COSC345 Week 17 Notes

This continues the theme of "errors are normal, so what are we going to do about it"?

Let me introduce a paper I rather like, "Learning to Think Mathematically: Problem Solving, Metacognition, and Sense-Making in Mathematics", by Alan H. Schoenfeld (http://mathforum.org/library/view/71348.html). The part that's relevant here is on p63: "As [the instructor] moves through the [class]room he reserved the right to ask the following three questions at any time:
1. What (exactly) are you doing? (Can you describe it more precisely?)
2. Why are you doing it? (How does it fit into the solution?)
3. How does it help you? (What will you do with the outcome when you get it?)"

These questions can and should be asked during programming exercises. Eventually, the students need to start asking themselves, automatically. These are also questions that should often have answers in comments. The "what" can be readily seen from the code, but the "why" is far less obvious and "how it helps" relates to the next level up in the layers-of-maps.

Static verification is all about checking that we are really doing what we meant to be doing and that it makes sense to do that. I sometimes tell students that a hallmark of software engineering is the *controlled* use of redundancy. You don't want to say the same thing over and over again. I have been known to complain that my students are not lazy enough; they will do any amount of repeated work rather than stop and think about what they are doing. On the other hand, if you eliminate redundancy completely, there is nothing left for a consistency check to find. This is one of the reasons why I still find Java utterly inadequate for software engineering compared with Ada: all integral types smell the same, so errors involving the use of the wrong integral type, or worse still, the right type but the wrong variable, cannot be detected. (More about Java later.)

The "static" in "static verification" is to distinguish it from testing and run-time checking for null pointers, array bounds checks, and contract violations (assertions). Both are important, but stuff to be checked at run time has to be runnable, and stuff to be checked statically doesn't. "Non- executable items" emphatically includes documentation, and emphatically includes
1. spelling
2. punctuation
3. grammar
4. cross-references (no dead links!)
5. consistency
6. remaining within a controlled vocabulary *etc* (look up Attempto Controlled English, especially at http://attempto.ifi.uzh.ch/site/pubs/)

The figure of 50% to 90% of faults findable by static means comes mainly from Boris Beizer's great book, but also from papers on formal inspection. These days I would say that the higher end of the range is closer to it *if* programmers put the work into annotations. The state of the art in checking tools has advanced *amazingly*, but you have to give them something more to check than is expressible in the standard language.

This is a good time to mention some tools. For C programmers, the classic static checking tool is lint. The history of lint is often misrepresented. It would be a good idea to get a copy of the original lint paper and hand it out (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841). On a 16-bit machine, there simply wasn't *room* for code generation and high quality error reporting. Indeed, in later versions of PDP-11 UNIX, there wasn't even room for error messages in the compiler; they had to be held in a separate file. lint was the first well known program checker to exploit the fact that although a refusal to produce code for a program must be based on a clear and unambiguous violation of language rules, *warning advice* need be neither sound (false positives are tolerable if there are not too many) nor complete (false negatives are tolerable because it's an easy proof from the Halting Problem that not all mistakes can be found algorithmically). Using a commercial compiler (Sun C) on a commercial system (an UltraSPARC II; second-hand value somewhere around USD 50) I enjoy the use of a real lint, which has advanced considerably since 1978. I am amazed that gcc still doesn't come with a companion "glint". You can approximate it. I have a script with that name, whose contents vary slightly from system to system, depending on what each gcc will do. One version is

```
#!/bin/sh
#   NAME        glint
#   AUTHOR      Richard A. O'Keefe
#   PURPOSE     Use GCC to get as much checking as possible.
#   I want -O2 -Wall to catch uninitialised variables, but that
#   means I can't use -fsyntax-only.  Hence the use of -c.
exec gcc -std=c99 -c -ansi -pedantic -O2 -Wall -W \
     -Dgets=DONT_USE_GETS -Dlint\
     -Wunused -Wundef -Wcast-align -Wwrite- strings\
     -Wshadow -Wpointer-arith -Wnested-externs - Winline $@
```

