

Get Agile

Program Outline

Prepared by:

Dave Halpin
Centre for Software Engineering

Tel: +353 1 700 5622
Fax: +353 1 700 5605
E-mail: dave@cse.dcu.ie

Contents

Background 1

Course Structure 1

Why can't I just read the books? 2

Object Discovery 3

Refactoring Case Study 3

Method 3

Class 5

Inheritance 6

Missing Piece 8

Package Discovery 9

Behaviour Bloat 9

Reuse 10

Practicalities 11

Bibliography 12

Background

The majority of working systems follow the Big Ball of Mud [\[Foote\]](#) architectural pattern; this is why developers have to work long hours and still end up with unhappy customers. Big ball's of mud happen because people don't keep code tidy, it's a case of looking after the penny's and the pounds will look after themselves.

Before you start tidying up you need to be able to recognize different types of messes, commonly called Bad Code Smells. When you identify a specific smell then you need to know how to remove it, this is where Refactoring comes in. Refactoring involves transforming a design without affecting its observable behaviour. Martin Fowler has documented 72 Refactorings and 22 Bad Code Smells in his seminal book: Refactoring – Improving the Design of Existing Code [\[Fowler\]](#).

You can only become good at Refactoring through continuous practice. Each course module is packed with exercises big and small that allow you to develop your refactoring ability. The exercises come in two flavours, simple ones that allow you to practice a specific refactoring, and more complicated ones where you may have several competing bad code smells and different refactoring choices to make.

Along the way we learn about design patterns [\[Gamma\]](#) as they act as goals for refactorings, and how to do Test Driven Development [\[Beck\]](#) to get us out of sticky situations where refactoring can't help.

What you have in effect is a Fitness Gym for the Programmer's Mind, a place where you can over time incrementally build up your agile design capability through continuous practice.

Course Structure

Immersion Training needs to be balanced correctly, too much and you just get swamped, too little and you don't have enough to start applying the stuff in your day to day work. With this in mind we have broken up our training into two passes.

The first pass tackles design problems that you hit against every minute of every day. We call this pass *Object Discovery* because its all about the stuff you need to know to be agile at the level of writing methods and classes.

The second pass is about what is traditionally called architecture, i.e. how do you separate out concerns into different subsystems? How do you best deal with

integrating with other people's code? How do you fine tune performance without compromising flexibility? This is called the Package discovery pass.

Both passes have the same delivery format, which is, several days immersion at the CSE, "*great code and even greater stakes!*", attendees on return to the work place are then allocated dedicated revision and case study solution development time. Finally a half day is spent reviewing case study work in the presence of a CSE consultant. The times break down as follows

- Immersion Training at the CSE(3 days Object Discovery, 2 days Package Discovery)
- Dedicated Revision and Case Study Work (2 days)
- Case Study Work (5 days at 2 hours a day)
- Case Study Review Session (3 hours)

Why can't I just read the books?

The books listed in the bibliography are excellent at covering agile development practices. However the Get Agile program provides the following benefits over a self study approach.

- *Complete Package*: No specific technique is focused on to the exclusion of others; you learn how Refactoring, Design Patterns, Test Driven Development, and Aspects can be used in combination to improve your designs.
- *Mentored Work*: Your work will be mentored and reviewed by agile experts.
- *Free Access to new Materials*: Once you attend a Get Agile Module you then have free access to the continuously growing set of materials associated with the module. Specifically you will be able to access the growing base of challenging case study exercises.
- *Learning together*: It's stressful being the only agile believer in a team of sceptics, its better if you have some company. Organisations can fast grow agile expertise by handpicking developers to Get Agile.

Object Discovery

20 hours

In the Object Discovery Module we focus on Agile basics, things like; how to disentangle a complex method, how to decide which features belong in which classes, indeed, which classes do I need in the first place. We do not look at concerns that span code packages, which is the subject of Package Discovery.

Introduction

30 minutes

Introduces the “Get Agile” approach. Puts the Object Discovery Module in context.

Refactoring Case Study

90 minutes

Provides an introduction to refactoring using a case study. Covers most of the code smells touched on in subsequent Object Discovery Modules.

Method

4.5 hours

Looks at what can go wrong inside a method and how to fix these things.

