Revision Notes for CO220 Software Engineering Design

Autumn 2017

1 Test-Driven Development

TDD Cycle

1. Write a failing test (API design): should be simple, have descriptive name and

test a single behaviour. 2. Code to pass test (internals design): should be as simple as possible. 3. Refactor (structural design): don’t forget to refactor!

Refactoring Process of improving the design of a piece of code, without changing its functionality.

• Should be applied little and often to continuously improve design.

• Only refactor in a green state. Tests ensure that behaviour is preserved.

• Should be automated to be done quickly and reliably.

• Small transformations are combined to achieve larger refactorings.

Example Transformations

1. Compose (extract) method: Break down method into chunks to make it shorter. Allows us to give a name for a concept and increase level of abstraction. Try to keep a constant level of abstraction. 2. Inline variable: Instead of using a temporary variable, inline its usages. Reduces

number of elements in method. 3. Extract to common class: First work to make duplication exactly the same.

Then refactor it, e.g. to another object. Reduces duplication.

Technical Debt When features are added quickly, in an inelegant way. If not fixed quickly, technical debt builds up.

JUnit and JMock

public class TestObjectTest {

// Set up mockery, constants, mock objects and tested object @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

final Order EXAMPLE\_PARAM = new ...;

CalledObject calledObject = context.mock(CalledObject.class);

TestObject testObject = new TestObject(calledObject, ...)

@Test public void doesSomethingSpecific() {

// Set up local variables for exceptions / return values SomeException exception = new SomeException();

// Set up expectations context.checking(new Expectations() {{

// Ignore: ignoring / allowing ignoring(unimportantMockObject); allowing(someMockObject).someMethod(with(any(ParamType.class)));

will(throwException(exception));

// Expect: exactly(n) / atLeast(n) / atMost(n) exactly(1).of(anotherMockObject).someOtherMethod(exception); exactly(1).of(anotherMockObject).anotherMethod(EXAMPLE\_PARAM);

will(returnValue(x));

// Don’t expect: never never(aDifferentMockObject).aDifferentMethod(); }});

// Set up triggers testObject.testedMethod(EXAMPLE\_PARAM, ...); testObject.anotherTestedMethod();

// Make assertions assertThat(testObject.getSomeValue(), is(x)); } }2 UML Diagrams

UML Class Diagrams

1

UML Object Diagrams

UML Sequence Diagrams

3 Object Oriented Design

Four Elements of Simple Design In order of importance:

1. Behaves correctly. 2. Minimises duplication. 3. Maximises clarity. 4. Has fewer elements.

Bad Design

• Rigidity: software is hard to change.

• Fragility: when we change one part, other parts break unexpectedly.

• Immobility: it is hard to reuse elements of the code in other applications.

Commands vs. Queries

• Commands: Ask another object to do something for us. Don’t care how it’s done, don’t expect return value. Changes state of invoked object.

• Queries: Ask another object to tell us a value (so we can do something with it). Should return value, but not have side effects on state of invoked object.

Tell Don’t Ask Objects send messages, requesting actions, but do not expect return values. Only queries return values.

Visibility

Coupling and Cohesion

• Coupling: How dependent two classes are towards each other. Reducing cou- pling reduces fragility and allows reuse.

• Cohesion: An object should have one basic responsibility. Increasing cohesion makes objects easier to reason about and reuse.

We can reduce coupling by using interfaces as roles.

4 Design Patterns

4.1 Behavioural Patterns

4.1.1 Null Object Pattern

Problem Checking for null is ugly.

Solution Use an interface and add a Null Object class that does nothing.

Law of Demeter Only talk to your immediate friends. Implementations that depend on pieces of the system further away result in tight coupling. Avoid train wrecks: getX().getY().getZ().doSomething().

• Encapsulation: ensure object’s behaviour is only affected through its API. Im- plementation and state of objects should be encapsulated. Reduces fragility.

• Information hiding: conceal how an object implements functionality. Increases abstraction, chunking up program into concepts.

Key idea: Make things private unless they should exposed as an API for other objects (and generally avoid protected).

