How to Present a Programming Project

Leon Tabak

Associate Professor of Computer Science

Cornell College

600 First Street West

Mount Vernon, Iowa 52314–1098

Voice: (319) 895-4294

Fax: (319) 895-4478

E-mail: l.tabak@ieee.org

URL: http://people.cornellcollege.edu/ltabak

Abstract

Extended research and independent study in computer science challenge students in special ways when it comes time to present the finished work. In particular, models for the presentation of research that work well for students of the natural sciences do not always have obvious application for students of computer science. This paper identifies some of the special challenges and shares some methods by which students can meet those challenges.

Special Challenges

> "Nevertheless, the scientist builds in order to study; the engineer studies in order to build. A little reflection shows that the power to make things, in imitation of our Maker, is a gift for our sake, not his. As he scornfully reminded the people of Israel, he doesn't need our creative powers: 'The cattle on a thousand hills are mine; if I were hungry, would I ask you?' (Psalms 50:12) So we must conclude that the ability and the call to create are given to us to enrich our lives and to enable us to enrich each other."
>
> —Frederick P. Brooks[7, pp. 62–63]

Because they recognize the benefits of extended research and independent study, teachers of computer science actively encourage students to pursue projects, assist students in the identification of topics for exploration, and guide students over technical hurdles they encounter in the work. However, many students need more help in the final phase of their projects.

Students in all disciplines gain confidence by explaining to other students, teachers, and professional practitioners what they have accomplished. At symposia, conferences, and professional meetings they meet peers whose expertise and special interests differ from their own. The practice of making their own work interesting and comprehensible to diverse audiences helps students develop important skills from which success in professional practice will follow.

However, students of the natural sciences enjoy some advantages over students of computer science. For example,…

- At a symposium where students share the results of their research, physicists will explain methods they used to test a hypothesis. A computer scientist may not have a hypothesis.
- Chemists will show diagrams of apparatus they used in the laboratory. Research in computer science might require only the kind of computer that is already familiar to all.
- Biologists and geologists will show photographs of the sites where they collected their specimens. Projects in computer science rarely take students to volcanoes, rain forests, or coral reefs.

Students of the natural sciences know that most members of their audience have studied at least a little biology, chemistry, geology, and physics, if only in high school. They can count on the audience's familiarity with the kinds of questions that engage scientists in those fields. Students of the natural scientists rely upon a familiar formula when they compose their presentations. Those who come to listen to a talk about research in one of the natural sciences expect to hear a clear statement of a hypothesis, followed by a sketch of methods, a summary of data, and conclusions.

By contrast, a student of computer science might begin a project with a list of design goals rather than a hypothesis and end with a better understanding of how a system

works, but no data to which a curve might be fitted. Although members of the audience will have experience using computers, many will have made no study of computer science. Presenters cannot assume that members of their audience know what computer science is about.

For an undergraduate student of the natural sciences, research usually means measuring something. Research for an undergraduate student of computer science usually means building something. In the case of biologist's or geologist's talk, listeners might accept the volume of data collected as evidence of the time invested in an investigation. On the other hand, people who are used to using very sophisticated commercial software but who have never studied computer science and never tried creating software themselves might not recognize the effort invested in a student's creative design of a program that in the end executes only a single function.

