COSC345 Week 24 Notes

Reverse and re-engineering are about software maintenance. One thing that should be brought out in all of these lectures is that most programmer work is maintenance, not green-field development. You'll find various estimates around. Use the most up to date one you can find, but 80% of development as maintenance isn't a bad figure.

This is one reason why it's a good idea for the project for this paper to be a maintenance one. Students don't find maintenance as *enjoyable* as developing their own pet projects, but neither do employed programmers.

Now when you actually come to do some maintenance, you find that everyone who came before you was an illiterate fool. Nothing ever got written down that you need to know, except for stuff that turns out not to be true. (My own Smalltalk compiler is currently in that state. My original implementation ideas are written down, but in the course of actually building the program almost all of the original ideas were discarded. For example, I intended to implement each Smalltalk block as a separate class. Now I don't. I intended to implement characters and Booleans as full objects. But then along came Unicode, and I really wasn't willing to tie up a million objects, so characters, booleans, and nil are also immediate. I put a good deal of work into devising an efficient way to refer to global variables, only to discover in 2010 that it not only didn't pay off on a PC, as I'd expected, but it didn't pay off on the RISC it was originally devised for. So the design notes are about as useful as the design notes for the airship mooring tower for the top of the Empire State Building: that's not what got built!)

The basic idea is incredibly simple: before you can fix a program, you have to know what it is supposed to do and how it does it. If the documentation doesn't tell you, you have to write the documentation that you wish you had been given in the first place.

I give the students a personal anecdote. If you've had much experience, you'll have your own. Here's mine.

I used to work at a company that made a compiler for a high level language. Our system could be considered as made of

1. editor interface
2. parser
3. code generator
4. threaded code emulator for VM instructions
5. runtime system
6. library
7. documentation

One day one of the founders quit. A month or so later, we noticed that there was a mistake in the code generator. I was given the task of fixing it. In about 2000 lines of code, there were two comments. One was a copyright notice, and the other was four lines of code commented out with a note "this doesn't work". The procedures all had tolerably clear names, but all variables were either one capital letter or one capital letter followed by one digit, although the language allowed arbitrarily long names. This being a dynamically typed language, there were no type or structure declarations. It turned out that the code generator went through about 6 stages of rewriting, each with its own tree structure. I spent about two weeks going through that code, documenting all

the data structures and giving all the variables more meaningful names (such as Live_Variables instead of L). As I went, I wrote pretty-printing and validity-checking procedures for each data structure, so that I could check that I hadn't missed any cases (which of course I had, so it was a good thing the procedures were there to find out and tell me), so by the end of that time I was in good shape for making changes. This is very closely related to the stuff that Martin Fowler talks about in his book on Refactoring; the checking and testing procedures are related to test-driven development, which is touched on in that book. Once I knew what the data structures were, I was able to improve them and make the code simpler and easier to read. This actually speeded the compiler up by 25%.

There was one part of the compiler's code generator I really could not understand. Something very clever was going on in there, but what? I knew what it was for, and I knew how to tell if the output was correct, but I didn't understand the algorithm. Needless to say, that's where the bug was. Having made the compiler faster, I was now able to afford the time to run an extra pass after the buggy bit to tell whether the bug had occurred or not, and if it had, to repair the output. If there had been no other work to do, I'd have put a lot more effort into understanding that part, but sometimes near enough is good enough, and we never had any more trouble with that particular bug.

The really annoying thing here is that the original author *knew* what was supposed to be going on, he just didn't bother writing down, because it was always going to be him working on it. Except it wasn't.

One lesson from this is that internal documentation doesn't have to be big to be helpful. It could just be a matter of using intention-revealing names.

Martin Fowler's "Refactoring" book really has a small number of key ideas, and they are very important.
1. Before you can make the changes you really want to make (adding a feature or improving performance or fixing a mistake or whatever) you very often have to tidy up messy code. This tidying up is called refactoring.
2. Any kind of change, even refactoring changes, is risky. As a rough guide, even for good programmers, by the time a program gets to one million lines of code, any time you fix a bug you are likely to introduce at least one more. (Hercules and the Hydra.)
3. So you really need good test cases for the code you are working on and you want to run them after each change to make sure that you haven't broken anything.
4. This makes each change slower, because running the tests takes time, but it makes the change process faster, because your mistakes get caught much sooner and are easier to undo.

