COSC345 Week 14 Notes

We normally expect the COSC345 projects to go very well. The code isn't always brilliant, but people are sparking with ideas and get something interesting done.

This year, something has gone wrong. I want to find out what.

So what did go wrong?

1. Was it a resource problem? Could you not get to the kinds of machine you needed when you needed to?
2. Was it a requirements problem? Did we not say clearly what we wanted you to do?
3. Was it a specification problem? When you turned what we said into your own words, did you leave stuff out? Did you add too much detail?
4. Was it a planning problem? Did you give yourselves enough time for every step you needed?
5. Was it a tools problem? Did you have trouble using compilers or editors or IDEs or debuggers?
6. Was it a motivation problem? Since it wasn't *your* idea in the first place, did the project leave you yawning so you didn't get stuck in?
7. Was it a fear problem? Did the project look so hard that you just sat there paralysed? (This even has a name, "ambitious paralysis".)
8. Was it a personnel problem? Did group members have a hard time getting on with each other?
9. Was it a knowledge problem? Were there things you needed to do that you didn't know how to do, and did you not know how to find out how to do them?
10. For example, were you able to describe the structure of a command in a clear and unambiguous way and relate it to the actions your program would need to take as it processed each keystroke?

Perhaps the most important question is "what do we do about it", but we can't address that until we know what "it" is.

One thing that will clearly have to be done is to scale the problem back somehow. But out of all the features Andrew asked for, which will save the most effort for the least harm?

The obvious question about this lecture is why it exists. Some years ago, at another university, I noticed that even fourth-year students were having difficulty with what I regarded as fairly basic problem solving. Give them a collection of functions and a task to be solved with them such that the functions could be combined in only one way that could possibly work, and they couldn't find it. It's often the case that even fairly smart people cannot figure out problem-solving strategies to begin with, but once you tell them how to do it, they can not only use those strategies but can start to develop their own.

So I tried to boil down Polya's "How To Solve It" book into a single lecture. It could be done much better. I could do it much better, if only I had more than one lecture to do it in. As you revise these papers to better suit your needs, please try to make **conscious** problem-solving part of every year. Polya's book is now available electronically, a local copy is http://www.cs.otago.ac.nz/cosc345/lecs/L14-polya.pdf. It's *still* worth buying a paper copy.

Here's a recent example. I needed to transcribe a numerical algorithm into a programming language that — for good reason — doesn't have infix operators.

I made a lot of mistakes doing this, so I decided I needed a program to do it.
1. Is there a program to do that already?
2. Since there isn't, what am I going to need?
3. Is there a program to generate lexers in my chosen implementation language?
4. Is there a tokeniser component for it?
5. How do I configure it?
6. How do I call it?

Back in the 1970s there was an effort in the Computer Science community to develop methods of programming that would be less error prone. The books by Dijkstra, Reynolds, and Gries in the full reading list are part of this. Indeed, the whole "Year of Programming" series is well worth a lecturer's time to read it. This got swamped by more "industry-related", "practical" approaches (*i.e.,* how to make large amounts of badly broken code using cheap labour). These days the formal approaches are making something of a comeback, and the capabilities of many formal verification and testing tools are amazing. Look at Daikon, for example: a program that runs Java programs on test cases and infers invariants from traces. However, the 1970s school said that if you want to develop reliable software, you don't just hack away and then at the end try to show that your program is right, you try to "develop the program and its proof hand-in-hand" so that the program is correct *by construction*. Now that there are tools like ESC/Java 2 for annotating Java programs with preconditions, postconditions, invariants, and so on, and automatically checking some of them, this **conscious** development of algorithms needs to be back in the curriculum.

Note added in 2013: ESC/Java 2 has been superseded by OpenJML jmpspecs.sourceforge.net, which is up to date with JDK 1.7. It now requires a separate SMT (Satisfiability Modulo Theories) solver to solve the checks that it generates from the source code.

Note added in 2014: SPARK is a similar tool for Ada, which continues to thrive. It comes free with the GNAT Ada toolset, which is a free compiler for Ada 2012. Ada 2012 drastically improved Ada's support for assertions.

We are faced with stunningly complex systems and the potential consequences of error are horrifying. The book "Normal Accidents" deserves mention here.

