labs, the student is asked to implement a class given its API. When done, the student runs the `eCheck` program [8], or brings the program to a TA. Checking examines three aspects:
  - The program's output must meet the specification in terms of format and layout.
  - The program generates the correct output for a number of test cases.
  - The source file conforms to a given style code (mainly naming convention, indentation, and the placement of braces).

  If the program did not pass a test, the student is shown why and is asked to investigate and then re-check. The process can be repeated as many times as needed until the lab is successfully checked.

5 Examples

The explorative section is obviously the heart of the lab and one that must be designed with utmost sensitivity to the (average) student background. In Lab 1, for example, we assume no previous exposure to classes and features, and hence, the pace is slow, terminology is defined whenever used, and the scope is limited to the edit-compile-run cycle, the `main` method, anatomy of an API, static features, primitive types, operators, and expressions. All examples are done "on the fly" so that we can postpone the introduction of local variables to the next lab (see Figure.1). The exercises in Lab 1 examine static features in other classes; e.g. given a string constant like "123", write a program to multiply it by 2. Students are directed to look for the `static parseInt` method in the `Integer` class.

Using the terminology of CC2201, this labs covers topics in the following units: PF1, PL4, PL6, and SE2.

- Look at the API of the supplied packages and get acquainted with its three-frame structure and navigation. Identify the field, constructor, and method groups.
- Observe naming styles; e.g. class names start with a capital, constructors have the same name as the class, finals are capitalized, etc.
- Observe constructor overloading and identify signatures.
- Notice the two qualifiers besides each field and method: `static` or not, and type.
- Invoke `static` methods for I/O and access static fields.
- Use the `static` assert method to validate input (and try various boolean operators).
- Perform "on-the-fly" arithmetic computation using constants and arithmetic operators.
- Access static fields in `Math` and invoke some of its methods. Read the API of `ceil`, `floor`, `rint`, and `round` and come up write a program to expose their similarities and differences.

Figure 1. Explorations in Lab 1

A second example is provided in Figure.2, where the explorations of Lab 2 are shown. Here, the setting is that of a software project (to convert temperatures) and the student is guided through the various phases of the development process. The material covered here includes the software development process; declaration of local variables; assignment and casting; testing; and round-off errors. The units covered are the same as Lab 1 plus SP5.

An example of the checking section is shown in Figure 3 for Lab 2. Recall that at this stage, students know about primitive types, arithmetic and boolean operators, mixed-type expressions and casting. They have also been fully exposed to `static` features, both in the supplied packages and in the standard base libraries; and in particular, the `Math` class. They don't know about selection but can still do input validation by using the `assert` method. Note that we place strong emphasis on formatting even though it may be argued that getting the answer right is good enough at this stage. We do so for two reasons: (1) it is important at this early stage that students relate to the notion of program correctness *relative to specification* rather than some subjective measure, and (2) we facilitate automated lab checking via eCheck [8] by eliminating the so-called "output variability" problem that usually complicates automated checkers.

- Read a temperature in Fahrenheit, validate (using assertion), convert to Celsius, and then output with some formatting.
- Observe that this cannot be done without local variables.
- Declare local variables and assign meaningful names to them.
- Try assigning values of one type to variables of another; casting.
- Test your program using correct and incorrect inputs, boundary cases.
- Think about responsibility (whose fault is it?) when an error occurs.
- Add statements to convert back to Fahrenheit and compare with original input. How is that different from black-box testing? Why does the self-test sometimes fail?

Figure 2. Explorations in Lab 2

Write a program that reads the altitude of a satellite (in km) as an `int` and outputs its orbital period in hour, min, sec. Use assertion to terminate the program (and print an appropriate error message) if the entered altitude is not positive. Otherwise, output the sought period with layout and format precisely as shown in the following two sample runs:

```
Enter the satellite altitude in km
1000
**********************************
Orbital period:
1 hr, 45 min, and 5.7 sec.

Enter the satellite altitude in km
35800
**********************************
Orbital period:
23 hr, 56 min, and 29.0 sec.
```

The following formula computes the period P (in sec) in terms of the altitude A (in km), where K = 0.00995 is the Kepler constant and R = 6378 is the Earth radius in km: $P = K (A + R)^{3/2}$

Figure 3. Checking in Lab 2

6 Conclusions

We have presented a number of guidelines for the design of CS1 labs and the Java packages that must accompany them. We hope these strategies will be helpful to those seeking to incorporate labs as teaching instruments in their first-year courses. Implementing these strategies will probably vary somewhat from one institution to the next based on the program (CS, CE, Information Tech, or mixed), which affects the choice of application areas, and more importantly, on whether the two or the three-semester implementation [3] of CS1/CS2 is in place. Our own experience is based on two courses ($CS111_O$ and $CS112_O$), but if the three-course sequence ($CS101_O$, $CS102_O$, $CS103_O$) is used, which is expected to become standard over the coming years, then we would make all labs in $CS101_O$ object-based. This means students will see inheritance, method overriding, and polymorphism for a full semester without ever writing a class that extends another. (This is to be contrasted with the current $CS111_O/112_O$ in which inheritance usage and implementation are in back-to-back lectures; inhibiting any abstract model from forming.) In that case, all labs in $CS102_O$ would be object-oriented.

References

[1] Bucci, P., Long, T., Weide, B., Do we really teach abstraction. *Proceedings of ACM SIGCSE 2001*.

[2] Chamillard, A., Joiner, J., Using lab practica to evaluate programming ability. *Proceedings of ACM SIGCSE 2001*.

[3] IEEE-CS/ACM Joint Task Force *Computing Curricula 2001, Computer Science, Steelman Draft.* Available WWW: http://www.computer.org/education/cc2001/steelman/cc2001. The final report is expected later this year.

[4] Koffman, E., Wolz, U., A simple Java package for GUI-like interactivity. *Proceedings of ACM SIGCSE 2001*.

[5] Lischner, R., Explorations: structured labs for first-time programmers. *Proceedings of ACM SIGCSE 2001*.

[6] Meyer, B., Object-oriented software construction. second edition, Prentice Hall PTR, Upper Saddle River, NJ (1997).

[7] Long, T., Weide, B., Bucci, P., Cielint view first: An exodus from implementation-biased teaching. *Proceedings of ACM SIGCSE 1999*.

[8] Roumani, H., eCheck, to be published.

[9] Sitaraman, M., Weide, B., eds., Component-based software using RESOLVE. *ACM Software Eng. Notes*, Vol. 19, No. 4, 1994, pp. 21-67

[10] Turner, J., Zachary, J., Javiva: a tool for visualizing…. *Proceedings of ACM SIGCSE 2001*.