ENGR101 T1 2024

Software quality, testing, debugging

School of Engineering and Computer Science
Victoria University of Wellington

Software Quality

How do we characterize *'quality'* software?

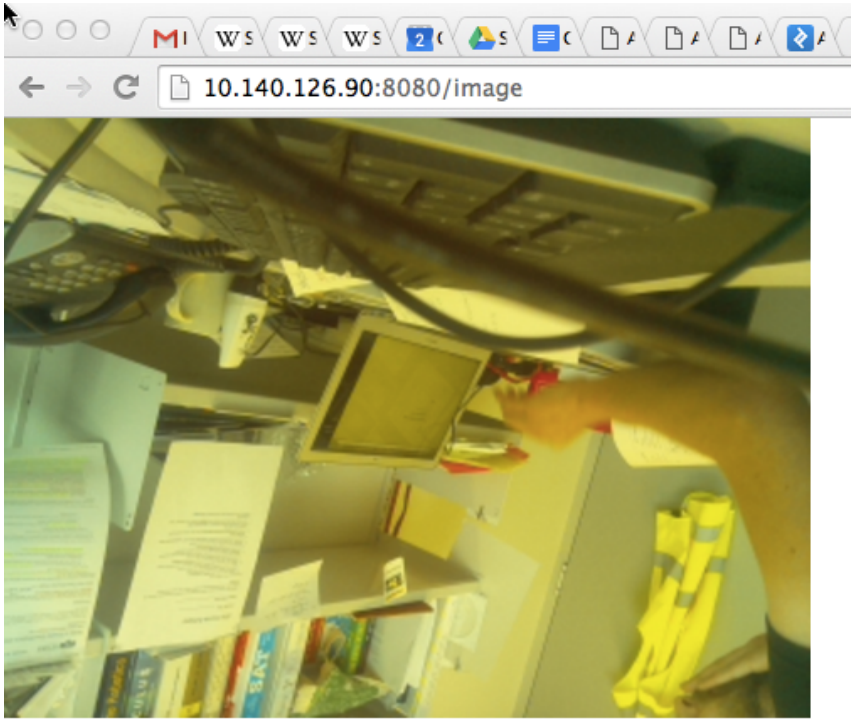
Simple definition seems to be:

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component, or process meets customer or user needs or expectations.

But there is more to it...

At the end of the day it comes down to code

Functionality is only PART of what makes quality software. . .



Also:

- Correctness
- Efficiency
- Fault Tolerant
- Maintainable
- Secure
- User-friendly

Correctness

An algorithm or program is correct if it is free from error.
There are several types of errors:

1. Compile-time errors - easiest. Compiler (if there is one) tells you where error is.
2. Run-time error - can only be detected when program is running. **segmentation fault, division by zero** are typical examples
3. Logic errors - program compiles and runs without obvious problem. But result is wrong.
   There are several ways to detect logical errors. Testing is simplest but not only one.

Efficiency

- Time behavior: Characterizes response times for a given thruput, i.e. transaction rate.
- Resource behavior: Characterizes resources used, i.e. memory, cpu, disk and network usage.

Keep in mind that modern compilers work miracles in code optimization.
Things to watch will be:

- If you are running **for** inside **for** - try to calculate as little as possible inside inner one
- Do not pass big arrays by value, copying is very slow. Use passing argument by reference.

AVC: try to avoid passing whole image as an argument to the function.

Fault-tolerant

Murphy's law: If something can go wrong, it will.

Murphy's Law was born at Edwards Air Force Base in 1949 at North Base. It was named after Capt. Edward A. Murphy, an engineer working on Air Force Project MX981, (a project) designed to see how much sudden deceleration a person can stand in a crash.

Smith's law: Murphy was an optimist.

Fault-tolerant

Your input files will be corrupted, network will not respond, variables will have un-reasonable values when they should not(example later).
Deal with it.
Trust nothing.
AVC: What program does when robot does not see black pixels any more?

Fault-tolerant

You don't want your program to crash if something is wrong. Recovery blocks - both in Java and C++

Listing 1: try..catch

```
#include <iostream>
int div1(int a, int b){
      if( b == 0 ) {
           throw "Attempted to divide by zero!";
       }
     return a/b;
}

int main(){
   int c;
   try{
      c = div1(3000 , 0);
    } catch(  const char* message ) {
           std::cout<<"Exception caught "<<message<<std::endl;
      }
    std::cout<<" c="<<c<<std::endl;
}
```

try_catch0.cpp

Maintainable

Your program WILL have to be modified. PERIOD. By somebody else, may be many years from now..
How to write Maintainable software:

- Write short units of code.
- Write simple units of code.
- Write code once - Duplication of source code should be avoided at all times, since changes will need to be made in each copy.
- Keep your codebase small. A large system is difficult to maintain.
- Write clean code: variable names, COMMENTS, indents, etc.