Tabs

30 minutes

If I see indented code covering more than two lines I get uncomfortable, surely it could do with being extracted out into its own method. If I see a method one line long, I might consider inlining it.

Refactorings Covered: Extract Method, Inline Method

Tangled up in ifs

90 minutes

There are some ways of writing If statements that are common practice but add needless complexity. It's important that you know how to simplify these complexities.

Refactorings Covered: Apply DeMorgan's Law, Consolidate Conditional Expression, Consolidate Duplicate Conditional Fragments, and Replace Nested Conditional with Guard Clauses.

Spaghetti Loops

60 minutes

Loops can get very confusing if you don't follow some simple rules. For example, a poor programming idiom is to use a control flag, where a break or return statement could do in its place. The control flag is often motivated by a fear of having more than exit point from a function, a hang up from the days of structured Programming

Refactorings Covered: Remove Control Flag, Separate Query From Modifier, Fuse Loops, and Introduce Loop.

Local Heros

60 minutes

Temporary Variables can be like Hero Programmers, that is they take on too much responsibility and get in the way of design improvements. The best thing you can do is bring these little tyrants down a peg or two.

Refactorings Covered: Split Temporary Variable, Replace Temp with Query, Remove Assignments to Parameters, and Replace Method with Method Object.

Confusing Creation

30 minutes

Tidying up code duplication in constructors is different from tidying up duplication in methods.

Refactoring: Extract Catch All constructor, Replace constructor with Creation Method.

Class

2.5 hours

An OO system should be a well regulated community of objects with clearly defined responsibilities and needs. The focus on these smells is encapsulation, that is moving data and behaviour to the appropriate class and then minimizing that classes interface so that it hides as much of its internals from prying eyes as is possible.

Feature Envy

45 minutes

A code fragment is more interested in features outside of its parent class than it is in its parent class's features.

Refactorings Covered: Extract Method and Move Method.

Message Chain

45 minutes

Objects should only talk to their close friends, they don't do faceBook!

Refactoring Covered: Hide Delegate

Primitive Obsession

60 minutes

Developers especially novice OO Developers tend to shy away from creating classes. Instead they allow data and behaviour which should be brought together inside one class to be diluted inside another class or dispersed across a system.

Refactorings Covered: Introduce Parameter Object, Extract Class

Inheritance

7 hours

This module looks at the motivations for using inheritance, and the different forms inheritance based solutions can take.

Switch Statements

120 minutes

Humans, fishes, and birds share traits common to all members of the animal kingdom. But they also have features that vary. Representing them in one class is probably going to end in tears.

Design Patterns Covered: State and Strategy

Refactoring Covered: Replace Conditional with Polymorphism

Null Checks

60 minutes

Wouldn't it be nice to get rid of all the "myObject!=null" statements that litter our applications!

Design Patterns Covered: Null Object

Refactoring Covered: Introduce Null Object

Duplicate Code

90 minutes

Implementation Inheritance is a mechanism for removing code duplication.

Design Patterns Covered: Template Method, Composite

Refactorings Covered: Refactoring to Template Method, Extract Superclass, Push Up Feature, and Pull Down Feature.

Inappropriate Intimacy

60 minutes

You have to be careful using inheritance that your code does not become dependent on implementation details of the superclass you are inheriting from. Also, how do you discover the subtype of an object without resorting to the dreaded instanceof operator?

Design Pattern Covered: Double Dispatch

Refactoring Covered: Replace inheritance with Delegation

Implicit Language

90 minutes

An implicit language is normally evidenced by a series of methods which have conditional statements that differ in their structure but which reference the same set of fields.

Design Pattern Covered: Interpreter

Refactoring Covered: Replace Implicit Language with Interpreter.

Missing Piece

4 hours

How do you finish a jigsaw which has a missing piece? You can't - you first have to get down on your hands and knees and root around for the delinquent piece.

How can you refactor something that is missing an essential object field? There are plenty of design problems that you can't refactor away from. What do you? Do a complete rewrite? Well possibly, but there is an alternative; when you find yourself refactored into a dead end, and can't seem to steer the design towards where you feel it should be via refactoring, then you can try adding in some new tests, and getting them to run.