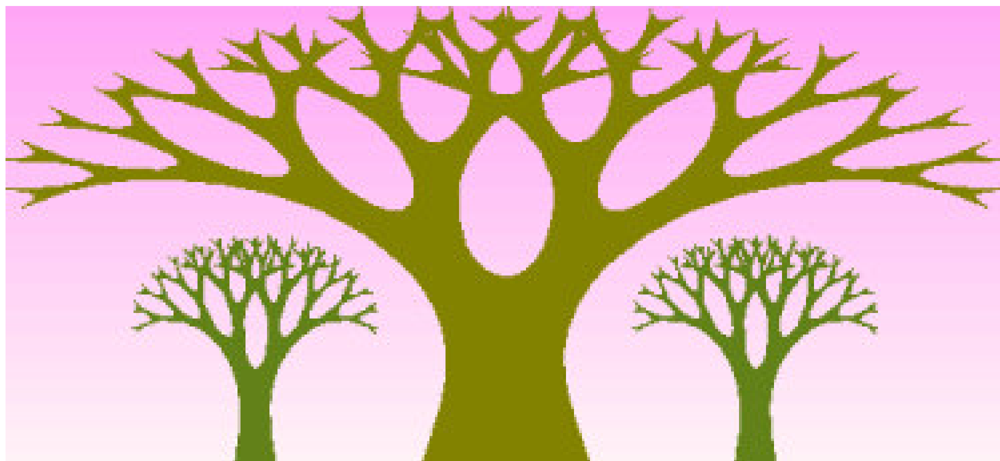

Figure 1: An experiment with Bezier curves, captured with the GIMP program.

Variety of Projects

> "It's the way I study—to understand something by trying to work it out or, in other words, to understand something by creating it. Not creating it one hundred percent, of course; but taking a hint as to which direction to go but not remembering the details. These you work out for yourself."
>
> —Richard P. Feynman[11, page 15]

Computer science lies at the intersection of mathematics and engineering. Computer science is an applied and interdisciplinary field. Although students can try to extend theory or measure the performance of hardware and software, most students prefer exercises in design. For example,…

- A student might write a program as a way of gaining a deeper understanding of an algorithm. For example, a student might express a mathematical description of public key encryption in the Java programming language.

- A student might create software to use in teaching others how an algorithm works. For example, a student might write a program that illustrates each step in the execution of a sorting algorithm.
- A project can deepen a student’s understanding of a particular discipline, style, or method of programming. For example, a student might explore client-server, event driven, functional, or extreme programming.
- A student might use the computer to produce from numerical data an image that makes plain patterns that are invisible in tabulations of numbers. For example, students might use the computer to draw images of fractal sets.
- A student might simulate a natural phenomenon in software. For example, a student might model changes in the population of predators and prey animals, or flocking and schooling behaviors.
- A student might create a tool for a client’s use. For example, a student might write a program that displays images on the screen in rapid succession for a professor of psychology who wants to measure human visual perception.

Projects, more than exercises that students complete overnight or during a single week, offer students the opportunity to appreciate the each part of a software product’s life cycle and the variety of skills needed to create excellent software. Some kinds of projects give students practice determining a client’s requirements. Other kinds give students more practice responding to a client’s requests for enhancements or adaptations of initial versions of software. All kinds of projects give students practice writing specifications and testing software.

Content and Format

> “I did not distinguish between ‘art’ and ‘science’ and still don't.”
>
> —Alan Kay[18, page 39]

> “The experience of seeing the unseen has provided me with insights and questions my entire life. It seems to have struck a responsive chord in many others as well.”
>
> —Harold E. Edgerton[10]

Students often begin their projects with immediate and practical goals. Students want to makes themselves experts with the languages and tools that are most popular with employers. Although a research project can advance such goals, mastery of a language's syntax or a software product's menus cannot be the principal goal of a project that a student presents at a conference.

