ECE155: Engineering Design with Embedded Systems Winter 2013

Lecture 12 — January 29, 2013

Patrick Lam *version 1*

Software Bricolage

I'd like to make things a bit more real in this course. Today's topic is going to be about how developers actually put software systems together. This will be relevant to you on your co-op terms as well as for any software you may write for your fourth-year design project.

`http://cacm.acm.org/blogs/blog-cacm/159263-teaching-real-world-programming/fulltext`

Programming for School. Our goal is to provide well-specified assignments where we clearly describe the requirements and provide the tools and libraries that you need to use. For instance, we provided you with the `LineGraphView` for Lab 1. In the assignments, you've been using the Android libraries.

In terms of writing code, you only have to write under 50 lines of code for Assignment 4, and similar amounts for the labs. Android imposes most of the structure for your code. In other courses, you get template code and only need to fill in the blanks. Real-world software is sort of like that, but you have to find the template code yourself.

Programming for the Real World. As I've said, often you will check out a large codebase and fix something in that codebase.

Sometimes, however, you need to start a project from scratch. However, these days, you never actually start from scratch. What do you really do? Let's assume that you have a goal that you'd like to accomplish.

First, you have to formalize the goal. We're only going to briefly discuss requirements in this class, even though they're important: high-quality software isn't useful unless it meets the requirements. ECE451 is all about requirements.

1. **Forage**. Look for software that does something like what you want. Maybe you wrote it yourself, or maybe someone else did. Knowing about what software is out there is critical. From the entry: "If I'm really lucky, then it will come with helpful documentation." Note that the software may well be in a number of different languages.

2. **Tinker**. Figure out what the code you have can actually do. This is a hands-on activity. Experiment with the software. Give it inputs and see how it behaves. Instrument the code. Modify it. This is very much like debugging, which we saw last week.

   If you find that what you have isn't what you need, forage more—loop between steps 1 and 2 until it looks like you have what you need.

3. **Weld**. Combine the pieces that you've foraged. Here are some potential real-world problems:
    - dependencies: Other people have also bricolaged their software. Unfortunately, sometimes there are missing and conflicting dependencies.
    - impedence mismatches: Part A produces XML output, while Part B requires CSV input. You have to bridge them.
4. **Grow.** Here's where you start actually building the code that you need to. Start with something very concrete. It just has to work on simple examples. Since you have these bad welds, you'll have to fix bad interfaces between subsystems.
5. **Doubt.** Avoid re-inventing the wheel. Be familiar with what's in the libraries. Ask the authors (but expect the worst). Sometimes you can contribute to the libraries.
6. **Refactor.** Clean up your code and make it more general (as needed). We'll talk more about refactoring later. Improve the interactions between your code and the outside code.

Repeat steps 4 to 6 as needed.

Effectively Using the Web for Programming

Next, I'm going to provide some tips on using the Web for programming. Beware: there may be policy 71 implications as well as copyright implications to using the Web inappropriately. However, it can be a valuable tool when used properly.

The reference for this part of the lecture is a paper by Brandt et al [BGL+09], entitled "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code." I'll provide the highlights in an easy-to-use format. So, here's how to use the web:

- Learn concepts—by reading tutorials. This is relatively slow, and it can be hard to find good tutorials. But it can give an understanding of how things work.

  Experiment with the samples. Try to see if they work for you. Skim the tutorials and extract information you need.

  Tip: You'll do better if you understand how something works. That allows you to generalize your knowledge. It's what we're attempting to impart at a university.

- Clarify existing knowledge—look at things you vaguely know, but aren't quite sure about. Leverage your existing knowledge.

  When programmers import code from the web, they tend to get stuck on code that they didn't quite adapt properly (didn't change all the variable names). Test imported code too!

  You can also look up error messages on the Internet. stackoverflow is a particularly good resource for this.

- Remind yourself of details—like looking up information in a textbook, but faster. Syntax isn't that important to remember by heart.

Refine your queries to find better results. If you realize you need some specific type of tutorial, put it in the query.

References

[BGL+09] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pages 1589–1598, New York, NY, USA, 2009. ACM.