

COSC345 Week 22 Notes

There is a lot of material available on the web and elsewhere about testing, especially from the "Agile" people. These slides were heavily influenced by Boris Beizer's books.

Important issues are

1. why do we need to test?
2. what do we need to test?
3. how do we test?

The first point is simply that there always are mistakes in programming. I keep on telling the students the following rule of thumb: one mistake every twenty-five lines before checking, for most kinds of text typed up by a good typist. For background on this kind of thing, it's worth looking up ("human error" rate). At the time of writing, Ray Panko's site was at the top of the list. Pretty much everything near the top of that list is worth reading. I've been thinking that maybe there should be an entire lecture just on the subject of error.

In particular, it's worth noting that there is no such thing as an error that can be solely attributed to a human being. Every human mistake happens while performing some task in some technical and social context. The context can do much to reduce or increase the frequency and impact of mistakes. The figure I quoted above, 1 mistake every 25 lines, relates to the *typing* task, which isn't quite the same as the *word processing* task. For example, I am far more productive and make far fewer mistakes using SGML or LaTeX than using Word: Word has its own ideas about what I *ought* to be doing, and fighting it not only wastes my time by diverts my attention from what I *really* ought to be doing. Word increases my error rate. I'd also expect my error rate to be much higher typing French or Latin than typing English, and as for typing Arabic, I couldn't do it no matter how much you paid me or how much you punished me for getting it wrong.

Different kinds of mistakes include (1) mistakes in perceiving the situation (see "change blindness" for some really scary stuff about people not noticing stuff), (2) mistakes in deciding what to do about what we've perceived, and (3) mistakes ("slips") in carrying out our plan.

One of the key points in Software Engineering is that the longer a mistake goes undetected in a system, the harder (work) and costlier (money) it is to correct it. We need tools and processes that will help us deal with mistakes.

One way a tool can help us is to reduce the number of mistakes we make in the first place. It can do that by being easy to think about. For an example or two of doing this wrong, look at the C standard library. The `fputs()` function adds a newline to the string you write, the `puts()` function does not. The `fgets()` function takes a length as well as a buffer and retains the newline, the `gets()` function does neither. The `fprintf()` function takes the stream as its first argument, the `fwrite()` function as its last. I've been programming in C since 1979 and still have to keep on looking up the manual, something I don't have to do using Ada IO or Scheme or Prolog or ... Or take the `lseek()` function in the POSIX interface: it has three arguments, all of them some kind of integer. Get them in the wrong order, and the C compiler won't notice. One of my favourite examples of a language that helps in this kind of way is Smalltalk. With procedure calls like

```
i := anArray findNext: item from: lastPlace to: anArray size
      ifAbsent: [self error: 'no next item'].
```

it is extremely hard to get arguments (`anArray`, `item`, `lastPlace`, `anArray size`, the block) in the wrong order.

Another way that a tool can help us is to reduce the amount of work we have to do. 'make' is an example of this. No matter how complex a UNIX program is, I expect to be able to install it by doing

```
./configure --prefix=$HOME/local
make
make install
```

All the complexity is hidden inside a repeatable script. Even within a Makefile, there is a system of defaults and macros to reduce the amount of code one has to write. When people found they were repeating stuff within a Makefile, the ability to include definitions from another file was added, so things could be written once *and maintained in one place*. Scripting languages like Javascript, Python, and Lua try to help reduce the number of mistakes by reducing the amount of code you have to write. But you don't have to limit yourself to scripting languages. Erlang claims, with empirical evidence to support it, a factor of 5 to 6 reduction in the number of lines of code you need compared with Java. Haskell adds very strong type checking, and reduces the line count still further. Even things like the old UNIX `lex(1)` and `yacc(1)` programs help with errors by simplifying the code writing task so that you have fewer chances to make mistakes. I've often found that `lex` isn't quite good enough for some lexical analysis task, and typically when I rewrite from `lex` (with embedded C code for various tokens) to hand-written C, the size of the file at least triples (on the other hand, so does the speed...)

A third way that a tool can help us with errors is to catch them as early as possible. Yacc, for example, reports ambiguities in a grammar. If the grammar were used to make a recursive descent parser, those mistakes might never be noticed, or not for years. Programming languages with strong type checking can catch errors before the program runs. For example, using Java 1.4, you might put a String into a collection and later pull it out under the impression that it was an Integer. In Java 1.5 and later, where classes can have type parameters, the compiler can easily catch this mistake at compile time. (As could Haskell, Clean, Mercury, Eiffel, C++, and even Ada-81).

A lot of research went into devising "safe" programming languages over the years. Recent years *have* seen something of a retreat from C and C++, which are about as unsafe as you can get short of assembly code, especially C. But they have not been a retreat to the ground of statically checked languages. They have been a retreat to object-oriented languages where a great deal of type checking has to be delayed till run time (that is, after all, the *point* of Object Orientation) and to scripting languages like Perl and Javascript where there is even less compile time checking. This is not necessarily perverse: strong static checking is one way to reduce errors, reducing the total code volume is another. The relevance to testing is that it is only mistakes that you can be *certain* are not present that don't need to be caught by testing.