What I'm saying here is that developing test cases can be a good way to develop your understanding of the code you are working on, and once developed, they can be a good way of recording that understanding.

One specific form of reverse engineering is trying to reconstruct the source code from the object code. You can certainly reconstruct assembly code from the object code. It won't be the assembly code that was originally used, but it will be assembly code that would have resulted in the object code you have. I once saw a COBOL program that was produced this way: there was an object program of some economic value on machine X, which was now dead. A disassembler reconstructed assembly code. Another program turned that assembly code into COBOL. The result wasn't really readable, but it could be compiled

and executed on the new machine, which was more than the old object code could.

At some point you might want to mention Canterbury Pascal as an example of dealing with legacy software. Canterbury Pascal is now available from http://www.mhccorp.com/mhcform.shtml (it's a commercial product, not free, but it's pretty cheap). There are two versions. The US$56 one compiles Pascal to JVM .class files. The US$112 version compiles Pascal to Java.

Disassembling stuff you don't own rights to is illegal in many countries. I know this is in the slides but it bears repeating. There is nothing wrong with with disassembling open source software or software that you own, but make very sure you really do own it. There was a project I was involved with in Australia, where the aim was to develop a completely new system from the documentation for the old. We weren't even going to be in the same city as any copy of the original program. However, the large American company that produced it said "we own *all* the ideas, we *will* sue if you do this." You can understand their point: they were charging AUD two *million* for each change. Their position was legal but highly immoral. Needless to say, the project was cancelled. How the Australian government failed to end up owning something they had paid a very large amount of money for is an interesting question, but not really a software engineering one. The New Zealand government deliberately sought *not* to have the intellectual property for Novopay. Go figure! And then Talent^2 decided they couldn't commit to maintaining Novopay, so the government *had* to buy the IP rights anyway or they wouldn't have been able to fix it!

Dealing with legacy code can be a problem because not only may it be hard to find documentation for the program, it may be hard to find accurate documentation for the programming language or the operating system. If someone gave you a program written for MS-DOS 2.0 in Microsoft Fortran, would you be sure what it did and how it did it? Where would you get manuals? Sometimes the manuals are not accurate. CWI in the Netherlands have done a lot of work with programming language description tools called ASF+SDF. See http://www.asfsdf.org/ In related work, someone or other (I cannot locate the paper because we moved buildings and a lot of stuff is still in boxes, it may well have been the Amsterdam group) did things like automatically extracting grammars from IBM manuals, and found that they were seriously buggy. Actually reconstructing a working grammar for old COBOL was surprisingly hard. But it had to be done before they could analyse the old code.

A lot of "legacy" code exists and does need to be maintained. Modern Fortran contains a very pleasant "structured" programming language with records and pointers and recursion and modules and vectorised operations and all the stuff you'd want — the latest version even has objects — but a lot of old code doesn't use that stuff. Here is some Fortran 77 code, from which I have stripped the comments.

```
      REAL FUNCTION GAU(Z)
      REAL P, PI, X

      X = ABS(Z)
      IF (X .GT. 5.5) GO TO 10
        P = EXP(-((83.0 * X + 351.0) * X + 562.0) * X /
     1   (703.0 + 165.0 * X))
        GO TO 20
   10 PI = 4.0 * ATAN(1.0)
      P = SQRT(2.0/PI) * EXP(-(X * X/2.0 +
     1    0.94/(X * X))) / X
   20 GAU = P/2.0
      IF (Z .GT. 0.0) GAU = 1.0 - GAU
      RETURN
      END
```

Here's the same thing in Fortran 95.

```
real function gau(z)
   real, intent(in) :: z
   real, parameter  :: pi = 4.0*atan(1.0)
   real             :: p, x

   x = abs(z)
   if (x <= 5.5) then
      p = exp(-((83.0*x + 351.0)*x + 562.0)*x/(165.0*x + 703.0))
   else
      p = sqrt(2.0/pi) * exp(-(x*x/2.0 + 0.94/(x*x))) / x
   end if
   if (z > 0.0) then
      gau = 1.0 - p/2.0
   else
      gau = p/2.0
   end if
end
```

You don't need to know much Fortran to find this much more readable. What would it be like to convert from the old form to the new form automatically? That's (one kind of) dialect conversion. C and C++ are not, despite uninformed claims, compatible; the C++ standard lists page after page after page of incompatibilities with C, starting with the obvious (sizeof 'x' is 1 in C++ but could be 2, 4, or even 8 in C) and proceeding through quite surprising things. How if we could automatically convert from C to C++? I have some old C++ code that C++ compilers no longer accept; I'd rather like to run it, but it isn't worth the effort to rewrite thousands of lines by hand, especially as I don't have C++ documentation of that vintage. How if we could convert automatically?

