ECE155: Engineering Design with Embedded Systems Winter 2016

Lecture 30 — Software Bricolage; Licenses

Patrick Lam & Jeff Zarnett

Software Bricolage

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 a small number of lines in your lab solutions. 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.

Remember from earlier that, you have to formalize the goal. We've already discussed software requirements, and in the future, the class 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. It's possible that you'll be finished after this step – if you found something that adequately meets your needs, you're done. But the chances of that are pretty small.
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 is coming up soon.

   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.
   - impedance 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.

Software Licenses

A software license is a legal instrument that tells us how a piece of software may be used or distributed. It grants you the rights to do things that would otherwise be an infringement (violation) of copyright law [GK12]. It answers questions like:

- Where and how and how often can you install the program?
- Can you copy, modify, or redistribute it?

- Can you look at the underlying source code?

Software licenses are not something we can ignore. If you do not explicitly declare a license, you effectively get one anyway. Code, like other works, is automatically copyrighted by default. People can read the code, but they have no legal right to use it; to use code in this situation it is necessary to contact the author directly and ask permission [Atw07].

Proprietary License. It's impossible to generalize about proprietary licenses because they are all different. The rights you have are specifically defined and that's all you get.

GPL. The GNU General Public License (GPL) is the most common example of an open source software license. It is a very clever use of copyright law (called "copyleft") intended to ensure that code can be used, copied, modified, and redistributed freely. Code under this license can't be used in proprietary programs. Any changes you make must be licensed under GPL as you got it, so everyone gets the same rights you had ("fair is fair") [Upf07].

More recently, version 3 of the GPL was released. This was intended to address a shortcoming in the GPL version 2 called "Tivoization", named after the Tivo (digital video recorder). Tivo was based on Linux, open source software, but Tivo added hardware protection to prevent users from modifying the software. This violated the principle of being able to modify and run your software... you could modify it but not run it. [Atw08].

LGPL. The Lesser GPL is intended for software libraries; the code itself, once compiled, can be linked to a proprietary program, without having to license the proprietary program under the same license. The library code itself, however, must be follow restrictions like the GPL [Atw07].

BSD. The BSD license is an open source license and it is quite permissive. It effectively allows any use of the code, even turning it into a proprietary product, but disclaims all liability. Unlike the GPL, there are no requirements to share your changes with others or license additional code under the same license.

Mozilla License. The Mozilla Public license is an open source software license from the Mozilla Foundation, the group responsible for the Firefox web browser. It allows source code to be "mixed" – some open source and some proprietary code. Any file licensed directly under the Mozilla license must be available in source form, but unlike the GPL it does not require that all files of the program be GPL licensed. This license can be described as a hybrid between the BSD and GPL licenses, balancing the interests of open and closed source development [Lau04].

Public Domain A public domain work has no copyright owner and may be used by anyone for any purposes. All copyrighted works will eventually enter the public domain, but it takes a very long time (something like life of the author plus 70 years). Things like Shakespeare's plays are in the public domain. The only other way is if the author explicitly puts the code in the public domain.

Digression on public domain: in the USA, the amount of time it takes for a work to enter into the public domain has been extended many times. Disney has been a big advocate of this to prevent Mickey Mouse from entering the public domain. In the year 1998, the first Mickey Mouse animation, Steamboat Willie, was on the verge of entering the public domain. Governments then passed laws to extend the term of copyrights to keep this from happening. You may therefore safely assume that a work is effectively never going to enter the public domain unless the author explicitly puts it there.

License Violations

License violations are taken seriously and lead to legal action. Importing open source code into your software can get you into a lot of trouble. Be careful in your co-op terms that you avoid this difficulty.

Recent Example The Regional Court of Hamburg (Landgericht Hamburg) found a company, FANTEC, guilty of violating the GPL in their media player, the FANTEC 3DFHDL. They distributed their firmware, containing some software licensed under the GPL (`iptables`), and failed to live up to the license terms (they did not distribute the source code as the GPL requires). The judgement required required FANTEC to pay a penalty fee plus legal costs [GV13].

References

[Atw07] Jeff Atwood. Pick a License, Any License, 2007. Online; accessed 7-July-2013.

[Atw08] Jeff Atwood. Tivoization and the GPL, 2008. Online; accessed 8-July-2013. URL: `http://www.codinghorror.com/blog/2008/02/tivoization-and-the-gpl.html`.

[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.

[GK12] Ariel Gilbert-Knight. Making Sense of Software Licensing, 2012. Online; accessed 7-July-2013. URL: `http://www.techsoup.org/support/articles-and-how-tos/making-sense-of-software-licensing`.

[GV13] GPL-Violations.org. Regional Court Hamburg Judgement Against FANTEC, 2013. Online; accessed 7-July-2013. URL: `http://http://gpl-violations.org/news/20130626-fantec_judgement.html`.

[Lau04] Andrew. M. St. Laurent. *Understanding Open Source & Free Software Licensing*. O'Reilly Media, 2004.

[Upf07] Peter Upfold. The differences between the GPL, LGPL and the BSD, 2007. Online; accessed 8-July-2013. URL: `http://fosswire.com/post/2007/04/the-differences-between-the-gpl-lgpl-and-the-bsd/`.