DEPARTMENT OF COMPUTER & INFORMATION SCIENCES

University of Strathclyde Science

Software Engineering

Lecture 6: Construction

Billy Wallace
w.wallace@strath.ac.uk

CS993

Lecture Outline

- Complexity
- Tools
- Test Driven Development (TDD)

Not just programming

- While we construct code, we will also be doing low-level design and testing.
- We create a large number of artefacts at this stage and we should have them under version control.

Complexity again

- We need to control complexity. We don't do this for the computer, we do it for people. So write code for people.
- Code is read many more times than it's written.
- You are also writing code for your future self.
- Simple *is* clever.
- Observe coding standards and make it easy on the entire team.

Comments

- Try to document code before you write it.
- Comment what the code *should* do.
- In Java, use JavaDoc to document your code.

Complex Flow of Control

- Object oriented code can be difficult to follow. It isn't always possible to see what method implementation will be called.
- Inheritance can cause this.
- Also polymorphism.
- Your IDE is your friend here.

On Being Pedantic

- We often need to be pedantic to save ourselves problems later.
- For example, decide if you want to use tabs or spaces for indentation and make sure that your IDE doesn't switch from one to the other.
- Likewise for CR/LF or LF or whatever Macs use.
- Automation and good configuration can help to prevent problems here.

Automatically Pedantic

- One way to automate our pedantry is to have metrics that will look for bad practices such as uninitialised variables.
- There are plug-ins for IDEs that will do this, e.g. SonarLint.
- Remember complexity. Having something look after the small things means that you're less likely to have bugs because of them.

Various

- Design by contract - can use assertions.
- APIs can be thought of at the same time as this.
- Defensive programming
- Reuse - Don’t Repeat Yourself (DRY).

Anticipate Change?

- I do this for simple things like formatting.
- It is tricky to predict the future.
- Agile/TDD says to do the minimum to pass the tests.
- It's expensive to develop code we might never use and even more expensive to maintain it after that.
- Refactoring when we learn of new requirements is a better approach.
- YAGNI

Config and I18n

- It's best to look at how you get configuration info into the system on day 1.
- Retrofitting internationalisation is a pain, so do it immediately if you think the app has any chance of being used overseas.

Tools

- A good IDE is essential.
- Software engineers are expensive, so it's surprising that so many companies will not pay for good tools for them.
- Tools like profilers can save hours of work tracking down performance problems.
- A good workstation is essential. If you have an engineer sitting for many cumulative hours waiting for builds, you're burning money.
- Two monitors is another no-brainer. Having the IDE on one screen and documentation/browser on the other makes it faster to get things done.

Programming Languages

- Choose the right tool for the job, and that includes programming languages.
- In larger projects we often practice polyglot programming, I.e. we use different languages for different parts of the system.
- e.g. Swift on iPhone, Java at back-end, Javascript for web stack, etc.

In the flow

- The best tool for development can be a good environment that's quiet enough to get work done.
- Open plan offices can be nightmarish, and are very common - they're cheaper than giving people individual offices.
- It can take around 30 minutes to get into the flow, but only a second to lose it.
- Noise cancelling headphones can help.

Processes that can help

- Code reviews are a great way to spot complex code and also to help junior stuff to be better at their job.
- Pair programming can also be useful. The person typing has a different viewpoint from the person observing.

TDD

- Write test.
- Run the test and check it fails.
- Write the code.
- Run tests.
- Refactor code.
- Repeat.

TDD

- This drives a lot of positive behaviours.
- It forces you to think about APIs, how to make things testable, how to make things simple, etc.

Spring

- We used Spring in our first lab, although we didn't have to understand anything about it.
- Spring is built around the concepts of Dependency Injection and Inversion of Control.
- This says that any dependencies are put in place by an external framework.
- We don't have constructors that create a bunch of objects.
- The framework knows how the pieces go together and maintains the flow of control.
- This goes well with TDD.
- However, it can make it even more difficult to understand flow of control. IDEs can have problems finding dependencies when they are in configuration files.

University of Strathclyde Science

Summary

- Complexity, complexity, complexity.
- Make code simple, clear and well-documented.
- Tools can help.
- The Holy Grail is to “get into the flow”.
- Test Driven Development.
- Spring.

University of Strathclyde Science

Moving on from here …

- In the practical this week, we'll look at TDD.
- We'll look at developing the DynamicRandomQueue using TDD.

University of Strathclyde Glasgow