1994

Professional literacy: Labs for advanced programming courses

Henry Etlinger

Michael Lutz

Follow this and additional works at: http://scholarworks.rit.edu/other

Recommended Citation


This Conference Proceeding is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Presentations and other scholarship by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
PROFESSIONAL LITERACY:
LABS FOR ADVANCED PROGRAMMING COURSES

Henry A. Etlinger
Rochester Institute of Technology
Rochester NY
has@cs.rit.edu

Michael J. Lutz
Rochester Institute of Technology
Rochester NY
mjl@cs.rit.edu

ABSTRACT
Our contention is that there now exists a considerable body of lab exercises that may be used in conjunction with introductory courses. There are fewer models available for instructors of more advanced programming courses, especially those courses which attempt to introduce students to current practices in software engineering. In this paper, we report on our experiences in building a second-year programming course that includes a significant lab and project component. These labs and projects are the vehicle we use to introduce students to the world of professional practice in software development.

INTRODUCTION
For some time now, computer science educators have recognized the value of integrating labs into the curriculum (see, for example, [5] or [10]). Most of the attention, if judged on the basis of conference reports, journal articles, and lab manuals that accompany textbooks, seems to have focused on CS1- and CS2-type courses, as well as introductory computer science or general computer literacy courses ([1], [2], [3], [7], [9]). Perhaps this is not surprising. In teaching students the details of a programming language or a software tool (such as an editor or a spreadsheet package), numerous exercises suggest themselves. One might argue, in fact, that many labs merely transfer practice from the "paper and pencil" domain to that of the computer. We hasten to point out, however, that there's certainly much to commend this approach - utilizing computers to help students learn about computers makes sense.

Other labs allow students to simulate some application (for example, setting up a computation using a Turing machine) without the need for going through an entire course. Even labs that deal with basic data structures are relatively straightforward. Students often implement operations for an ADT when provided with a given representation for some object. Other labs provide the ADT package and the student must use the given routines to solve some problem (e.g., students might make use of a linked list package to create routines that perform arbitrary precision arithmetic).

Recognizing that there's now a considerable wealth of lab examples available for basic courses, we turn our attention to more advanced programming courses. These courses focus on more advanced data structures, as well as issues that arise in software engineering. Students need to develop fluency in configuration management tools, in conducting inspections, and in working as part of a team. In addition, they must begin to augment their technical skills in programming per se with other technical skills that involve testing programs, reading and analyzing programs, and designing programs and program components. Finally, students must learn how to present their ideas, whether through an oral presentation or a design document or a user's manual, and to do so according to professional standards.

Our the last two and one half years, we have refined a set of lab experiences that address the issues raised above. The remainder of this paper describes the course in which these lab experiences take place and describes the labs themselves and our experiences with them.

PROGRAM OVERVIEW
RIT operates on a quarter system. Under the current computer science curriculum, students will take four courses during the first two years that include programming in Modula-2 (other courses cover computer hardware fundamentals, assembler language, and either systems or scientific programming). The two courses taken in the freshman year focus on language mechanics and basic data structures. In addition, students become familiar with our computing environment (Sun 3's, Unix, and tools such as window managers, compilers, and text editors). In the sophomore year, we continue the study of (advanced) data structures and file structures, as well as introduce students to the rudiments of software engineering. There's a focus on design in many of the assignments during the second year.

All students in our program must complete four quarters of co-operative education after the second year. The initial co-op assignments are often as junior members of a development team, so there is pressure on us to provide more than technical skills — our students must have early exposure to issues of software engineering. Our goals for the course we will describe are to provide the necessary experience, and to establish a professional attitude towards software development on the part of our students.

COURSE OVERVIEW
Most students take Programming III - Design and Implementation in the fall of their sophomore year. Initially the course was an eclectic collection of topics with no overarching theme. A portion of the course was dedicated to data structures, including trees and graphs, that were only cursorily presented in the second course. Sorting and searching were covered here primarily because they didn't fit anywhere else. Mixed in with these "hard" topics was "softer" material.
such as life-cycle issues and structured analysis and design. Finally, students were expected to produce software products exhibiting modular design, robust implementation, and quality documentation. Unfortunately, this rarely occurred in practice.