Test First Programming Case Study

90 minutes

Introduces Test First Programming(TFP) through a simple example case study.

Test List Writing

30 minutes

Learn how to describe a problem in the form of a collection of tests.

Test Writing Patterns

120 minutes

Learn how to drive out a strong design one test at a time.

Package Discovery

14 hours

You only get so much mileage out of the concept of a class. At some point you have to bite the bullet and split the code up into separate packages that address different concerns.

To get the complete picture we also need to address real world issues such as having to reuse other people's code, that just might not be quite as fantastic as yours. Also at some point we do run into hard nosed technical problems such as unacceptable performance, so we need to look at patterns that tackle these issues.

Finally there are some situations where you can't rely on refactoring alone to fix the design, you have to write brand new code, we look at how test first programming can address these situations.

Package

7 hours

This module looks at the motivations for splitting a package into two, and gives you refactorings and patterns that will make the process as painless as possible.

Behaviour Bloat

240 minutes

The same data can have many associated behaviours. These behaviours may be entirely different, e.g. pretty printing V syntax checking, or variations on a theme, e.g. a Swing UI V an eclipse SWT UI. The normal rule with objects is that you keep the behaviour with the data, but when the behaviour gets too varied or elaborate, you need to separate embellishments from fundamentals.

Patterns Covered: Controller, Observer, Chain of Responsibility, Mediator, Visitor, Mock Object

Cross Cutting Concern

180 minutes

How do you modularise code that seems to need to be everywhere in the application? E.g. log statements that appear at the start and end of every method. The answer is you can't with plain objects but you can with aspects. Patterns Covered – Decorator, Aspect-Oriented Programming

Reuse

2.5 hours

Most programmer's have this instinctive dislike of using other people's work. It's a deadly combination of lazyness, ("oh no not another API I have to learn") fear ("how do I know it even works"), and ego ("I'd love to give writing an ORM framework a shot, I'm sure I could do it in a weekend"). In the real world we have to reuse other people's code no matter how much it irks.

Incomplete Library class

30 minutes

Time: 60 minutes

What can you do when you discover feature envy but can't fix it because you are not able to change the class where the code rightfully belongs.

Refactorings: Introduce Foreign Method, Introduce Local Extension

Alternative Implementations

120 minutes

You don't want your code to be dependent on a specific third party code base, when alternatives exist.

Patterns: Adapter, Dependency Injection

Practicalities

4.5 hours

If you can't expect people to wait 20 minutes while your application loads into memory, then you just may need to use the Proxy pattern. There is a small coterie of patterns like Proxy that don't find their motivation in bad code smells, instead they owe their existence to pure technical necessity.

Performance

90 minutes

A standard way of tuning OO application performance is not to build objects until they are needed, and to reuse objects already in memory rather than to continuously create them from scratch.

Patterns: Proxy, Lazy Loading, and Flyweight

Distribution

120 minutes

There are standard ways of making distribution as unobtrusive as possible.

Patterns: Façade, Proxy, and Serialization

Singleton

60 minutes

How do you ensure only one instance of a class exists.

Bibliography

- [Beck] Test Driven Development by Example
Kent Beck
Addison Wesley, 2003
An insightful and entertaining introduction to TDD.
- [Beck,IP] Implementation Patterns
Kent Beck
Addison Wesley, 2007
Micro design practices that will keep you well clear of spaghetti loops, tangled ifs and the like.
- [Fowler] Refactoring - Improving the Design of Existing Code
Martin Fowler (contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts)
Addison Wesley, 1999
Documents mechanics of 72 refactorings, all Get Agile attendees get a free copy of this book.
- [Foote] www.laputan.org/mud/
The original Big Ball of Mud paper.
- [Gamma] Design Patterns – Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison Wesley, 1994
The Seminal book on Design Patterns.
- [Kerievsky] Refactoring to Patterns
Joshua Kerievsky
Addison Wesley, 2005
A sort of a sequel to Fowler's Refactoring book. Validates a reactive approach to design, i.e. patterns can be discovered from the code, and used as goals for refactoring.
- [Wake] Refactoring Workbook
William C. Wake
Addison Wesley, 2003
Excellent descriptions of Bad Code Smells, packed with examples and exercises.