Dirty Jobs

Rob Westgeest, Willem van den Ende, BCS SPA Conference 4 July 2012

Introduction

Rob & Willem like to help people “learn by doing”.

This session is all about working with legacy code – adding new features.

Initially we have a chance to inspect the existing code and review it together.

Then implement a new story using test-driven development.

Exercise 1 – run the code

Vehicle tracker and Vehicle Field are linked via RMI. Vehicle Tracker GUI displays tracker data on map.

Exercise 2 – critically examine the code

VehicleMessageDecoder

- Static int values are used for state names – why not use an enum?
- HandleMessage method is very long and mixes control with transformation
- Lots of duplicated code

VehicleTrackingService

- Variable names are poorly chosen (thread1 and thread2)
- Why is thread2 = new Thread(this) ? There should be more comments
- The run() method body is too long – very hard to comprehend what it is doing

Other groups’ observations

- Too many responsibilities in one class – the giveaway is the number of imports
- Vast switch statements
- Tracking Service creates network connections directly – no chance to mock out for testing
- Constructor calls start() – too much going on here
- “Dead” code that does nothing
- Class that implements runnable and has both a “start()” and “run()” method

Exercise 3

Story 1: As a tracker, I want the same colour as on the big screen (vehicle field) to be displayed when tracking vehicles.

NB the RGB colour value is returned as an Integer when you call the vehicle field to track a specific vehicle – method TrackableField.trackVehicle().

Findings

Using the approach advocated by Michael Feathers, try to identify suitable "seams" where the legacy code can be unpicked.

Minimally intrusive changes – e.g. add a parameter to a constructor in order to inject the tracking service, rather than using the global instance.

Alternatively, make a method in the class under test to get the dependent object, instead of calling getInstance() directly – this can then be overwritten in the test.

Exercise 4

New story: As a tracker, I want the tracked vehicle's position extrapolated whenever a vehicle stops sending information.

Exercise 5

What are now your main annoyances?

- Anything static
- Close coupling between objects and collaborators (e.g. instantiation from constructor)
- Over-active constructor of Vehicle Tracking Service
- HandleMessage is far too complex – needs to be refactored
- Not many seams available – not written for testability

Michael Feathers suggests two further approaches for certain difficult situations:

- Write a new method to replace a method that's hard to test – you might keep the old one for existing callers, at least for a while.
- Write new behaviour in a new class and manually plug it in. Over time, more of the behaviour will migrate to the new classes.

These approaches are sometimes called the "strangulation technique".

NB it's useful to write some end-to-end tests to find out which parts of the code are actually being used. Otherwise you could end up writing unit tests and refactoring parts of the program that are never executed. Check out the "Amazing Refactoring Challenge" on the C2 Wiki.

Conclusions

- Legacy code is code without tests
- Think about seams when it comes to testing software
- Test-first drives loose coupling
- Avoid all singletons if possible
- Sometimes, things have to get worse before they get better (adding code, for example)

This is a skill worth practising – the best developers are at least 3 times more productive at creating tests for legacy code.

To learn TDD, don’t start with legacy code.