Why would we convert? Because the old language isn't available any more (on the Macintosh we lost first Think Pascal, then MR Pascal, then Metrowerks Pascal) or because we want to maintain the code using new tools that only deal with a modern dialect, or simply because it's much more maintainable in the new dialect.

Another personal anecdote: I once wrote a dialect converter from IBM Prolog to Quintus Prolog. It wasn't possible to do a perfect job, but it was quite enough to make a key customer happy. It fully preserved all comments, and even put them in reasonable places. Some people will tell you that dialect converters/language translators lose comments. They are wrong. (The current user-land Haskell parsing kit from Hackage can preserve comments well enough to be useful.)

It's important to remember that reverse engineering (working backwards from products of the forward engineering process to their inputs, like requirements, specifications, documentation, test cases, *etc*) and any kind of refactoring, these things are only means to an end, and that end is re- engineering: moving forwards to a better state of affairs. There is no point in doing this unless the old code still has economic value, and unless all this effort will enhance this value.

According to a 2009 survey commissioned by Micro Focus and conducted by Harris Interactive:

1. 70–75% of the business and transaction systems around the world run on COBOL. This includes credit card systems, ATMs, ticket purchasing, retail/POS systems, banking, payroll systems, telephone/cell calls, grocery stores, hospital systems, government systems, airline systems, insurance systems, automotive systems, traffic signal systems.

2. 90% of global financial transactions are processed in COBOL.

3. The language supports over 3×10^{10} transactions per day.

4. The average American still interacts with a COBOL program 13 times a day.

5. There are 1.5–2 million developers, globally, working with COBOL code.

6. There are around 2×10^{11} lines of COBOL code in use.

7. Around 5×10^{9} lines of new COBOL code are added to live systems every year.

8. The investment made into COBOL systems over the past 50 years is said to be worth about USD 2×10^{12}.

Think about all the COBOL'85 code that needs to be updated to COBOL'02. Wipe the drool off your chin. Think about the companies wanting to convert COBOL to Java. Wipe it off, I say! Think about the missing documentation from 20-year-old software. I *said*, wipe it off. Think about 40 million lines of code running the New Zealand tax system. Oops!

For working with source code, you can't go past the Rascal http://www.rascal-mpl.org/ project from CWI or the older Meta-Environment http://www.meta-environment.org/ that preceded it. For working with object code, there are a number of older systems, notably the Executable Editing Library http://pages.cs.wisc.edu/~larus/eel.html by Larus *et al.* There is a newer system called the University of Queensland Binary Translator http://www.itee.uq.edu.au/~cristina/uqbt.html. That can convert binary code from one architecture to

another, possibly an earlier or later version of the same machine, possibly quite a different one. Figuring out switch() statements was surprisingly tricky.

For an example of emulators, you might like to look at Andrew Trotman's Poly Preservation Project http://www.cs.otago.ac.nz/homepages/andrew/poly/Poly.htm where you will find an emulator for the Poly computer.

For another example, Apple Macintoshes used to use a CPU called the Motorola 68000. Other companies did too: I used to have an Atari 520ST running an operating system called TOS with gui kit GEM. There is now an emulator called ARAnyM (aranym.org) which emulates Atari ST, TT, and Falcon machines running TOS, FreeMiNT, MagiC, and Linux-m68k on modern machines, so old software written for those machines can still function.

Disassemblers: your system may have one called 'dis' or 'disas' or 'ndisasm'. For Java, there's 'javap', which can tell you quite a lot about JVM code you don't have sources for. You really should demonstrate 'javap' in class. The Java Decompiler 'JD' at http://java.decompiler.free.fr is pretty impressive, actually. There is a program called HT which can handle x86 binaries and I believe Java class files; target platforms are Linux, *BSD, and Windows. See http://hte.sourceforge.net/.