Software Engineering CS3003

Lecture 7 Fault-proneness

Lecture schedule

Week	Lecture Topic	Lecturer	Week Commencing
1	Introducing the module and Software Engineering	Steve Counsell	20th Sept.
2	Software maintenance and Evolution	Steve Counsell	27th Sept.
3	Software metrics	Steve Counsell	4th Oct.
4	Software structure, refactoring and code smells	Steve Counsell	11th Oct.
5	Test-driven development	Giuseppe Destefanis	18th Oct.
6	Software complexity **Coursework released Tues 26th Oct.**	Steve Counsell	25th Oct.
7	**ASK week**	**N/A**	**1st Nov**
8	Software fault-proneness	Steve Counsell	8th Nov.
9	Clean code	Steve Counsell	15th Nov.
10	Human factors in software engineering	Giuseppe Destefanis	22th Nov.
11	SE techniques applied in action	Steve Counsell	29th Dec.
12	Guest Lecture (tba) **Coursework hand-in 6th December**	Guest Lecture	6th Dec.

Lab schedule

Week	Labs	Week Commencing
1	**No labs**	20th Sept.
2	Lab (Introduction)	27th Sept.
3	Lab	4th Oct.
4	Lab	11th Oct.
5	Lab	18th Oct.
6	**No lab**	25th Oct.
7	**ASK week**	**1st Nov.**
8	Lab	8th Nov.
9	Catch-up Lab	15th Nov.
10	Work on coursework (no Lab)	22nd Nov.
11	Work on coursework (no Lab)	29th Nov.
12	**No lab**	6th Dec.

A clarification on terminology

- Fault is the same as a bug
  - Fault = bug
- Either a bug of fault can therefore lead to a failure
- An error is a human mistake
- So:
  - “Error” leads to a “fault (or bug)” and this may or may not lead to a failure

Psy's horse-galloping dance move was immortalised in the Gangnam Style video

Why did Gangnam crash YouTube?

- Happened in 2014
- YouTube's counter used a 32-bit integer to represent data in computer architecture
  - This means the maximum possible views it could count was 2,147,483,647
- YouTube posted: "We never thought a video would be watched in numbers greater than a 32-bit integer... but that was before we met Psy."

Structure of this lecture

This lecture will cover the following:

- What are software faults?
  - Some preliminary things about faults
- Causes of faults
- Antipatterns
- Debugging
- Code reviews
- What affects the lifetime of a system

Where do faults occur in software?

- Faults can occur in any software artefact:
  - Many faults are domain-specific
- Code faults:
  - 30-85 faults per KLOC before testing
  - 0.3-0.5 faults per KLOC delivered to customers
  - Interface faults seem to be often reported
    - i.e., coupling between classes
    - See the CBO metric

Why should we search for code faults?

- Important to find faults quickly
  - ATM crashes for example
  - British Airways system problems
- Some faults more likely to result in failures
- Some failures more important than others
- Some faults introduce security vulnerabilities
- The accurate prediction of fault-prone code is important because:
  - it can help direct test effort
  - reduce costs
  - improve quality

Causes of faults

Causes of faults

- We can identify a number of cause for why faults are invested into code:
  - 1. Requirements were wrong
  - 2. A deviation from requirements
  - 3. Logical errors in coding
  - 4. Testing inadequacies

1. Requirements were wrong

- Requirements Definition
  - Usually considered the root cause of software faults
    - Incorrect requirement definitions
      - Simply stated, ‘wrong’ definitions (formulas, etc.)
    - Incomplete definitions
      - Unclear or implied requirements
    - Missing requirements definitions

2. A deviation from requirements

- Deliberate deviations from software requirements:
  - Developer reuses previous / similar work to save time
    - Ariane 5 disaster
  - Developer(s) may overtly omit functionality due to time / budget pressures
    - System testing will uncover these problems but it's too late then!
    - Developer inserting unapproved 'enhancements'
      - Too much refactoring is done leading to "over-engineered" code

3. Logical coding errors

- Arithmetic
  - Division by zero
- Control
  - Infinite loop
- Resource
  - Buffer overflow (also includes security violations)
- Interface
  - Incompatible subsystems
    - One system fails when it tries to talk to another system