2

Example

4.1.2 Template Method Pattern

Problem Requirements change over time, sometimes we need to adapt a small part of an algorithm.

Solution Extend via inheritance.

Refactorings

1. Extract method for smallest differing block of code. 2. Extract superclass and make differing method abstract.

Don’t forget to remove duplication in tests (e.g. using lambda to create simple instance of superclass).

Example

Benefits

• Follows Hollywood Principle (don’t call us, we’ll call you): Concrete classes only define methods that are called when the superclass needs them (they don’t call up).

• Follows Open-Closed Principle: You should be able to extend a class’s be- haviour without modifying it.

Benefits Does the same as the template method but with looser coupling.

4.1.4 Observer Pattern

Problem E.g. do X when someone presses button Y . (Interactive applications).

Solution Subscribe to changes of state.

3

– Change behaviour by adding new code, rather than changing existing code. – Separate things that change from things that stay the same.

Drawbacks Immobility caused by coupling.

Solution Extend via delegation.

Refactorings

1. Extract method for smallest differing block of code. 2. Extract delegate for the method. 3. Extract interface from the delegate class. 4. Define constructor, pass in the delegate class as argument.

Don’t forget to remove duplication in tests (use a mock object).

Example

4.1.3 Strategy Pattern

Problem Requirements change over time, sometimes we need to adapt a small part of an algorithm.

Example

4.1.5 Command Pattern

Problem Want to queue / log executed commands.

Solution Wrap up a piece of behaviour to do now or later, in an object.

Example

4.2 Creational Patterns

4.2.1 Factory Pattern

Problems

• Constructors are not clear.

• Required type may not be known until runtime.

Solution Name constructors.

Refactorings

1. Replace constructor with factory.

4

Examples

4.2.2 Builder Pattern

Problem May be many constructor parameters.

Solution Collect object’s configuration parameters.

Refactorings

1. Replace constructor with builder. 2. Rename create() to build(). 3. Replace (builder) constructor with factory. 4. Statically import factory.

Example

4.2.3 Singleton Pattern

Problem Require that you only have one of a type of object.

Solution Keep a single instance and make the constructor private.

Example

Drawbacks Accessing object requires tight coupling.

4.3 Structural Patterns

4.3.1 Adapter Pattern

Problem Have an X but need a Y . E.g. translate a message from a common format used in a message bus.

4.3.4 Proxy Pattern

Problem I have an X but it’s too slow.

5

Example

Problem Have an X but want a better X.

4.3.2 Decorator Pattern

Example

4.3.3 Facade Pattern

Problem Have an X but want a simpler X.

Example

Solution Control access to a surrogate object.

Points to Note

• You may need an adaptor to get an external service to implement the same interface as your proxy.

• The adapator should have a few basic tests.

Example

Extensions Caching can reduce latency of subsequent calls.

4.4 Architectural Styles

4.4.1 Model-View-Controller

Problem Same data but different views.

Solution Separation of concerns in interactive apps.

Example

4.4.2 Presentation-Abstraction-Control

Problem GUIs with hierarchical structure.

4.4.4 Map-Reduce Pattern

Solution Divide computation into map and reduce phases to allow easy parallelisa- tion.

Solution Define a tree of MVC agents that communicate up and down.

Example

Solution Keep a queue of commands.

Example

Problem Large scale data processing.

6

Alternative Use an event bus to allow communication between all agents.

4.4.3 Publish-Subscribe Pattern

Problem Producers and consumers / balancing load.

Example

4.4.5 Ports and Adapters / Hexagonal Architecture

• Separate core application logic from services which the application depends on.

• Access other services only through adapters.

• Unit test individual objects, mocking external services at the adapter level.

• Integration test the adapters.

• System tests run a small set of end-to-end test scenarios.

5 Metrics

Coupling

• Afferent coupling: How many other classes use (arrive at) this class (measures responsibility).

• Efferent coupling: How many classes are used by (exit) this class (measures independence).

• Instability: Ce/(Ce + Ca). Core parts should be stable. Parts at the edges, e.g. UI, don’t need to be.

Cyclomatic Complexity

• Counts nodes and edges in the control flow of a program (number of possible different executions).