Secure

You have whole course on that...

User-friendly

- Simple
- Clean
- Intuitive
- Reliable

- User interface is like a joke. If you have to explain it, it is not that good.
- Linux is user-friendly. Just particular who the friends are.

Which users?

`https://www.youtube.com/watch?v=Ze3hthGRbRo`

Software failures
Toyota

used Drive-by-wire system.
Car accelerates to maximum speed.
"...has been able to demonstrate how a single bit flip can cause the driver to lose control of the engine speed in real cars due to this software malfunction that is not reliably detected by any fail-safe backup system."
`http://www.sddt.com/`

Software failures

The **Patriot missile** battery at Dhahran had been in operation for 100 hours, by which time the system's internal clock had drifted by one-third of a second. Due to the missile's speed this was equivalent to a miss distance of 600 meters.
The radar system had successfully detected the Scud and predicted where to look for it next. However, the timestamps of the two radar pulses being compared were converted to floating point differently: one correctly, the other introducing an error proportionate to the operation time so far (100 hours) caused by the truncation in a 24-bit fixed-point register.

Software failures

Y2K $400 billion in fixing.

many programs represented four-digit years with only the final two digits — making the year 2000 indistinguishable from 1900.

2038 problem

The latest time that can be represented in Unix's signed 32-bit integer time format is 03:14:07 UTC on Tuesday, 19 January 2038 (231-1 = 2,147,483,647 seconds after 1 January 1970).Times beyond that will wrap around and be stored internally as a negative number, which these systems will interpret as having occurred on 13 December 1901 rather than 19 January 2038. This is caused by integer overflow.

Software failures
X-ray machine: Therac25

It was involved in at least six accidents between 1985 and 1987, in which patients were given massive overdoses of radiation. Previous models had hardware interlocks to prevent such faults, but the Therac25 had removed them, depending instead on software checks for safety.

Software failures
Mars orbiter 1998

Software that calculated the total impulse produced by thruster firings produced results in pound-force seconds. The trajectory calculation software then used these results - expected to be in newton seconds - to update the predicted position of the spacecraft.

Software failures
Eve Online's game

deployment of the Trinity patch erased the boot.ini file from several thousand users' computers, rendering them unable to boot. This was due to the usage of a legacy system within the game that was also named boot.ini. As such, the deletion had targeted the wrong directory instead of the /eve directory

Software failures

According to the FAA, there's a software bug in the 787 Dreamliner that can cause its electrical system to fail and, as a result, lead to "loss of control" of the plane. $248 days == 2^{31}$ 100ths of a second.

So what are we really trying to achieve?

Building software systems . . .
that do the right thing . . .
cost effectively . . .

What's the problem?

Reality!

Good, fast, cheap: Pick any two!

- You can have good and fast, but it wont be cheap.
- You can have good and cheap, but it wont be fast.
- You can have cheap and fast, but it wont be good.

This idea is sometimes called the *'project triangle'*.

Topic 2: Producing Quality Software

- Documentation
- Debugging
- Testing

Documentation

Trying things is fun. But...

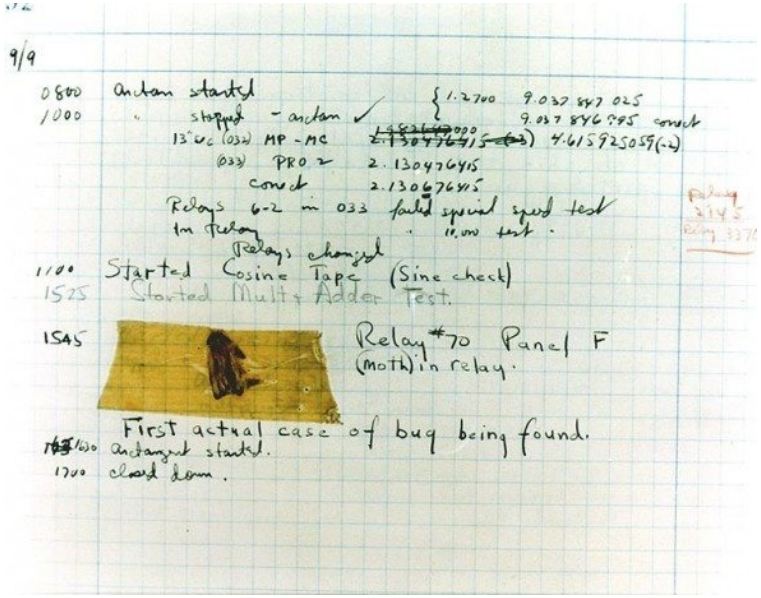
- Do not expect your tools to be user-friendly. They are for professionals and so you should be one.
- Trying things takes a loooong time. It is faster to read the manual.