4. Testing inadequacies

- Shortcomings of the testing process
- Likely the part of the development process cut short most frequently
- Incomplete test plans
  - Parts of application not tested or tested thoroughly
  - Boundary conditions…
  - Path testing, branch testing … (coverage measures)

Fault density metric

- Fault density = fault count/size of the release
- Suppose we have three classes in our system of sizes:
  - 100, 200 and 300 LOC
- Faults in those three classes were 3, 10 and 4, respectively
- Fault density = 17/600 = 0.028 faults per LOC

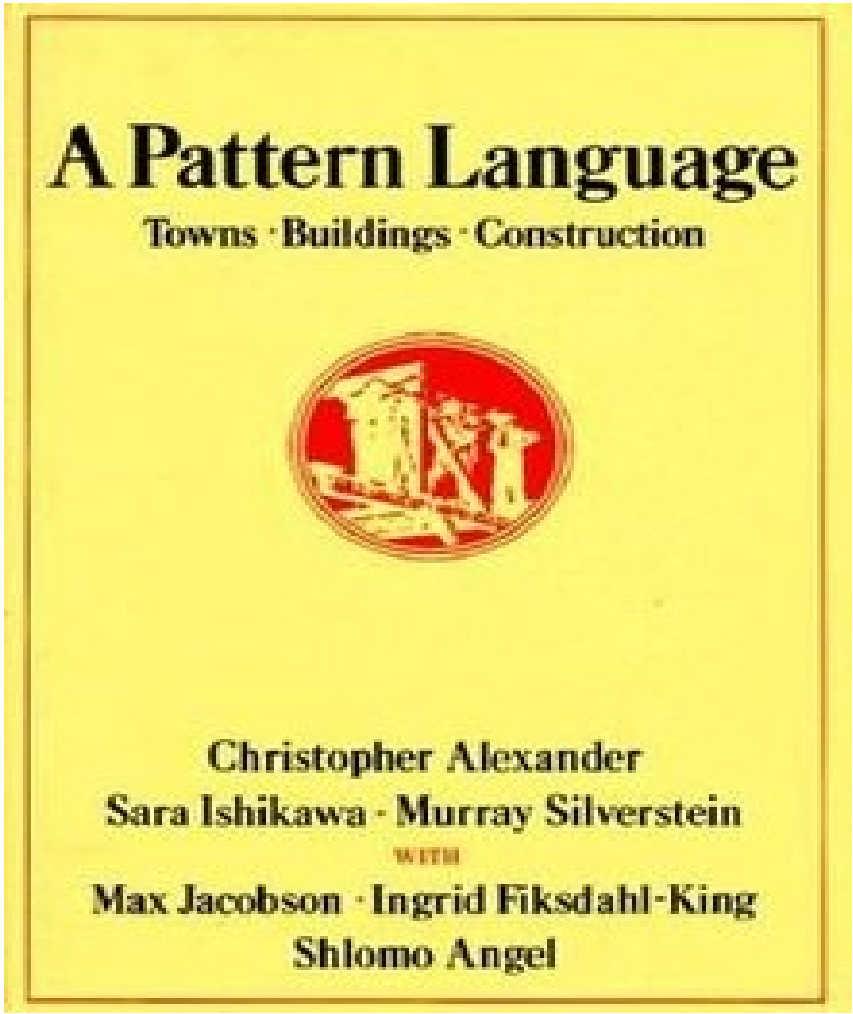
Does a low fault density imply quality code?

- Tester Skill: A highly skilled tester will be more likely to find more faults (and faults of higher quality) than a lower skilled tester
- Time Spent Testing: fault density doesn’t take into account the amount of time spent testing. It simply takes a snapshot of time and states “this is how many faults are in the software for this area/lines of code at this time”
- Fault Type: Something that fault density doesn't take in to account is different bug types: trivial, minor, major, critical etc.
  - If I have a product that has four minor faults, versus a product with two major faults, which product has the lower quality?

