

# Implementation Issues



CMPT 276

Slides # 17

© Dr. B. Fraser

# Topics

- 1) Programming is complex; how can we combat this?
- 2) Can we find bugs by reading each other's code?
- 3) Do different coding style help?
- 4) Can software reuse solve our problems?

# Limiting Software Complexity

# Limiting Software Complexity

- Writing software involves..

**working out complex interactions**

(McConnel: Code Complete 2, 2004)

- Developer must reason about..

**single bits up through billions of bytes**

- Beyond human competency

- Humans cannot cope with these 10 orders of magnitude all at once.
- An Analogy:  
think about a scientist trying to work with subatomic particles and galaxies in one calculation.

# Limiting Software Complexity

- (McConnel 2004)

## Software's Primary Technical Imperative:

- **Managing complexity**
  - We must simplify the problems in order to be able to think about them.
- Use encapsulation to reduce cognitive load
  - A good design allows you to..  
**forget about the details and work at a higher level**
  - A bad design requires you to work at low and high levels simultaneously, across multiple modules.

# Complexity Example

- Compare the levels of abstraction in the following two competing interface designs to control SkyTrain:

A

```
int isSpeedReadingValid();  
long getSpeedSensorReading();  
void setBrakeBits(long brakeBitMask);  
void setMotorRPM(long rpm);
```

B

```
double getSpeedInMps();  
void emergencyStop();  
// May speed up or slow down  
void accelerateToNewSpeedInMps(double speedInMps);
```

**B screw up less (less complicated)**

# Code Reviews

# Code Reviews

- A code review is having..  
**developers look at source code to find bugs**
- Can be informal:
  - a walk-through by the author to show how code works.
- Can be formal:
  - Devs use.. **checklists of defect types** to pre-review code
  - Have meeting to review code line-by-line
  - Record all bugs found
  - Estimate total number of defects by counting #defects found by 0, 1, or 2 devs during pre-review



# Practical Code Review Tips

- During a code review look for
  - logic errors (logic backwards, missing else, ...)
  - poor error handling
  - poor security (buffer overrun)
  - poor readability/comments
  - common errors (== vs =, null ptr, memory leak)
  - requirements misunderstanding
- Can do a “code review” on design, test plans, test code, deployment scripts, ....
  - Not just for shippable code.

# Theory side of Code Reviews

- Code Review Effectiveness (Jones 1996, in McConnell 2004)
  - Informal code reviews catch.. ~25% of defects
  - Formal code reviews catch.. ~60% of defects
  - Unit testing catches.. ~30% of defects
- If multiple devs do a code review, they find ~20% overlapping bugs.  
Therefore... Each dev finds different bugs!
- Best to give devs a checklist of things to look for (formal)

# Style Guide

# Coding Style

- Coding is hard!
  - Developers must actively think about:
    - **Architecture** (design patterns, classes)
    - **Logic** (algorithms)
    - **Low level** (data types)
    - **Syntactic issues** (spaces, naming, brackets)
- Syntactic concerns are often "religious" issues
  - Devs feel passionate about tab size (2, 3, 4, 8)
  - Not usually possible to “convert” someone to a new style without a lot of effort.

# Code Style Example

- Linux kernel style guide:
  - Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of  $\pi$  to be 3.  
(some text omitted...)
  - Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.  
(some text omitted...)

# Style Guide

- A style guide.. **formalizes coding style decisions**
  - Consistent code style across project makes it faster to read and modify code.
  - Instead of syntactic disagreements, devs can think of.. **improving quality of code design and algorithms**
- Can address some common issues in a language:
  - `int x = 0;`  
`print(x?x++:++x);`
  - `int y = 100;`  
`if (y < 5 && y > 0 && y % 2 == 1) y--;`  
`y = 10;`  
`print(y);`

# Code Reuse

# Reuse cost

- Reusing well tested component can..  
**improve the quality of your system**
- But, it's not free
  - Must find and evaluate existing components.
  - Must spend time to integrate into new system.
- Reuse can cause errors
  - Some disasters caused by reusing software which had an unknown bug.
  - We tend not to test them well enough because..  
**we trust the reused components too much**



# Caution on reuse

- Therac-25: Canadian made radiation therapy machine. Failure... **killed people**
  - Reused buggy software that \*relied\* on hardware safeties, which were left out in the later version.
- Ariane 5 rocket: Initial test flight... **self-destructed**
  - Reused a module from Ariane 4 which converted a floating point number to a 16bit integer.
  - Ariane 4 rocket never encountered an error.
  - Exception handling was turned off for efficiency.
  - Both primary and backup computers encountered the error at the same time and shutdown.
- **Reuse of components can lead to overconfidence**

# Summary

- Primary technical imperative: manage complexity.
- Formal code reviews more effective at finding defects than informal ones or unit testing.
- Use a style guide to free developer from syntactic decisions.
  - Can instead focus on higher-level issues.
- Consider possible reuse of existing software.
  - Beware of over confidence.