Source Code Analysis

Large Scale Programming, 1DL410, autumn 2009
Cons T Åhs
What is Great Code?

- The goal of this course is to write great and beautiful programs.

- We give a number of guidelines, but how can we really determine if our code is great and beautiful?
  - Asking the creator of the code might not always yield an unbiased answer.
  - Get another view of the code, i.e., ask someone else!

- Getting code reviewed is a great way to test the communicative skills of the program which is aimed at the other audience, i.e., not the compiler.
  - Code should be easy to understand and maintain.
  - You’ll spend most of your time as a developer reading and maintaining code that is not written by you.
  - There is research showing that seven (7) times more bugs are found during code reviews than during post development testing (by a separate QA team). Doing code reviews is thus well spent time.
  - Those doing the review might end up having to maintain the code later; it is thus in their own interest that they understand it.
  - Reviews also has the nice side effect of spreading knowledge about the code throughout the team.
Design & Code Reviews

› Your code is being reviewed each time you hand in a part of the assignment.

› You can increase the quality of your code by having reviews within your team before handing in an assignment.

  › You have probably divided the code between you, but you should all have good knowledge of the code.

› Knowing that code will be reviewed increases the quality from the start; observation affects that being observed (Heisenberg).

› Discussing and reviewing code and design will often lead to a better solution; your first idea might not be the best way to solve the problem.

  › alternatively, you might have a better idea than the one presented - don’t be afraid to speak up and offer a better solution!

› The cost of fixing a bug increases the later in the chain it is discovered, so finding bugs during code reviews is very cost effective. QA will never even see them.
Contents of a review

‣ Does the design follow sound principles?
‣ Are there any disadvantages with design?
‣ Does the design fulfil the requirements?
‣ Is it automatically testable?
‣ Do we have a reasonable package structure and package dependencies?
‣ Is the level of abstraction reasonable?
‣ Does the code follow the design?
  ‣ If not, are the deviations really improvements?
‣ Does the code adhere to code standards?
‣ Do the unit tests provide adequate coverage?
‣ Is the code understandable?
‣ Are the unit tests understandable and cover the requirements?
  ‣ Can you read the tests and get a clear understanding of what is being tested?
‣ Can the code be made better even if it does what it is supposed to do?
  ‣ This is called refactoring.
Automatic Code Analysis

- The benefits of various reviews should be clear, but we should really spend our time checking if the code standards and other simple rules are followed?
  - Since we like automation, we should ask ourselves if this can be automated?
  - Yes, it can!

- The most trivial task to automate is adhering to code standard - most, if not all, IDEs will let you adjust settings for code formatting, indentation and so on.
  - Do this and then you can mostly ignore this problem.

- There are also tools that can do a lot of automated analysis.
  - Compilers can be set to warn for questionable code such as variables that have not been initialised or an assignment instead of a comparison (a = b where you really meant a == b) etc.
  - lint was an early tool that did a lot of analysis that the C compiler did not do. This is now built into the compilers.
  - pmd and checkstyle are very competent freely available open source tools that have been installed on the department computers.
pmd

- Automatically scans your code for
  - bad style
  - common problems,
  - patterns that might indicate bugs.
- Highly configurable by indicating rulesets that you want to use.
- Try at least these rule sets:
  - basic - a set of good practices that you should follow.
  - code size rules - long parameter lists, complexity, number of fields and more.
  - braces - avoid using if, while etc without braces.
  - design rules - find questionable designs
  - unused code - find dead code
- It is also possible to write your own rules.
- Also contain cpd - a copy paste detector, i.e., it finds duplicated code (DRY violations). This is better than the corresponding part of checkstyle.
- Find more info at http://pmd.sourceforge.net/
  - also shows how to integrate with your IDE and ant
checkstyle

- Originally written to check code standards
  - extended for more checks in later versions in much the same style as pmd
  - design related rules, metrics (code complexity), naming etc
- Like pmd has a number of rule sets and can be extended by writing your own rules, should you find the need.
- Also find duplicate code, but pmd does it better.
  - has example config file using SUNs coding conventions - use it!
  - integrates nicely with ant and IDEs
Automated analysis

- You should try at least one of the mentioned tools and see what it says about your code.

- If it complains a lot, you should understand and fix the problems before handing in the assignment.
  - The same goes before doing a code review with your team members.

- Automated analysis is a good starting point for checking code, but it should be used as a complement to, not a replacement for, reviews performed by the team.
  - Code passing automated analysis can still be bad in many ways.
  - Code not passing automated analysis can still be good, but you should probably consider the complaints from the tools anyway.
Debugging

- What’s the first thing you should do when trying to fix a bug?
  - Add test case(s) to expose the bug. Why?
  - Apparently it was missing..
- Questions to ask (yourself and the team):
  - Why wasn’t the bug discovered earlier?
  - Why, why, why..?
  - Is this bug somewhere else in the code?
  - Does my change affect other parts of the code?
  - How can we prevent this from happening again?
- Find the root cause of the bug and fix it there.
  - Avoid the urge for a quick fix where it manifests - the real bug is often somewhere else entirely.
  - Quick fixes will often come back and haunt you later and lowers the overall quality of the code.
- When following the chain of events you might discover more broken stuff - fix them all!
- Use a rubber duck for debugging!