Antipatterns

The origin of patterns

https://en.wikipedia.org/wiki/WikiWikiWeb



Antipatterns

- Identifiable traits in the system that contribute to poor maintenance
- Similar to code smells, but are not always to do with the code…as we will see
- Opposite of patterns
  - Patterns help with doing nice things to a system, antipatterns do the opposite

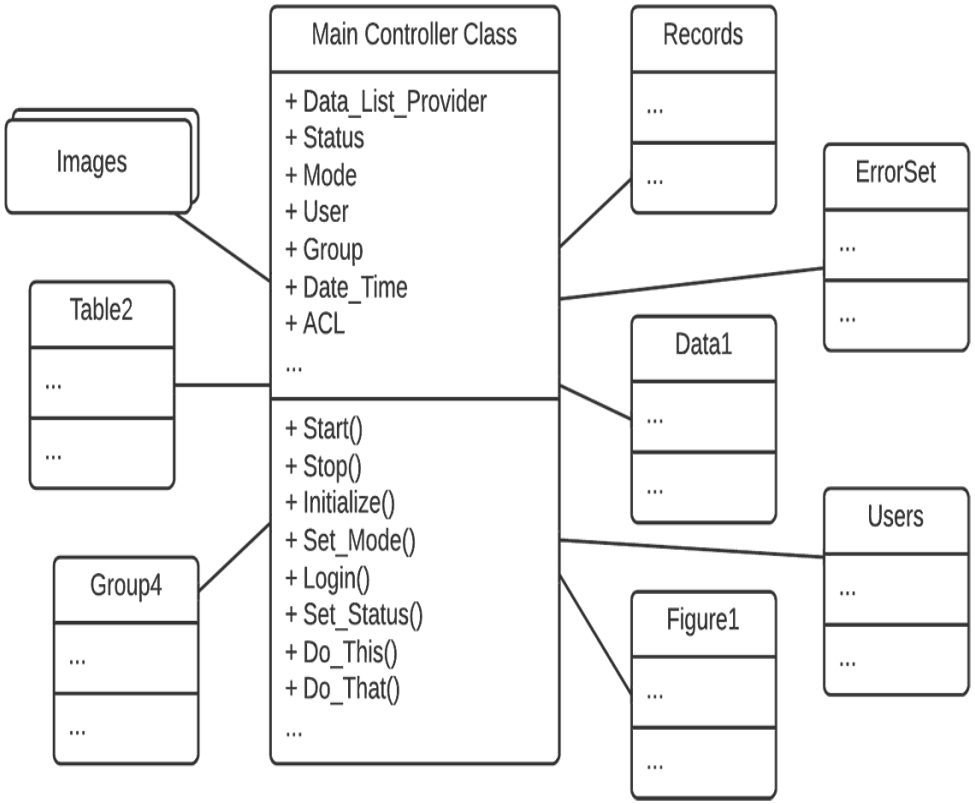
Antipatterns (cont.)

- Cut/copy and pasting
  - Leads to significant maintenance problems
- Poltergeists
  - "*A poltergeist is a type of ghost or spirit that is responsible for physical disturbances, such as loud noises and objects being moved or destroyed*." (Wikipedia)
  - Classes with limited responsibilities and roles exist in the system with a limited lifetime
    - They clutter software designs, creating system bloat

Antipatterns (cont.)

- Continuous obsolescence
  - Technology is changing so rapidly that developers have trouble keeping up with the current versions of software and finding combinations of product releases that work together
- Blob class
  - Found in designs where one class monopolizes the processing and too many responsibilities.

Blob class



Source: sourcemaking.com

Symptoms of a Blob

- Single Class
  - Large number of fields
  - Large number of methods
- Unrelated fields and methods
  - Overall lack of cohesiveness
  - Lots of coupling
- Too complex to reuse and test

Antipatterns (cont.)

- Frankencode
  - Refers to code parts that were never designed to work together,
    - Being pulled into a single application and held together with duct tape, baling wire and poor glue
- Broken windows
  - Small problems, left uncorrected, signal a lack of care about the state of things.
    - Discipline degrades and problems multiply.
    - Boy/Girl Scout Rule

Frankenstein

Source: The conversation.com

Antipatterns (cont.)