We can say that there are three basic ways to program: programming by patterns, programming by proof, and programming by punishment (trial and error).

When people are just learning to program, really it's programming by punishment. You try something, and it doesn't work. So you try something else and it doesn't work. So you try something else and you **think** it works but it really doesn't, but you don't find that out until too late. That's not software engineering!

Programming by patterns refers to knowing a lot of schemas, recognising a problem, and saying "I know a schema that fits that problem, let me instantiate it." (Of course, to do this, you have to have the idea of a schema or pattern and the idea of instantiating it, which means that you already have to have at least the beginnings of the idea of code fragments as fluid transformable stuff rather like algebraic expressions.) This goes back to classic work by Soloway and the famous AI "Programmer's Apprentice" project of Rich, Waters, and Shrobe. There's work on "schemas" and "algorithmic skeletons". The Gang-of-Four "Design Patterns" book came much later, and had an Object-Oriented

Programming focus, but was basically about the same thing. The Patterns literature is a good source of schemas. Learning to program really is, amongst other things, learning a whole lot of (problem : schema) pairs that you can use to understand other people's code (I see this schema so it must be to solve that kind of problem) and to write your own.

But what do you do when you find a problem that isn't fitted neatly by some pattern that you know or can find? That's when you turn to programming by proof. Trying to develop an algorithm and its proof of correctness *together* is the only way I know of coping with **new** problems. The "proof" need not be carried out. It certainly need not be formal. But you must have **some** reason to believe that your method works.

A good simple example is binary search. Once you recognise the problem "search for an item in a sorted collection" binary search should immediately suggest itself, and if you have a "canned" binary search in your library (UNIX C traditionally has `bsearch(3C)`) you are finished. But if you have to develop one, it is astonishingly hard to get right.

A theme that comes up early in this lecture is errors. One of the themes of the whole paper is errors. To err is human. http://en.wikipedia.org/wiki/Human_error is a good place to start with. *Human error: models and management* (James Reason, BMJ. http://www.bmj.com/cgi/content/full/320/7237/768) makes the extremely important point that "We cannot change the human condition, but we can change the conditions under which humans work". But the web site I constantly refer to in this paper is Panko's: http://panko.shidler.hawaii.edu/HumanErr/ The section on "error rates in programming" is of high relevance to this paper. One paper is http://www.cs.otago.ac.nz/cosc345/lecs/L14-humerr.pdf

I find the distinction between understanding a problem, planning a solution, carrying it out, and examining the result (taken from mathematics) directly relevant to computing. Understanding a problem often involves activities that look more like play than work. Later on we talk about building prototypes, programs that are not going to be part of the final solution. This is where they come from: exploring the problem space and helping to make sure that we have some idea of what we are trying to do. Planning a solution is like designing the architecture of a program, and carrying it out is like coding it. Examining it is of course testing and other kinds of checking. During the lecture I constantly draw parallels between the mathematical context Polya was talking about (finding proofs, finding counterexamples, constructing mathematical objects with various properties) and programming, and even answering examination questions. The slides do not and are not meant to spell out all the parallels.

These ideas apply at the level of coding (I want to test whether a mouse click landed in a particular polygon, how do I do that?), at the level of system architecture (I need a high reliability location service, how do I do that?) and at all the levels in between.

Sometimes what looks like a minor point can be of great practical significance. For example, "Can you change the problem to make it easier to solve" leads to a discussion of the difference between private interfaces, which can be changed freely without anything outside the module caring, and public interfaces, which may be difficult or impossible to change, hence the higher importance of getting the public interfaces right. It also leads to mention of making it easier to devise loops by splitting variables into multiple variables from which

the original value can be obtained, but which are easier to update piece by piece. It also leads to mention of Plass's Stanford thesis http://www.cs.otago.ac.nz/cosc345/lecs/L14-plass.pdf on how to break a sequence of lines into pages, done under Donald Knuth's supervision. Knuth devised an efficient algorithm for optimally breaking a sequence of words in a paragraph into lines. Plass wanted to do the same for lines and pages, but Chapter 2 of his thesis (probably more than half of his research time) was devoted to showing that a series of natural descriptions of the problem have no computationally feasible solution.