With Javascript, you are *certain* that there are no storage management errors. (No failures to allocate, no failures to free, no double frees.) So you

don't have to test for those. In fact you are wrong to be certain, because there *is* an analogue of failure to free, and that is when a data structure is retained long after it is really useful because there is still a variable or a slot of some object that points to it. We *do* need memory retention diagnostic tools, and we do need to run them on JavaScript and Java. (The Haskell community have such tools for Haskell; the last time I used Clean there was something similar for it. I don't know enough about JavaScript tools to say what it has. I do know that such tools are becoming available for Java.)

If your language doesn't provide suitable guarantees, you may be able to use some tool that checks that your program belongs to a subset of programs that is better behaved. For example, consider something like the `strcpy()` function in C. What happens if either of its arguments is a NULL pointer? It's not defined by the standard. If you are lucky, you get a segmentation violation (UNIX) or general protection fault (Windows) or something similar. If you are unlucky, some bit of low memory gets overwritten. Wouldn't it be nice if we could say "this argument, or this variable, is never supposed to be NULL" and have the compiler, or something like a compiler, check that it's right? Yes it would, and the Splint checker for C (www.splint.org) can make just such a check. Exactly the same kind of problem exists in Java, although you are sure to get a language-defined exception if you pass null to something that expects a real object. And annotations are now defined as an add-on to Java so that a suitably annotated class can be trusted to be free of such problems.

There are always other kinds of mistakes. If people can be relied on to make mistakes (Panko's web site has some figures for how often people make programming mistakes, as has Beizer's big book), and if the language and compiler and any other static checking tools you have don't promise to find them, then you *will* get mistakes in your programs even after they've been checked.

One way of dealing with that is Formal Inspections, the subject of another lecture. Inspections are amazingly effective, and they are good value for money, but if you have large amounts of code, it's very hard for people to read all of it, and you don't want to keep on inspecting over and over again as you change something. We need something that we can do at least semi-automatically *before* spending much time on inspection.

That's testing.

You thought it was static checking? Yes, it's that too. Testing only applies to things that can be "run" in some sense, and static checking doesn't have that limitation.

People often confuse testing with debugging. But debugging is a response to failed tests. We can't know in advance what will need to be debugged: if we did, we would fix it sooner. But we *can* know in advance what will need testing. We can plan testing early, and we can start writing test cases before the code is finished. Indeed, the Test-Driven Development approach insists that you must write the test cases first. I'm still in two minds about Agile methodologies. But it is quite clear that trying to write test cases is a good way of ensuring that you understand the system requirements well enough. If you don't have any idea how to test the system you're planning to build, how will you ever know that you've done it right? Indeed, there is a slogan: "if it isn't tested it doesn't work!"

The great thing about testing is that we can get the computer to do the

work for us. We can build up a set of tests for some component, a method or a class or a package or whatever, and every time we make a change, we can rerun the tests. As we encounter mistakes, we construct even more tests specifically for those mistakes, and we keep on rerunning them.

In the Good Old Days (bad old days?) when computer time was extremely expensive and programmers were cheap, it made sense to try to keep the amount of testing down. In New Zealand, a programmer with less than 5 years experience typically gets about NZD50,000 a year (checked in 2009; I don't know how current the figures are). You should expect overheads to match salary. Suppose to keep the arithmetic easy we say that the cost to a company of a (cheap!) programmer is \$80,000 a year. You can get quite a decent desktop machine for \$800 new. I've seen a 1.4Ghz Mac Mini for sale at \$120 US, let's call it NZD200 to keep the arithmetic simple. So you can buy *one hundred* new computers a year for the price of one programmer, or *four hundred secondhand computers* for the price of one programmer. (Of course electrical power will then be an issue. Call it 100W per machine, 400 machines is 40 kW, which is really serious electrical power. Say the electricity price is 20 cents per kilowatthour, as it has been at times, that's about \$70,000 per year in electricity. So once you've bought those machines, they'll be about as costly to keep as a programmer. This really needs to be properly costed.) The point is that we can now afford to do a **lot** of testing as long as we don't have to involve people in running the tests.

Test harnesses are important. In the last decade, a framework that originated in the Smalltalk world (SUnit) has spread to several other languages (e.g., JUnit for Java, eUnit for Erlang). I wish very much that we had time in this course to spend several weeks on testing and these unit testing frameworks in particular. They really do make a huge change to how easy it is to test.

There is another approach which comes from the Haskell world, and has spread to Erlang, Scheme, Lisp, Perl, Python, Ruby, Java, Scala, F#, and Standard ML, and that is the QuickCheck approach. (The Wikipedia page is a good place to start.) In one phrase: specification- based random testing.

Random testing is amazingly powerful. The Haskell version of QuickCheck gets enormous help from the Haskell type system, but as shown by the list above, it's being used in languages with no compile- time types as well.

The slides show a minimal shell-based test harness, not because I think this is the most important kind of testing, but because it is small. This year I intend to show a JUnit example and a QuickCheck example as well, but in handouts, not in the slides. The material on test plans should be accompanied by a test plan template. Mine was not available when I wrote these notes due to a reorganization of the Department's file servers. The one at www.softwaretestinghelp.com will do. I was guided by the IEEE testing standard, which is copyright material I can't give you.

Brian Marick has a lot of good stuff about testing at his web site, including too stuff from his old web site. I have one of his stickers, "to be less wrong than yesterday" on my office door.