- Analysis paralysis
  - Aiming for perfection and completeness in the analysis phase leads to project gridlock
- A Big Ball of Mud
  - A "big ball of mud" is a system that lacks a perceivable architecture
  - Due to business pressures, developer turnover and constant change

Antipatterns (cont)

- Walking in a minefield
  - When software is released before it is ready
    - Users of the software are made to find all of its bugs and shortcomings,
    - The users feel as though they're *walking in a minefield*
  - It's important to release software as quickly as possible to minimize feedback loops, but
  - What is released should work so that users do not lose confidence
- If the system is constantly shifting such that things that work one day fail the next
  - Users will demand a different system altogether

Antipatterns (cont)

- The Duct Tape Coder
  - Someone who is able to cobble together software that solves the immediate problem
  - But without any concern for the code's quality or maintainability
  - Sometimes, a bit of duct tape is exactly what the situation calls for
  - But other times a lack of attention to detail and basic code hygiene constitute professional negligence

Debugging

(Some slide material due to Worcester Polytechnic institute, US.)

Debugging Principles (part 1 of 3)

- Fix one thing at a time
  - Don’t try to fix multiple problems at once
  - Change one thing at a time – test hypothesis
    - Change back if doesn’t fix problem.
- Question your assumptions – don’t even assume simple stuff works, or that “mature” products work
  - Example: libraries can have bugs

Debugging Principles (part 2 of 3)

- Check code recently changed – if bug appears, it may be in latest code
- Use debugger – breakpoints, memory watches, stack etc…
- Break complex calculations into steps – may be equation that is at fault or “cast” badly
- Check boundary conditions – classic “off by one” for loops, >=, <= etc
- Minimize randomness –
  - The more random the input, the more opportunities for things to go wrong

Debugging Principles (part 3 of 3)

- Take a break – too close to the problem so you can't see it
- Explain bug to someone else – helps retrace steps, and others provide alternate hypotheses
- Debug with partner – provides new techniques
  - Same advantage with code reviews, pair programming
- Get outside help –
  - Stack overflow posts etc

Rubber duck debugging

- Has become popular through the programming industry.
- Focuses on making the programmer carefully examine each line of their code and not just assume that it does what they expect, but explain how it works
- Programmer should make use of a rubber duck toy (or any other object to stand in as a person), and then
- Go through each line of their code and explain it to the duck as if it were a normal person with no programming
- Relies on proper communication and explanation

Wolf fence debugging

- Proposed by Gauss in 1982
  - Imagine there is only one wolf in Alaska. How would you find it?
  - The most effective way would be to fence Alaska in half and wait for the wolf to "howl.“
  - When you know which half has the wolf split it and again wait.
  - Keep repeating until you find the wolf
- A sort of “binary search”

Code Reviews

Why do code reviews?

- More than one person has seen every piece of code
  - If you know someone will be reviewing your code, you tend to raise quality threshold
- Forces code authors to justify their decisions
- Hands-on learning experience for novice coders without hurting code quality
  - Pairing them up with experienced developers
- Team members involved in different parts of the system
  - Improves overall understanding

Actual Studies

- Average fault detection rates
  - Unit testing: 25%
  - Function testing: 35%
  - Integration testing:45%
  - Design and code inspections: 55% and 60%.

- 11 programs developed by the same group of people
  - First 5 **without** reviews: average 4.5 faults per 100 lines of code
  - Remaining 6 **with** reviews: average 0.82 faults per 100 lines of code
  - Faults reduced by > 80 percent.

- After AT&T introduced reviews, study with > 200 people reported a 14 percent increase in productivity and a 90 percent decrease in faults

(From Steve McConnell's Code Complete)

Code review variations

- Inspection: A formalized code review with:
  - Roles (moderator, author, reviewer, scribe, etc.)
  - Several reviewers looking at the same piece of code
  - A specific checklist of kinds of bugs to look for
    - possibly focusing on bugs that have been seen previously
    - possibly focusing on high-risk areas such as security
    - specific expected outcomes (e.g. report, list of bugs)
- Walkthrough: informal discussion of code between author and a single reviewer
- Code reading: Reviewers look at code by themselves (possibly with no actual meeting)

Code reviews in industry

- Code reviews are a **very** common industry practice.
- Made easier by advanced tools that:
  - highlight changes (i.e., diff function)
  - allow traversing back into history
  - E.g.: Eclipse, SVN tools
- Some examples…….