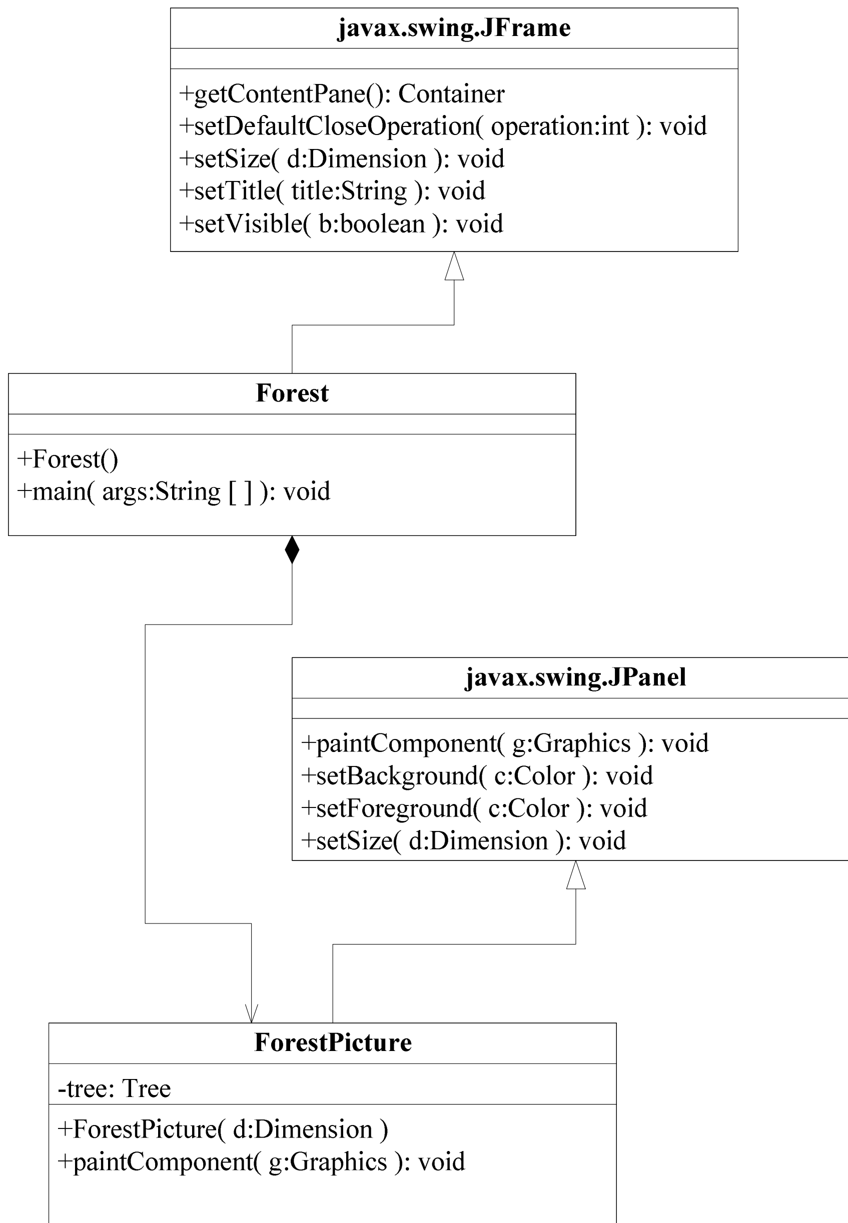

Figure 2. A UML class diagram, created with the *Dia* program.

In the first part of a presentation, students should describe the deeper goals of their work. Rather than speak of a desire to learn PHP and MySQL, a student might define a project as an experiment with multi-tiered architectures. Rather than say the goal was to learn how to use the Java3D package, a student could talk about a study of affine spaces or scene graphs.

Wherever possible, students should list objective and quantitative goals. If the student began the project with the intention of producing a software product, the student should list fully and honestly the schedule and functionality to which the student committed at the outset of the project. Success does not require that students meet all schedules with software that executes all planned functions flawlessly. Success does require a faithful and reflective analysis of what worked and what proved to be more difficult than expected.

In core courses, teachers introduce students to a programming language, to algorithmic thinking, and to fundamental data structures. In projects, students have the opportunity to apply the skills and knowledge their gained through coursework in larger contexts. Students can therefore identify the principal phases of their work and the principal components of their products. Students should emphasize the relationships among the parts. Planning for testing takes place during design, a hierarchy organizes classes that inherit one from another, client and server processes exchange messages, and so on.