Some of the slides have points labelled "-P-" that come from Polya, and also points labelled "-O-" that I added. You can think of the "O" as standing for "O'Keefe" or "Otago". I labelled them because I did not want to misrepresent Polya. You may wish to make the distinction another way, or you may decide that it is unnecessary.

It has been said that judges sometimes use backwards reasoning: listen to the case and figure out what would be a just response to it, then work backwards to find legal principles and cases to explain why that's the legal response.

The material about AI planners could be discarded; it's there to make the point that this advice is not airy-fairy waffle, it's about techniques that can be and were built into programs that actually do stuff. (At least some NASA interplanetary missions are done this way, and the US military used to use such a program to figure out how to load aircraft so that essential equipment could be unloaded quickly.)

The inspection handout is an extract from the NASA Formal Inspections Standard and the Guidebook; this material is freely available on the Web. There are other things that would do just as well.

In 2010 I decided to hand out the article "Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments", by Justin Kruger and David Dunning, Journal of Personality and Social Psychology, December 1997, Volume 77, Number 6, pages 1121-1134. An electronic copy can be found at http://www.cs.otago.ac.nz/cosc345/lecs/L14-unskill.pdf The *Dunning-Kruger Effect* basically says that incompetent people are too incompetent to realise they are incompetent. Its relevance here is that it is about *metacognition*: thinking about thinking. To learn to make more effective plans, people have to start by realising that there is something that they do need to learn.

Daniel K. Lyons wrote in http://news.ycombinator.com/item?id=4074497 that "Bad programmers I've dealt with tend to share certain traits:
• their code is large, messy and bug-laden;
• they have very superficial knowledge of their problem domain and their tools;
• their code has a lot of copy and paste, and they have very little interest in techniques that reduce it;
• they fail to account for edge cases while inefficiently dealing with the general case;
• they're always rushing around putting out fires, trying to look like heroes battling vast problems against impossible odds;
• they never have time to comment their code or break it into smaller pieces;
[and]
• Empirical evidence plays no role in their decisions.

It would be hard to self-test, but some clues would be: do you think you're the best programmer in the world? Do you find code with a lot of functions

messier and harder to understand than code with only a few large functions? Do you routinely copy code from one place to another and make a few small changes to it? Do your programs tend to be a few huge files or lots of small files? When you're asked to make a change, do you usually have to touch most of the code or just a small chunk of it? If you say 'yes' to most of these, you're probably bad. If not, you're probably alright. :)"

Does that sound like my earlier comment about industry trying to "make large amounts of badly broken code using cheap labour"? It should! Our third-year students are *not* expected to be great programmers. They *are* expected to be active life-long learners of how to be better programmers, and to reflect on what they are doing. If you are writing repetitive code, why? Can you refactor that as a function or a class or a template? If not, how about a macro or an AWK script? I remember one student's MSc, half of which was a listing of a Java program, about half of which could have been automatically generated by 3 pages of tables and about as much AWK. Mind you, this can be taken too far, as witness the AI joke "I'd rather write programs to write programs to write programs than write programs to write programs."

In single-inheritance object-oriented languages, one reason for code duplication is when two classes have a common interface and the implementation would be the same but cannot be inherited. You could use M4:

```
// common-code.m4
   boolean isEmpty() {
      return size() == 0;
   }
   void toArray(E[] a) {
      final int n = a.length > size ? size : a.length;
      for (int i = 0; i < n; i++) a[i] = get(i);
   }
// Thingy.java.m4
public class Thingy {
   // other stuff
   include(common-code.m4)
}
// Whatsit.java.m4
public class Whatsit {
   // other stuff
   include(common-code.m4)
}
# Makefile
Thingy.java : Thingy.java.m4 common-code.m4
m4 Thingy.java.m4 >Thingy.java

Whatsit.java : Whatsit.java.m4 common-code.m4
m4 Whatsit.java.m4 >Whatsit.java
```

This is not ideal, but it's better than repeating a lot of code. My Smalltalk system contains a chunk of C code implementing numeric functions like sinc() which has to be done three times, once each for float, double, and long double. So I write it once and generate it three times.