The introduction of formal labs in Programming III helped us sharpen the focus of the course. Much of our work was spurred by a change of format: The original four 50-minute lectures per week have been replaced by three lectures and a two-hour lab session. In addition, labs and lectures are decoupled, meaning students may well have different instructors for these two components. To facilitate this, lecture instructors must adhere to a strict syllabus that supports lab experiments and programming projects common to all labs. For administrative purposes, lab instructors grade quizzes, homework, and exams. The final grade is determined by lecturers using information provided by those teaching labs.

LABS — GENERAL FORMAT

A lab can last almost two weeks. The several lectures preceding a lab session are used to introduce topics that we hope will reinforce the next lab. Several days before the lab session, we also distribute a lab handout. The lab handout contains the following sections:

- objectives (several brief statements covering the major points of the lab)
- preparation (specific readings in the course textbooks or documents that should be reviewed before coming to lab)
- when work must be submitted (a lab officially begins the day of the lab session; work must be electronically submitted within five days; students are notified within approximately four days after this deadline whether their work was satisfactory; for unsatisfactory work, students will have several more days in which to resubmit solutions to a specific lab experiment - thus, for students who begin preparing prior to a lab session and who resubmit near the final deadline, two weeks have elapsed)
- what and how to submit (the specific submission commands are listed for each experiment; labs typically include from one to three experiments)
- lab procedures (typically, students are directed to copy certain files from a course account (files might contain source code or questions to be answered, for example); additional discussion follows that explains each experiment at a "global" level, perhaps provides supplemental comments for the more interested student, and, occasionally, poses thought questions for students to keep in the back of their mind as they work through the lab details)
- experiment sections (here the specific details for each experiment are given; the purpose of each file that was copied is described; students are reminded to submit their work once they've completed the experiment)

LABS AND PROJECTS - MAIN GOALS

We look to the lab experiments and projects to reinforce the following overall objectives:

- identify issues and concerns facing professional software practitioners
- expose students to software engineering tools and techniques that allow a practitioner to deal with specific issues
- encourage students to learn how to read and analyze code, both in terms of finding out about its performance as well as to gain insight into its construction and function
- give students guidance in working on a software project in a team setting and what that implies about cooperation, communication, and nitty-gritty details such as building, maintaining, and testing both designs and code
- provide students with models for how to specify code function and solution design
- reveal to students how to make use of more advanced features that a language has to offer

LAB #1 - BINARY TREES

In this lab, we present a collection of modules designed to practice recursive descent parsing based on a formal grammar describing legal input. In the first experiment, students are given a definition file that contains the public interface to a binary tree module. Their task is to complete the corresponding implementation module. In order to test their code, we provide additional modules and a main program (most in source code form, but with one module in object code form only). In particular, we provide them with a complete module to handle input operations. They can inspect this module to see how we make the transition from the formal grammar that describes legal input to actually reading character input from a physical file.

In the second experiment, we have students remove the object file we've provided and then build the expected source module. There's only one routine that needs to be written, but it's recursive and it utilizes routines from the binary tree library. While module and procedure specifications are written in an informal manner, this is an early introduction to the task of writing to spec. Also, as we'll see later, there is a direct connection between lab #1 and project #1, so it's quite important to students that they fully understand what happens in lab #1.

LAB #2 - MAKE

Make and the use of Makefiles is a skill we expect our students to master, yet rarely take the time to teach. We have in the past provided students with papers to read about Make, but it's not until they actually use it and use it for different purposes that the richness of the tool becomes clear to them. In this lab, we have students perform two simple experiments, using a nonsense application. In one case, they must explicitly include rules for compilation, while in the second case, they define implicit rules to handle compilation. One of the easy traps that students often fall into is to create a Makefile that "works", but by accident. For example, they only use the Makefile to create an initial version of an executable file, without testing to see if the Makefile "holds up" when changes are made to various dependent modules. Our test data begins to address this issue so that they also learn how to set up correct dependency information.
LAB #3 - GRAPHS

Graph applications can quickly become too large to conveniently fit into a lab. Rather than trying to do it all in this one lab, we’ve opted instead to start by looking at how we might implement graphs. We offer students two experiments, one of which represents a graph using a matrix, and the second of which creates a dynamic representation. In each case, the interface to a graph package is given, as well as a partial implementation. Students must complete the implementations. We also provide a simple test program that can be used to validate their implementations. Graphs may be a part of a subsequent project, so students will likely see graphs again. In addition, we expect students to create their own Makefiles and submit them in this lab. This reinforces concepts presented earlier to see that students really understood the main points. (As we’ve pointed out previously, students electronically submit their work. With our submission system, we use the student’s submitted Makefile to attempt to create a workable executable for testing. If the Makefile fails, the student knew about it right away and finds out how to repair the mistake(s). Once again, this mechanism helps reveal some of the more subtle kinds of errors that can occur when attempting to create a modular program, comprised of pieces, that can exist in a variety of forms and in a variety of locations.)