This does *not* match what a good lint can do for you. There is a free "lint", only better. It's called splint, and you get it from www.splint.org. Sadly, it's not being maintained as actively as it was, and the only C99 feature I know it handles is mixed declarations and statements. (Trying to add just one c99 feature would make a *wonderful* maintenance project. Declarations in 'for' loops would be a good one.) It used to be worth staying with c89 just for the sake of the extra checking that splint can do. In particular, splint can verify at checking time that there are

1. no null pointer references
2. no uninitialised variable uses (including pointers)
3. no doubly freed pointers
4. no memory leaks
5. no buffer overruns

The new 'clang' compiler is pretty compatible with gcc and has even better static checking abilities. With the --analyze option it can do everything old lint could and things that lint could only dream of.

There is a commercial tool called Coverity which does a fantastic job of everything except running on my machine. We have it. Any business developing software in a language Coverity supports should be using something like it or a rival product.

In 2011 I head about the Microsoft tool PRE*fast* and the annotation language SAL it supports. Apparently current releases of the "Team System" version of Visual Studio support this. The command line switch is /ANALYZE. This is typical Microsoft. It does amazingly useful things and is horribly ugly and seems designed to make porting to other environments hard. That said, anyone writing software in C for Windows had better have an *extremely* good reason for not using it, because while it can't do all the things that splint can, it can detect buffer overruns, and it can do some things that splint cannot.

I believe that the commercial FlexeLint product can do some of the things that splint does, and more, but splint is free. One of the key ideas in splint is annotations, so that you can tell the checker things that you cannot tell the C compiler. (Even clang.) The ANNA and SPARK processors for Ada do the same. If you have any interest in developing reliable embedded software in any of your courses, get GNAT Ada free from AdaCore (or as part of some gcc distributions) and then get SPARK at the zero low academic price from Praxis. The last time I downloaded GNAT, SPARK came with it. Amazing kit; read up about the Tokeneer project. Several checkers for Java use Java annotations to make *checkable* claims that "this object reference must not be null", which the Java type system *could* have enforced but doesn't.

FindBugs (http://findbugs.sourceforge.net/) is a static checker for Java, very much in the style of lint. Don't bother using a debugger until you have run FindBugs and checked everything it complains about. You can drive it from the command line or using a GUI. I have never yet run FindBugs on a program for the first time without it finding real issues aplenty.

It works from .class files, so it can find bugs in programs or libraries that you do not have source code for.

There was also a very powerful static checker for Java called ESC/Java 2, which you'll find on the web. Anyone seriously interested in developing Java really should look at its current JML-based successor. The Daikon program from MIT can be used to find candidate invariants. (Other tools from MIT's Program Analysis Group are also very interesting.)

Another Java checking facility that should not be overlooked is the extensible type annotation system described in JSR 308 "Type Annotations". That feature is now in Java 7. There is an extensible toolkit that use it for checking. One of the checks that's supported (well, it was obvious on the first day that Java was released that it was really stupid not to have this in the type system, and now, like ECMA Eiffel, it *is*) is nullity annotations on pointer (reference) types. By default, parameters are assumed to be non-null, but you can override that with a `@Nullable` annotation, and local variables are assumed to be nullable, but you can override that with a `@Nonnull` annotation, and the checker that you get with the free compiler checks this thoroughly. No more unexpected null pointer exceptions!

There are checkers of varying power for other languages.

The 'clang' compiler that comes with current versions of Xcode has static analysis for C, C++, and Objective-C.

For Ada there's 'gnatcheck', which can enforce various rules. 'Forcheck' is a commercial program for Fortran. It's possible to build your own. For example, my Smalltalk compiler lets you state rules like

1. "A class with #+ and #negated must also have #-"
2. "A class with #< must also have #>"

3. "A class that defines #= must also define #hash"

This found errors.