If you're stuck and you haven't checked the documentation then: *you're doing it wrong!*

- Learn where your system/language documentation is and USE it.
- Google and Stackoverflow are your friends, but be VERY wary of blindly copying and pasting.
- Definitive sources are better (`man` pages, javadocs, etc).

Debugging

First computer bug



The term ’bug’ was first used by Grace Hopper on September 9th, 1945 when a real bug, a moth, short-circuited an early computer on relay number 70 Panel F, of the MARK II Aiken Relay Calculator, in the Harvard University. The operators of the computer said they had "debugged" the computer, and ever since then the terms has not changed.

Bugs - debugging

- Set **breakpoint**. Program stops when this line is reached.
- See values of the variables (**watch**)

Demo

Debugging

You can use tool called **gdb**.

`https://en.wikibooks.org/wiki/GCC_Debugging/gdb`

Debugging

Pragmatically...Best way to avoid bugs is to act logically

1. Beg/borrow/steal/buy a rubber duck
2. Tell the duck you will explain some code to it
3. Explain to the duck what your code is supposed to do, line by line.
4. At some point you will tell the duck what you are doing next and realise that is not actually what you are doing. The duck will sit there serenely, happy in the knowledge that it has helped you on your way.

`http://www.rubberduckdebugging.com/`

Testing

We know the specifications - what code should do.
OK, we read all the manuals and are confident that in tools used.
We designed the algorithm.
We wrote the code.
We got rid of some bugs.

Testing is to eliminate bugs we did not know about yet...

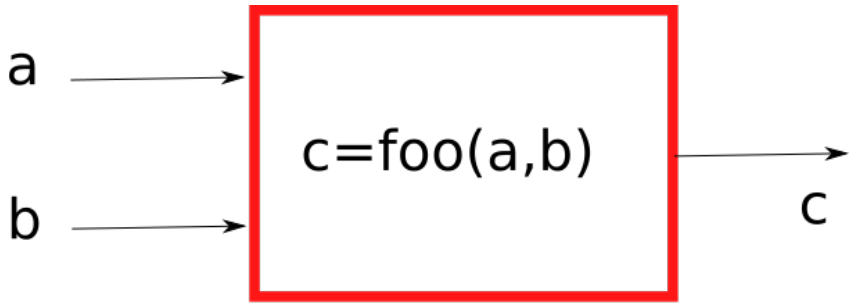
1. Define what your system SHOULD and SHOULD NOT do.
2. Creating your system.
3. Checking your system matches what you set out to achieve!
4. Repair the bugs and repeat.

Testing: Software Example Answers

Making a method to do division of two numbers.

- Does 1/1 = 1?
- Does 1/0 return infinity or an error?
- Does 2/-1 = -2?
- Does program reject inputs that aren't numbers?
- Does program round?
- etc

Testing: How much?



All possible values of a and b! Hm...

- To be 100 percent sure in our software we have to test it for all possible data
- Which seems to be impossible!

Testing: Can not give 100 percent guarantee

Testing can show the presence of bugs but not their absence
- Edsger W Dijkstra, (A physicist who became one of the founders of computer science).

Lubarsky's Law of Cybernetic Entomology:
There's always one more bug.

There are ways to prove that software works (or not) for all possible combinations of inputs. It is called **formal methods**. Using logic and done for life-critical software.

There are lots of different testing activities.

- **Unit Testing**: Testing your functions/methods as you write your code.
- **Regression testing**: maintaining a possibly large set of test cases that have to passed when ever you make a new release. Old test cases are run against the new version to make sure that all the old capabilities still work. Adding more code can break old one.
- **Integration testing**: testing if your software modules fit together.

Testing types

- Compatibility: Does it work with other software?
- Regression: Does it work after updates (or bug fixes)?
- Destructive: What happens when we try to break it?
- Performance: Does it run fast/responsively
- Usability: Can you use it?
- Security: Does it prevent (limit) intrusion?

Notice any similarities with the Software Quality measures?

Good news!

It's entirely possible to automate (parts of) testing using tools like *GTest*.
Testing is also one of the major software employment areas globally.
It's growing fast and is DESPERATE for people who can critically and thoroughly assess systems.

Summary

1. The quality of software is important.
2. Testing and Debugging are our best tools.
3. Finding bugs is HARD!