LAB #4 - SORTING

This lab turns out, unfortunately, to be one of our more demanding labs. There are three separate experiments. In the first experiment, we provide four different sorting routines (as well as a test harness). Two of the routines contain errors and students must locate and correct these errors (this represents some early, informal, testing and debugging practice). In the second experiment, students run each of the four sorting routines on data files that differ in the amount of data they contain as well as the type (sorted, random, etc.). Students must answer questions about performance (we suggest that they graph the results to help them see key relationships). Finally, in experiment 3, they must change one of the sorting routines to perform a different sort, and then again, run performance measures.

This lab allows us to cover a wide array of interesting topics. For example, we introduce procedure types (in which we pass the name of a procedure as a parameter to another procedure) since the experiment calls for running four different sorting routines through the same testing sequence. It’s easy to show students how to construct some simple Unix shell scripts (something that hasn’t been covered in previous classes) to set up loops to exercise the sorting routines against all the test data sets. We can also show students how to use a plotting package (such as Gnuplot) to produce the needed graphs. We also talk about plug-compatible modules. In lab #3, we used the same interface with two different implementations. Since we didn’t want to complicate the lab further, we told students to essentially set up two separate directories, one for each experiment. In this lab, we show students how to override the common convention that the implementation and definition modules must have the same name.

LAB #5 - SEARCHING

This lab closely follows the pattern set in the previous lab. Students manipulate both linear and binary search algorithms and also develop performance figures for the two algorithms. The techniques and tools we’ve introduced in the previous lab are now something students can use with ease.

LAB #6 - TESTING

This is the last of the individual labs and is purposely kept short. Students are given a complete program that includes a well-known algorithm (e.g., binary search). They are told that the program contains errors and told to develop test cases that reveal the errors. In lecture, we’ve now discussed approaches to testing, such as black box and white box. An added feature is that the code is specified much more formally, using pre- and post-conditions for the binary search procedure. They can study these specifications carefully as a precursor to developing test cases. Students must submit test cases as well as discuss the process they used to discover these test cases.

PROJECTS

There are three projects given in the course that complement the lab experiments. These projects are meant to be done independently and are more global in scope. The first two projects are done by each student, separately, while the third project is done as a team.

PROJECT #1

Project #1 has traditionally built upon lab #1. We ask students to build a theorem prover for a simple language. We provide the design for a solution and most of the needed modules. Students must provide the binary tree implementation (which they can take directly from lab #1), the module responsible for parsing input lines and building up trees representing two expressions (this module strongly resembles the input parser we’ve given them in lab #1), and a module that actually proves or disproves the equality of two expressions (the heart of this module is a recursive routine).

The project is relatively short, but allows students to get a clear sense of what we expect in adherence to programming standards. The design of the project can be viewed as a case study and reviewed later in the course when we discuss design issues. We do not require Makefiles for this project since students are still learning about them via labs #2 and #3.

PROJECT #2

Project #2 is a major, individual project. In past versions of the course, very few students were able to complete the project, nor did they seem to learn much from it. It is still difficult for many students to complete the project, but there is a higher probability that they will "learn" more, because we use the project as a vehicle for introducing a number of practices that relate to software development.

Students receive a project specification, a problem specification, and a high-level (or architectural) design document. The project specification sets down the details and constraints of the project. It points out, for example, that students have several options available when designing a solution (e.g., they can use part or all of a design that we provide or they can design a solution themselves). Students will be primarily responsible for designing test data for the project, although we will provide some baseline cases (the philosophy in earlier courses and in the beginning of this course is that we provide most of the test data for labs and projects; obviously, we need to move away from that approach and this is where we begin to do that). We will in fact cover both design and testing in lecture before this project is due. In
grading submissions to this project, we inspect functionality,
adherence to standards, and design.

We give students a problem specification. Some projects
have involved writing commands similar to those found in
Unix. For example, one project calls for development of a
make-like tool (this project is also attractive because the un-
derlying data structure needed by make is a graph). In these
cases, we provide Unix-like manual pages. Whatever form
is used, students clearly must spend some time reading and
understanding the specification (and, this activity dovetails
with the introduction of software process in the lectures).