In many sciences, students must serve long apprenticeships before gaining access to the sophisticated and expensive instruments needed for research at the frontiers of the field. Even then, such students often find themselves working on large teams. They see their own experiments as links in long chains of related experiments, from which understanding accumulates.

By contrast, computer science attracts students by giving them at the outset of their studies tools as powerful as those possessed by senior researchers. The appeal of independent work can tempt some students to overlook work already done by others and to present their own creative efforts as though they were *ex nihilo*.

In comparison to those who direct research in natural science, teachers who direct students' research in computer science may have to make a stronger case for the value of beginning research with a survey of the literature. They may have to give more explicit directions on how to use the literature.

All beginning students struggle to understand how original their research must be. In some disciplines, there is always something new to catalog—a census of song birds in yet another meadow, fossils in a quarry, an assay of drinking water. In computer science, every new implementation of an idea yields new insights. A student can improve upon prior work by providing a clearer expression of its (old) ideas. Style matters as much as performance. Subjective criteria have a role in the assessment of computer-human interaction. Aesthetic criteria can distinguish one program over another that offers equal efficiency.

Available Tools

> "Thus computing is, or at least should be, intimately bound up with both the source of the problem and the use that is going to be made of the answers—it is not a step to be taken in isolation from reality."
>
> —Richard W. Hamming[14, page 3]

Students need familiarity not only with software that will help them create programs but also with software that will help them present their programming projects. Students can find the software listed here on the Web (or packaged in distributions of Linux), licensed for download and use at no cost.

LaTeX

LaTeX[16] is a language for creating typeset documents. It is especially well-suited for scientific and technical exposition, not only because it includes commands for creating the various elements that an author might wish to place into a scientific report, but also because it assigns the elements beautifully proportionate sizes and spacing.

The language includes commands for the creation of all kinds of mathematical expressions, lists, tables, figures, a title page, an abstract, footnotes, and so on. The LaTeX compiler automatically numbers sections and subsections, equations, figures, and references. It can generate a table of contents.

Authors can also use a variety of available packages with LaTeX. For example, the *graphics* packages enables authors to include Encapsulated PostScript images in their documents and the *listings* package enables authors to include source code (with keywords of the specified programming language automatically highlighted in boldface font).

The related BibTeX program orders bibliographic items and the information within those items. The related SliTeX program (like Microsoft's PowerPoint) produces slides that authors can share via a video projector or an overhead project, The *dvipdf* program translates the output of the LaTeX compiler into Adobe's Portable Document Format (PDF).

The GIMP

Authors can use the GNU Image Manipulation Program[4] to edit scanned photographs, draw pictures, and capture images from the computer's screen. Authors can use the GIMP (like Adobe's Photoshop) to adjust brightness, colors, and contrast. They can crop, rotate, and scale images. The GIMP can encode images in the PostScript, JPEG, PNG, and other formats.

gnuplot

With *gnuplot*[5], authors can produce graphs by directing the program to read data from a file or by describing a curve with expressions and calls to functions in the C

programming language. The program can generate Encapsulated Postscript files that describe the graphs, so that authors can include the graphs in LaTeX documents.

Dia

With *Dia*[2], authors can draw class diagrams using the Unified Modeling Language. Class diagrams illustrate relationships among classes and the attributes and methods of each class (see figure 2).

The program's menu also includes symbols for entity relationship and other kinds of diagrams. Authors can save diagrams in the the Encapsulated PostScript, Portable Network Graphics, or Scalable Vector Graphics formats Authors can include diagrams saved in those formats in Web pages and printed reports.

ghostview

Encapsulated PostScript is a convenient format for images that will be included in LaTeX documents. *Ghostview*[3] is a tool for viewing PostScript files.

noweb

Noweb[17] is a system to support "literate programming." Donald Knuth, author of TeX, the program from which Leslie Lamport derived LaTeX, coined the phrase "literate programming" to describe an approach that mixes statements of a programming language with text that describes the code in a single file. That text can include LaTeX commands (or HTML tags).