This is one of the areas where the lecturer's own experience is important. I can tell the students truthfully that the company I used to work at (Quintus) made a compiler and libraries for Prolog, and whenever we developed a new checking tool, we found errors in existing code that we thought worked. I can tell them about my checker for Smalltalk (indirectly) discovering a bug in the ANSI Smalltalk standard. (It found that a class was missing a certain method, and behold, there was no such method in the standard. But there should have been.) I can tell them about running lint on a large program written by a full professor of Software Engineering, and finding several real bugs. I sometimes give them a small handout showing what I found when I ran the 'glint' script above and a real 'lint' on the EMBOSS software (emboss.sourceforge.net).

Slide 6 shows an excerpt from a simple backpropagation neural net program written by one of our senior people, and given to me as a challenge: "can you make this go faster". (Yes.) In this case I used splint as a tool to help *me* understand the program. This slide shows a forward declaration for a function annotated with which variables it is allowed to read (@globals) and which it is allowed to write (@modifies). Having these annotations meant that when I was looking at the body of a function, I could look at these declarations and see at a glance what side effects there might be and when it was safe to re-order two function calls. Basically, I told splint to check these annotations, and kept on adding annotations until it stopped complaining. Here, for example, is a function which is declared to have no arguments and no results, but in fact it depends on 14 parameters (that should probably be packaged up as one or more records) and has six outputs besides I/O. The basic C declaration is *very* seriously misleading. This file should be made available to you.

The title of the slide is a reminder that the C compiler cannot check whether a function uses variables it is not supposed to or changes variables it is not supposed to, because it doesn't *know* what the function is supposed to do and cannot be told. (The programming language Euclid fixed this.)

The "What's wrong with these loops" slide shows technically legal C loop headers that are almost certainly buggy. I had a 4th year student write a checker for this. It's one of the few common things that splint doesn't watch out for, and adding this to splint would be a very nice and very useful project. Many things are legal C that are not sensible. This is even true in Java, albeit to a lesser extent. For example,

```
int a[] = new int[4]{1, 2, 3, 4}; a[-1]++;
```

is syntactically legal Java, but is certain to go wrong at run time.

I am particularly pleased that FindBugs notices

```
int x = ...;
int y = ...;
long product = (long)(x*y);
```

where the intention was almost certainly to get an accurate product, which can be done by

```
long product = (long)x * y;
```

but not by computing the wrong answer and widening it. Similarly, if you want to divide two integers,

```
int x = ...;
int y = ...;
double ratio = (double)(x/y);
```

first does a truncating integer division and then widens the wrong answer to double. Working code uses

```
double ratio = (double)x / y;
```

The "Example non-code verification" slide relates to my personal experience. One year the examination paper for this subject was proof-read by three different people on seven separate occasions, and still had easily spotted mistakes. That's when I realised that Microsoft Word wasn't just the wrong tool for the job, it was a terminally *stupid* thing to use a word processor for things like exams. I am rather fond of SGML. (And not fond at all of XML.) The point here is "here is something that is not code; what can we automatically check?"

The next slide lists 6 important things that can be checked by an SGML-aware tool but not with a word processor. (More precisely, not without writing a great deal of Visual Basic for Applications and some highly unnatural invisible markup by the typist. It's also, by my measurements, much easier to type this markup than it is to type into a word processor. And of course, Microsoft killed VBA for Mac anyway.)

Using programs to do static verification as much as you can is a good idea, but there are some things where programs are not much help. The advice here is to automate what you can, and then have people look at stuff for the rest. People are not good at small things, precisely because they are good at making sense of what they read. They can tell when the comments and the code disagree, quite often. This leads to Program Inspections.

My take on this is that any kind of inspection is better than no inspection, and that light-weight semi-formal inspections that are actually carried out are better than heavyweight formal inspections thare are so costly they are never done after an initial burst of enthusiasm. I provide the NASA books about formal inspection as reference material.