Code Reviews at Google

- "All code that gets submitted needs to be reviewed by at least one other person, and either the code writer or the reviewer needs to have readability in that language. Most people use Mondrian to do code reviews, and obviously, we spend a good chunk of our time reviewing code."

  -- Amanda Camp, Software Engineer, Google

Code reviews at Facebook

- "At Facebook, we have an internally-developed web-based tool to aid the code review process. Once an engineer has prepared a change, she submits it to this tool, which will notify the person or people she has asked to review the change, along with others that may be interested in the change -- such as people who have worked on a function that got changed.

  At this point, the reviewers can make comments, ask questions, request changes, or accept the changes. If changes are requested, the submitter must submit a new version of the change to be reviewed. All versions submitted are retained, so reviewers can compare the change to the original, or just changes from the last version they reviewed. Once a change has been submitted, the engineer can merge her change into the main source tree for deployment to the site during the next weekly push, or earlier if the change warrants quicker release."

- Ryan McElroy, Software Engineer, Facebook

Single Responsibility Principle

- "There should never be more than one reason for a class to change” (Clean Code book of Martin)
- It's tempting to jam-pack a class with a lot of functionality
  - The issue with this is that your class won't be conceptually cohesive and it will give developers many reasons to change the class

How to avoid faults

- Avoid writing complex code
  - Stop smells (antipatterns) arising
- Use libraries where possible (reuse)
- Measure your code (use metrics)
- Refactor to remove complexity
  - Reduce coupling and improve cohesion
- Review your code
- Write “clean” code…….next week’s lecture

System lifetime factors

Factors that affect system lifetimes (from Somerville)

Factor	Rationale
Return on investment	If a fixed budget is available for systems engineering, spending this on new systems in some other area of the business may lead to a higher return on investment than replacing an existing system.
Risks of change	The danger with a new system is that things can go wrong in the hardware, software and operational processes. The potential costs of these problems for the business may be so high that they cannot take the risk of system replacement.
System dependencies	Other systems may depend on a system and making changes to these other systems to accommodate a replacement system may be impractical.

Factors that affect system lifetimes (cont.)

Factor	Rationale
Replacement cost	The cost of replacing a large system is very high. Replacing an existing system can only be justified if this leads to significant cost savings over the existing system.

Questions

- **Task 1:** Describe four differences between a pattern and an antipattern?
- **Task 2:** Describe any five antipatterns that you know of
- **Task 3: THREE** major causes of faults in code are often because: requirements were wrong, requirements were deviated from and finally, there were testing inadequacies. Explain your understanding of these three causes
- **Task 4:** What is the Single Responsibility Principle?

Reading for the week

- Interesting data on software faults that have occurred:
  - http://www5.in.tum.de/~huckle/bugse.html
- Sourcemaking.com (for antipatterns)
- https://deviq.com/antipatterns/ (for antipatterns)
- Really interesting paper on the evolution of code fault classification:

  Ploski J, Rohr M, Schwenkenberg P, Hasselbring W. Research issues in software fault categorisation. ACM SIGSOFT Software Engineering Notes, 2007, Vol 32, Num 6
- Carina Andersson Per Runeson A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems IEEE Transactions on Software Engineering, Volume 33 Issue 5, May 2007.
- Last two references can be retrieved from the online library catalogue

Reading for the week (cont.)

- Y2K bug: https://en.wikipedia.org/wiki/Year_2000_problem
- Gangnam style article: www.bbc.co.uk/news/world-asia-30288542
- Bugs in computer games:
  - https://en.wikipedia.org/wiki/List_of_software_bugs#Video_gaming
- https://stevemcconnell.com/articles/gauging-software-readiness-with-defect-tracking/