Within the single file, an author can describe a program in an outline that provides successively greater detail at deeper levels. The use of a markup language like LaTeX enables an author to fully describe the goals of a programming project and the means its of implementation with equations, graphs, citations, and so on.

Noweb provides one program that extracts and orders the elements of the outline. It associates code with the corresponding commentary. An author can then pass the output of that program to the LaTeX compiler to produce a typeset document for human readers..

Noweb also provides a program that ignores the comments and extracts only the code. It produces input for the Java (or other programming language) compiler.

doxygen

Doxygen[19] extracts comments from source code. Authors can include within comments HTML tags or LaTeX commands, then direct *doxygen* to order the extracted comments in hyper-linked Web pages or in a LaTeX document.

ant

Ant[1] is tool for automating the compilation of programs and documents. An author lists the files to be compiled, the dependencies of files one upon another, and the steps for compilation in an XML (eXtensible Markup Language) document. Each step invokes another tool: e.g., the *noweb*, LaTeX, BibTeX, *doxygen*, Java compiler, or Jar (archiving) programs. An author can describe several different tasks in a single XML file, then direct *ant* to execute one, several, or all of the tasks. With *ant*, an author can create an executable program, create a typeset document, create pages for display on the Web, create an archive, or generate all of the products related to the project at once.

```
<project
  name="forest"
  default="forest-noweb"
  basedir="~/formulae">

  <description>
    Targets: forest-noweb, forest-latex,
             forest-javac, forest-total
    </description>

  <target name="forest-noweb">
    <description>
      Input:  forest.nw
      Output: forest.tex,
              Forest.java, ForestPicture.java
              Tree.java, TreeParameters.java,
              DirectedEdge.java
      </description>

    <java
      fork="true"
      dir="${basedir}/forest/formulae"
      jar="${basedir}/literate.jar">

      <arg value="${basedir}/forest/formulae/forest"/>
      </java>
    </target>
  </project>
```

Figure 3. XML-encoded description of how to build a project, for the *ant* program.

An Abbreviated Example

> “Advanced students ought to present their software projects as architects-in-training do, to juries capable of aesthetic judgments.”
>
> —David Gelernter[12, page 130]

> "I still think that most people don't really know what engineers do. And I think that many engineers who try to explain get incredibly mired down in the nitty-gritty… I think the message is an easier one than that. It's this business of liking to solve problems and wanting to serve society… Technology keeps changing—exciting things keep happening in technology. One of the wonderful things about engineering is that it's a lifetime of learning. That's really fun. And we don't give that message."
>
> —Eleanor Baum[20, page 44]

The project illustrated in this article's figures began with a study of binary trees and an exercise that asked for a program to draw a visual representation of the data structure. The first version of the program gave the tree's trunk and branches no thickness. A subsequent version modeled branches with rectangles. The substitution of trapezoids for rectangles produced tapered branches. A further evolution of the program added fillets. These curves replaced the angles at points where branches meet with smooth bends. The result is an image like that shown in figure 1.

A student presenting this project could show the audience how the program's several classes are related through inheritance and aggregation (see figure 2). With a diagram, the student could outline the vector arithmetic that produces the vertices of the polygon that represents a branch with thickness. With cautious respect for the potential of mathematical exposition to overwhelm general audiences, the student could explain how the control points of a fillet are weighted averages of the connected branch's endpoints.

Alternatively, a student who presents this project could discuss problems encountered and lessons learned in the course of mastering the mathematics and writing the code. In this case, an incorrect sign blocked progress for a day. At a later stage, failure to include a vector in a sum gave birth to another elusive bug.

This project demonstrates a technique that has practical application in computer-aided design and manufacture. A desire to draw a pretty picture led the experimenter to learn how to use new tools and to develop an improved discipline for software engineering. A report should mention the discovery of links to other fields and the relevance of experience gained to future work.