ABC Metrics Counts occurrences of:

6 Legacy Systems

Legacy system: software you have inherited and that is of value to you.

Key Ideas

• Preserve existing behaviour: keep stuff that works and don’t change too much.

• Test harness: introduce automated tests around any changes you make.

– Seams: place where you can alter behaviour without editing it in that

place. – Every seam has an enabling point: place where you can make the decision

to use one behaviour or another. – Sensing: verify correct calls are made when we can’t access values our

code computes.

• Gives lower bound on number of tests required.

WILT Whitespace Integrated over Lines of Text is strongly correlated with cyclo- matic complexity.

• Assignments

• Branches

• Conditions

Flog is an ABC metric tailored to Ruby. Flay identifies duplication.

Lifeline Graph complexity over time.

Turbulence Number of commits made to each file, compared against complexity of code.

• Understand system structure: dependency graphs / dependency structure matrices.

7

7 Concurrency

Key Ideas Try to separate code that manages threads and concurrent execution.

• Implementing Runnable instead of extending Thread reduces coupling.

• Runnables can be run synchronously or asynchronously.

• Callables can also return values and throw exceptions.

• Executors can maintain a queue of commands. Queues can act as load bal- ancers.

• ExecutorServices allow us wait for all tasks to complete (or use a latch).

– ExecutorService executor = executors.newFixedThreadPool(n). – Future future = executor.submit(callable) (use a lambda to keep

concise).

• Futures allow us to wait for a result (future.get()).

– Need to catch InterruptedException | ExecutionException. In

most cases, just throw a new RuntimeException.

• If no return value is expected, we can use executor.shutdown() and executor.awaitTermination(n, timeUnit).

8 Interactive Applications

Graphical User Interfaces (Views)

• Use a JFrame (rather than extending it).

• Define a JPanel, to which we add JButtons, JTextFields, ....

• Add the panel to the frame using frame.getContentPane().add(panel).

• Make the window appear using:

– frame.setSize(x, y). – frame.setVisible(true). – frame.setDefaultCloseOperation(JFrame.EXIT ON CLOSE).

Controller Use lambdas to implement ActionListeners that pass input to a model. E.g.

// Controller private void addNumberButton(JPanel panel, int num) {

JButton button = new JButton(Integer.toString(num)); button.addActionListener(actionEvent -> model.input(num)); panel.add(button); }Observers Your view should add itself as an (Updatable) observer of the model. E.g.

// View private void display() {

calculatorEngine.addObserver(this); ... setup display and make visible ... }@Override public void updateWith(int value) {

outputTextField.setText(Integer.toString(value)); }// Model public void addObserver(Updatable view) {

observers.add(view); }private void notifyObservers(int value) {

observers.forEach(observer -> observer.updateWith(value)); }9 Web Applications

Key Ideas

• Serve data rather than pages: providing an API. We can then:

– Process and render (e.g. using AJAX) the data (to many clients). – Use it in another server side application.

• Model often involves business logic, data from DBs, etc.

• Controller has to handle Routing / parameters / GET and POST requests.

8

• Provide views (templates) for different clients.

• Cloud hosting often preferred to minimise responsibility/downtime and adapt quickly to demand.

10 Distribution and Remoting

Key Ideas

• Use HTTP (methods, status codes) to make requests.

• REST (Representational State Transfer): resources identified by URIs and have a representation (e.g. using XML or JSON).

• Hide away mechanics of how services are accessed.

• Richardson maturity model:

– Level 0: Don’t use URIs or different methods. Usually use a single URI. – Level 1: Use URIs to represent resources, but don’t HTTP use methods. – Level 2: Use URIs and different methods, and return appropriate status

codes. – Level 3: Fully RESTful. Representations contain hyperlinks to other re- sources. E.g. do a search for records rather than looking up a given record.

11 Continuous Delivery

Agile Methods E.g. Extreme Programming, Scrum, Kanban. Favour an iterative approach with small development cycles.

Continuous Integration Developers should merge in work often to avoid integra- tion problems. Run automated tests to keep master healthy.

9