COSC345 Week 13 Notes

The obvious question about this lecture is why it exists.

The answer is that we spend much more time reading code than we do writing it. So why do we teach much more about writing code than about reading it? Something odd there.

It's particularly urgent that we give a lot of attention to code reading these days because the world has changed. When I started programming, programs were written on punched cards. The longest program I wrote as an undergraduate was 3 boxes of punched cards (6000 lines). When a program is only 100 pages long it isn't too hard to read it. The Burroughs Algol compiler was a listing about 10cm thick, but it was really very well written, quite a pleasure to read. In each case, all you needed to know in order to understand the program was the manual for the programming language, which was itself only a couple of hundred pages long. When, as a PhD student, I met the UNIX operating system, the complete and rather usable documentation for the operating system, languages, and libraries was a couple of ringbinders: it was possible to take it home and read the whole thing over a week-end.

These days we have to deal with vastly bigger libraries. Software reuse is now a reality, and it's a painful one. Simply finding the right class in the Java Development Kit is fairly daunting, and once you have found it, you have to integrate information from all over the place.

In 2009 we taught a paper based on the Windows Research Kernel from Microsoft: it's 800 000 lines of code: none of us has read the whole thing, and none of us ever will. A good rule of thumb is that we can **carefully** read 100 lines of code per hour. Students should measure themselves to find out how fast they can read code **carefully**. This means it would take 8,000 hours to read the Windows Research Kernel. At 40 work-hours per work-week, that's 200 work-weeks. What with holidays and illness and time lost due to meetings and so on, another good back-of-an-envelope number is 40 work-weeks per year. (Some people use 2,000 work-hours per year: that must include overtime.) So that's 5 years just to read the sources. By the time you've finished reading the WRK, it's different, and you'd need to read it again. Do you suppose anyone at Microsoft has ever read the whole WRK? Or *could*?

I'm writing a Smalltalk→C compiler. The compiler alone is 12,951 lines of C (8,397 SLOC). Depending on how fast you read and how many hours a day you spend, that's somewhere between 1 and 2 weeks to read it all thoroughly. It's practical to read that much code as part of a commercial project, provided you don't do it often. But put it another way: allow NZD 300/hour for programmer time including all overheads (including cleaners, accountants, and so on), and each line of code you have to read thoroughly and carefully costs your business NZD 3.

The run-time library is another 12,418 lines of C (8,263 SLOC) plus 159,917 lines of Smalltalk (108,834 SLOC). You can do the arithmetic yourselves.

These days we *cannot* just read a whole program from start to finish the way we would read a novel. We can only read parts of programs, and finding those parts is one key skill, integrating information from them another.

This lecture was originally written before I had heard of Diomidis Spinellis's books "Code Reading" and "Code Quality". I cannot recommend them highly enough, and I hope to set some exercises in code reading based on those books. I

don't know what tools you will end up using, whether it will be Emacs + ctags, or whether you will use Spinellis's CScout, or the "Programmer's Bookshelf", or Eclipse or NetBeans or what. But you should take some lab time to practising using your selected tools.

For Oman we split COSC345 into Software Project A (development) and Software Project B (maintenance). The main difference between those parts, when it comes to the practical work, is the difference between writing a program yourself (using a possibly very large library) and rewriting part of a program other people have written. We did a project here once where students had to maintain a version of AWK. The changes the students had to make were not particularly complex; finding where to make the changes required understanding how the program worked, in the absence of a design document, and that was the challenge. Apparently this experience made some of the students dislike open source software. That was a very bad mistake on their part, because by the standards of commercial software, that program was particularly *easy* to maintain.

This also links, of course, to the lecture on reverse engineering, where the point is that sometimes, before you can make a change, you have to reconstruct the documents you need.

This lecture comes after the working backwards one because code reading in a large program is a problem solving activity, and the trick is to do it **consciously**, to read with explicit goals, and to develop explicit strategies.

People are astonishingly bad readers. I don't mean that they cannot take a page of text and read aloud what the words are. Most people can do that if their schooling was adequate. But even people with PhDs (or perhaps especially people with PhDs?) can completely misunderstand what is plainly before them.

Here's a typical example. The 19th century English poet Robert Browning and his wife attended a séance given by the celebrated medium Douglas Home. Browning got quite annoyed, left with his wife, and ended up forbidding her to have anything to do with Home. In later years Browning wrote a poem called "Mr Sludge the Medium", in which Mr Sludge, having been caught out, spills everything. Both the people who believed in Spiritualism and the people who didn't read Browning's poem as an attack on Spiritualism. You will find an article at the CSICOP web site that takes this view, for example. But in fact Browning has Mr Sludge say that despite his own trickeries, he has seen too much to believe that trickery is all there is; there is something there. That's Browning's *character* affirming the reality of psychic phenomena, not Browning *himself*. (This is another way in which people often fail to read correctly.) But it certainly isn't an attack on Spiritualism, only on *fake* mediums. It is, in fact, one of a series of poems in which discreditable characters are presented. One of them was about a bishop, and Browning certainly wasn't attacking the Christian faith. If you read English-speaking philosophers, it's amazing how much of their writing is "X attacked my article M, but the points X rebuts are points that M rebutted, not that it argued for" and the like. I have had some of my own writing misinterpreted in this way. There is a lot of reviewing about that makes one wonder whether the reviewer actually read the book or watched the film. For example, a review of the Lord of the Rings films attacked the book for glorifying barbarian war and not having any psychological depth, when in fact the book has as its main theme a refusal to use evil power, and the way in which the desire for power can corrupt. The hero of the book ends up refusing

the refusal, and if that's not psychological depth, there's no such thing. If we are so bad at reading ordinary natural language stories, how can we expect to fare better at reading the vast hypertextual machines called programs? If we don't watch out, we shall read what we expect to see, not what is before our eyes.