Similar Challenges

> "It is not a question of creativity versus discipline; creative work is simply not possible without discipline."
>
> —Watts S. Humphrey[15, page 26]

Many guides for composing effective presentations contain advice relevant to presentations in all scientific fields. Articles and books[8,9,13] suggest rules of thumb for deciding how many slides to show during a twenty minute talk, how many items to include on a slide, how large the typeface should be on a poster, and so on. Of course, the most important common advice recommends an early start and numerous revisions.

The effort and time required to create a bug free program can consume most of an unwary student's budget and leave too little for the equally difficult task of creating a successful presentation. At the same time, code can be voluminous, even in programs that execute only a few functions. Students, who naturally want others to know how long and hard they worked, will strain against a temptation to show gory details of the source code.

Students benefit from the experience of working on projects over the course of a term, a summer, or a year. Projects give students opportunities to exercise their initiative and creativity. They allow students to discover their own individual interests within the field of computer science. In work on projects, students often see clearly for the first time connections between academic study and professional practice. Self-charted explorations help students select post-graduate plans. Projects provide a context for teamwork, leadership, and collaboration. Students develop interpersonal skills that complement their analytical skills. However, students can fully realize these many benefits only when they present their work well.

References

> "A good attitude to take, from the first day of any programming project, is that the system being built is fundamentally flawed and doomed. The goal of such a project, then, is simply to build a system that will last long enough for a better one to come along, and perhaps also be, for a brief moment suspended between eternities, the best program of its kind yet build."
>
> —Nathaniel S. Borenstein[6, page 102]

1. *The Apache Ant Project.* http://ant.apache.org
2. *Dia.* http://www.lysator.liu.se/~alla/dia
3. *Ghostview.* http://www.gnu.org/software/ghostview/ghostview.html
4. *The Gimp.* http://www.gimp.org
5. *Gnuplot Central.* http://www.gnuplot.info
6. Borenstein, N. S. (1991). *Programming as if People Mattered: Friendly Programs, Software Engineering, and other Noble Delusions* Princeton University Press. (page 102)
7. Brooks, Frederick P., Jr. (1996). The computer scientist as toolsmith II. *Communications of the ACM, 39*(3 (March)), 61-68.
8. Connor, C. W. *The Poster Session: A Guide for Preparation.* http://www.aapg.org/meetings/instructions/prepguide.pdf
9. Cranor, L. F. Research Posters 101. *ACM Crossroads,* (Winter 1996 3.2), 10 March 1996.
10. Edgerton, Harold E., *See the Unseen.* http://www.eastman.org/15_travex/edgerton.html

11. Feynman, R. P. (1996). Introduction to Computers. In In A. J. G. Hey, & R. W. Allen (Eds.), *Feynman Lectures on Computation*. Reading, MA: Perseus Books

12. Gelernter, D. (1998). *Machine Beauty: Elegance and the Heart of Technology*. New York: Basic Books / Perseus Books Group.

13. Hailman, J. P., & Strier, K. B. (1997). *Planning, proposing, and presenting science effectively: a guide for graduate students in the behavioral sciences and biology*. New York: Cambridge University Press.

14. Hamming, R. W. (1973). *Numerical Methods for Scientists and Engineers* (second ed.). New York: McGraw-Hill.

15. Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley.

16. Lamport, L. (1994). *LaTeX, A Document Preparation System: User's Guide and Reference Manual*. Reading, MA: Addison-Wesley.

17. Ramsey, N. *Noweb—A Simple, Extensible Tool for Literate Programming.* http://www.eecs.harvard.edu/~nr/noweb

18. Shasha, D., & Lazere, C. (1995). A Clear Romantic Vision. In *Out of Their Minds: The Lives and Discoveries of 15 Great Computer Scientists* (pp. 38-50). New York: Copernicus / Springer-Verlag

19. van Heesch, D. *Doxygen.*http://www.doxygen.org

20. Zorpette, G. (1993). Eleanor Baum: Careers/Profile. *IEEE Spectrum,* , 42-44.