I don't talk about Pair Programming in these lectures. One of the advantages of Pair Programming is supposed to be "continuous inspection". In practice, while one of a pair has the keyboard, the other is often staring out the window or counting spots on the ceiling or daydreaming about grabbing the keyboard back. Unless you have a large screen set to large print, or unless the pair like each other enough to sit closer together than Siamese twins, it's physically hard for two people to see the same screen at the same time, especially on a laptop. If you want to use pair programming, I strongly recommend using VNC to let each person stare at their own screen, or using something like SubEthaEdit that lets two (or more) people at different machines edit the same file at the same time. However, even then, the "continuous inspection" of Pair Programming is very focussed and fairly limited. You need to inspect the code that *hasn't* been kept up to date, and you need to keep the big picture in mind as well as the details.

The British Medical Journal article I cited about human error is very very important. Humans make mistakes. Systems make them *worse*. A humane

and technically savvy organisation is a learning organisation which knows that *system*-level measures are needed to contain (limit the consequences of) human error. In such an organisation, inspections are an amazingly useful tool, teaching the people and the organisation a lot. In an organisation that wants to point the finger of blame at individuals and not accept any responsibility for structuring their work environment well, inspections can be a disastrous way of amplifying distrust.

I should add to the list of resources at the end that the Visual Works 7.5 Non-commercial release of Smalltalk has even better "lint" integration in its IDE: "code critic" tells you about violations of a number of style rules (which you can select from and add to), and is indeed surprisingly effective at finding bugs in what one thought was working code. The same checks are available for Squeak with a different interface.

The resources slide says "we have SPARK". So should you. According to http://www.praxis-his.com/sparkada/universities.asp, "the fully supported professional SPARK toolset is available free-of-charge to university faculty members for teaching and/or research."

I should say more about how static checkers like splint work.

Syntax checks are one obvious kind of static checking.

It's easy to overlook syntax checks because they're "old technology" and a solved problem. We have tools like lex and yacc to make it easy to write parsers. Just because something is (now) straightforward, that doesn't make it not worth doing! If you are using XML configuration files, for example, and I hope you aren't, you should use a validating XML parser or schema checking parser to check them *before* your program has to use them. It's also worth adding your own checks that are stricter than the standard requires. For example, `0[a]` is perfectly legal C and always has been; it is no less legal or standard than `a[0]` and means the same thing. Yet some C compilers have been known to get it wrong, so you might want to ensure that your code doesn't do that.

Many other static checks are based on some kind of *data flow analysis*: what kinds of values can reach this point? Where can this value get to?

type checking can a value of the wrong type reach this point?

uninitialised variables can an undefined value reach here?

null pointer can a null pointer reach here?

interval analysis what's a usefully small range L..U such that any value that reaches here must be in that range? Good for optimising array bounds checks away. Also good for pointing out places where fixed size integers might overflow. Also good for pointing out where divide by zero might happen.

type state checking could we find ourselves trying to apply an operation to an object that is in a state where that operation is not allowed? (Writing to a stream in a reading state, or a closed state. Sending a message to a socket opened using `listen` rather than `connect` or `accept`.

escape analysis can a reference to an object escape out of the place where it was created to a place where something it refers to does not exist? The classic C example is

```
char *itoa(int i) {
    char buffer[32];
    return sprintf(buffer, "%d", i);
}
```

When the function returns, the array its result points to does not exist any longer. If a Java object cannot escape the thread that created it, it certainly doesn't need any locking. If it can, maybe it does and maybe it doesn't.

While these are the commonest examples, you will find others. For example, a few years ago,

Y2K checking could this variable ever hold a date with a two-digit year number?

was important. If you normally use some other calendar, you may think the problem doesn't affect you, but UNIX has historically used a 32-bit integer count of seconds for timestamps, which gives you only 136 years worth of seconds. The count will turn negative some time in 2038, just 24 years away, so some dates in 2038 will be reported as 1902. If they are reported in your calendar, they will still be wrong by 136 years. Of course, POSIX *has* addressed the timestamp issue. In the Single Unix Specification, 4th edition, the timestamp fields for a file are of type `timestruc_t` instead of `time_t`. Sadly, what they extended was the precision (there's a seconds field and a nanoseconds field), not the range. 64-bit timestamps, and *still* a Y2038 bug... So we *still* need a "could this variable hold a limited-range timestamp" checker.