Reading is an active process. One of the exercises students are given in the Humanities is to summarise. You will, for example, be given a 20 page essay, and told "summarise the main points of the essay in 3 paragraphs". Making textual and diagrammatic summaries as we go is useful when we are reading programs. Some diagrams can be automatically produced by tools. Many tools will make UML diagrams from source code, for example. (Diomidis Spinellis has a nice one on his web page.) I provide my students with tools to make calling diagrams using the GraphViz toolkit to display them. Spinellis's CScout makes diagrams for C. But what we really need is *selective* summarisation.

Just as we design a program at many levels of abstraction, so in understanding one we need maps at many levels of abstraction. You can replace references to New Zealand with references to Oman or any other country in the road map analogy. One nice little exercise is to take a map of the University campus and ask "How big would a map of the whole country be at the same scale? How much would it weigh? If you wanted to find your way from here to (some specified place), would that be the most useful kind of map?"

Javadoc is presented for several reasons. One is of course that it exists and these days you just cannot avoid it. It has been widely imitated: there's PlDoc for Prolog, EDoc for Erlang, Haddock for Haskell, DOxygen for C and C++, and a host of other imitators, so knowledge about Javadoc is helpful with the others. But it's also worth pointing out the limitations of Javadoc. It is absolutely useless for finding things in the code, because Javadoc doesn't keep any pointers to the code. Haddock, in contrast, *does* keep pointers to code, which means that Haddock *is* useful for finding code. For example, visit http://hackage.haskell.org/package/arithmoi-0.4.1.1/docs/Math-NumberTheory-MoebiusInversion.html and notice the [Source] links. Even Haddock, however, is utterly useless for finding where things are *used*. If you want to change the interface of a function or method, you desperately need to know where it is used. CScout makes a nice example: it's a full hyperlinked display and navigation and query system for C. I've demonstrated the Smalltalk IDE in class because I know Smalltalk and its IDE well, and it does let you navigate between definitions and uses in both directions. However, even Smalltalk can't help you with code it does not know about. Packages that have not been installed are *not* found.

The issue about examples is inspired by Smalltalk and Pop11, and these days, R. The Pop11 system from Sussex had, for its day, a large library. Its equivalent of "man pages" included examples. Put the cursor over an example and press a key, and the example would run. Smalltalk practice encourages programmers to include examples of how to use methods in the code. Double click, press a key, and the example runs. Something that might be more appealing to your students is the R statistics package, see http://www.r-project.org. It has the best on-line documentation I've ever seen for an open source project, and again, its equivalent of manual pages comes with examples: `help(whatever)` will show you the text, and `example(whatever)` will run the examples in the page. Examples you can try are a very good way to document what something is supposed to do.

Test cases are nearly as good as examples. Arguably they *are* examples, just not always as well documented. The Java equivalent of examples would be JUnit test cases. The next revision of this lecture will have more to say about XUnit-style test cases as a support for code reading. (Of course, JUnit is a clone of Smalltalk's SUnit; this approach to testing came from Smalltalk, not Java.) The slide about "Look outside the code" is where I would put that in. Test cases are particularly nice because they typically come with setUp() and tearDown() methods that tell you what you have to do to get a class ready for use.

Whatever programming language you use, make sure you have a decent pretty-printer for it. Artistic Style (http://astyle.sourceforge.net/) handles C, C++, C#, and Java. There are more capable programs around, but that one's free and pretty useful. Provide an example of some code as it was written, and the same code after pretty-printing, to show that the structure the programmer intended may not be the structure that is actually present. Provide an example using syntax colouring or styling to show problems with comments and strings. (Problems that Ada and Erlang, which only have end-of-line comments, are largely free from.)

One of the handouts for this lecture was the manual for cscope. Get it as part of the cscope distribution at SourceForge. For a while I did contemplate replacing mention of cscope with mention of CScout, but for medium size programs I still find cscope more light weight to use.

I've used Unravel (http://hissa.nist.gov/unravel/), but development apparently ended in 1995. Slicing is worth knowing as a manual technique. http://www0.cs.ucl.ac.uk/staff/m.harman/sf.html is a good thing to hand out about slicing.

Coverage tools are one way to guide reading: run a test case through a coverage tool, and then any statement that was not executed cannot possibly be relevant to whatever happened in that test case. There are programs that monitor the execution of Java programs and direct your attention to what's in the trace.

This lecture is connected with the Inspections topic.
http://en.wikipedia.org/wiki/Software_inspection is the Wikipedia article on software inspection, with some useful links off it.
http://www.cs.otago.ac.nz/cosc345/resources/fistd.pdf is a local copy of the NASA Formal Inspections Standard and
http://www.cs.otago.ac.nz/cosc345/resources/fi.pdf is a local copy of the NASA Formal Inspections Guide.

Some practice in reading is important for this paper.

Some examples: http://www.cs.otago.ac.nz/cosc345/laitinen.txt
http://www.cs.otago.ac.nz/cosc345/cracker-c-P.htm
https://github.com/xslogic/phoebus