The final component we give to students is what we call a
design package for a possible solution to the problem. This
package defines the modules we would have in our solution.
For each module, we convey the design "secret" contained in
the module as well as the responsibilities (or services) pro-
vided by the module. We also provide a "uses relationship"
which is a graphical/textual portrayal of the interconnec-
tions between modules. Our thinking about how to express
design has been heavily influenced by the work of Parnas
and Lamb (see, for example, ([6], [8]).

Students must clearly integrate a number of important ac-
tivities in this project. They have to read and understand
specifications, make a "real" effort at designing a solution,
build and maintain a sizable solution (including a required
Makefile), and test their own programs.

PROJECT #3

Project #3 is a culminating activity that we hope is inter-
esting, educational, and fun for students to do. Students
are assigned to teams of about four students and typically
have to write a program that plays a game (e.g., checkers,
backgammon, reversi). While programs must support the
rules of the game, teams have freedom to design the inter-
face and to add additional features that may make the game
more interesting or useful. We use the remaining four lab
sessions in the course for the following activities related to
the project:

- a team lab that introduces students to RCS, a standard
  configuration management package; we also encourage
  students to begin formulating a team organization and
  plan of work
- a session devoted to design reviews; each team is re-
  quired to present the complete game specification they
  intend to work on and to discuss their tentative de-
  sign solution; in addition, they turn in a high-level de-
  sign document patterned after the one we've distributed
  with project #2
- a session devoted to inspections; each team must in-
  spect two or three items (in lecture, we've already cov-
  ered inspections and shown the SEI videotape on soft-
  ware inspections ([4]); teams submit inspection reports
  for each item inspected
- in the last lab, teams present and demonstrate their
  final product; teams also submit a user's guide and a
  (revised) high-level design document

We're looking for students to utilize much of what we've cov-
ered throughout the term in this final project. We expect a
Makefile and hope they make use of RCS. We expect doc-
umentation to meet professional standards and we provide
feedback to encourage students to improve their presenta-
tions.

CONCLUSION

The changes to Programming III did not happen all at once.
The first time we used labs, we had only three or four. Later,
we added labs on make, graphs, and testing. Changes to
projects occurred after that. At that point, we knew we
had enough material to cover technical points, and so then
we aimed at making the process and the goals for the pro-
cess more visible. Instilling a professional attitude in our
students has been challenging, but an effort well worth our
time. Our students are now better equipped to meet the
demands of the work environment they will shortly enter.

Now that we have most material in place, we're ready to
throw it out! We expect to modify our curriculum in the
next several years to embrace an object orientation. In addi-
tion, it's highly likely that we'll replace our aging equipment
and we hope that our new environment offers us even more
exciting experiences to reveal to students. Will we really
lose everything we've done with respect to this course? No.
We have demonstrated convincingly that we can design suit-
able lab experiences to support more advanced programming
courses. We can teach and illustrate those critical founda-
tion skills that all practicing software developers must know.
Whatever approach we adopt next, we will certainly be able
to reuse experience we've gained in designing the current
course.

ACKNOWLEDGEMENTS

We appreciate the support our departmental colleagues have
shown us over the last several years. We gratefully acknowl-
edge the efforts of those in the software engineering edu-
cation community. Finally, we express our thanks to the
students who have seen our course grow from a random col-
lection of topics to a disciplined, well-organized, informative
introduction to modern software engineering practice.

REFERENCES

Laboratory Exercise for CS2." SIGCSE Bulletin 23, 1
[2] Decker, Rick; Hirschfeld, Stuart. Lab Manual to Accom-
pany Pascal's Triangle. Belmont, CA: Wadsworth Pub-
[3] Deimel, Jr., Lionel E. "The Uses of Program Reading,
[4] Deimel, Jr., Lionel E. Scenes of Software Inspections:
Video Dramatizations for the Classroom. Carnegie Mel-
[7] Parker, Jeff, et al. "Laboratories in the Computer Sci-
cence Curriculum" Computer Science Education 1, 3
(1990): 205-221.
[8] Parnas, David L. "On the Criteria to be Used in De-
composing Systems into Modules." Communications of
the ACM 15, 2 (December 1972): 1053-1058.
Programming Projects" SIGCSE Bulletin 22, 1 (Febru-
the ACM/IEEE-CS Joint Curriculum Task Force. NY,