

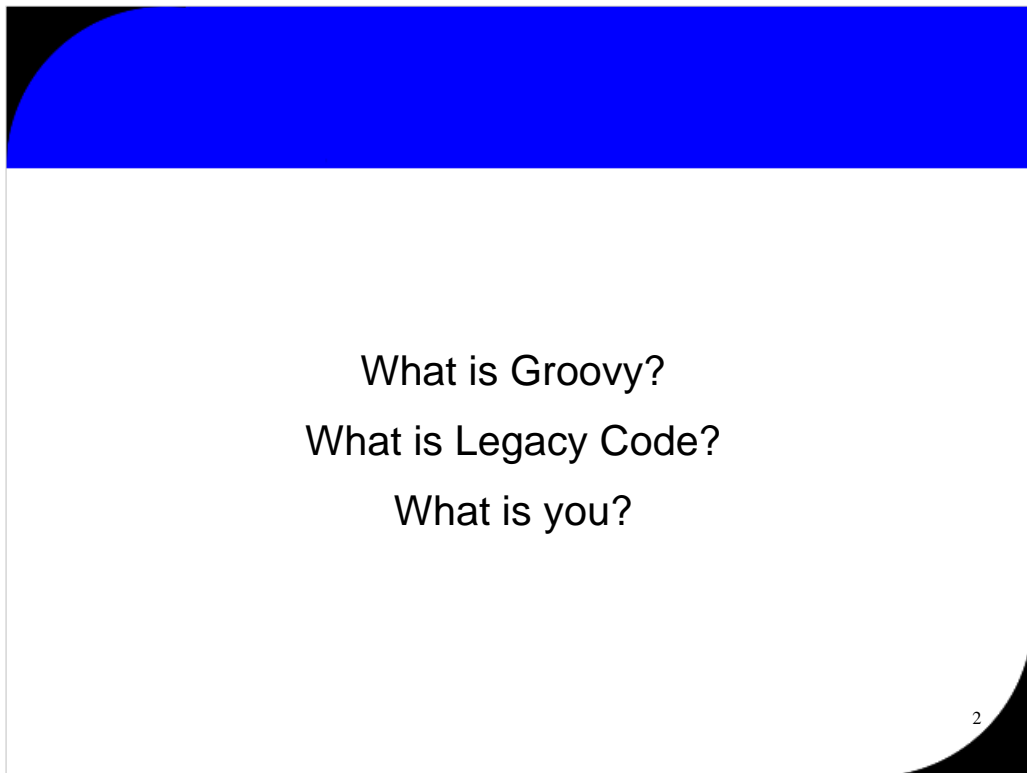
Groovy, Legacy Code, and You

Hamlet D'Arcy
Pearson

@HamletDRC
hamletdrc@gmail.com
<http://hamletdarcy.blogspot.com>

Slides: <http://is.gd/35Qih>



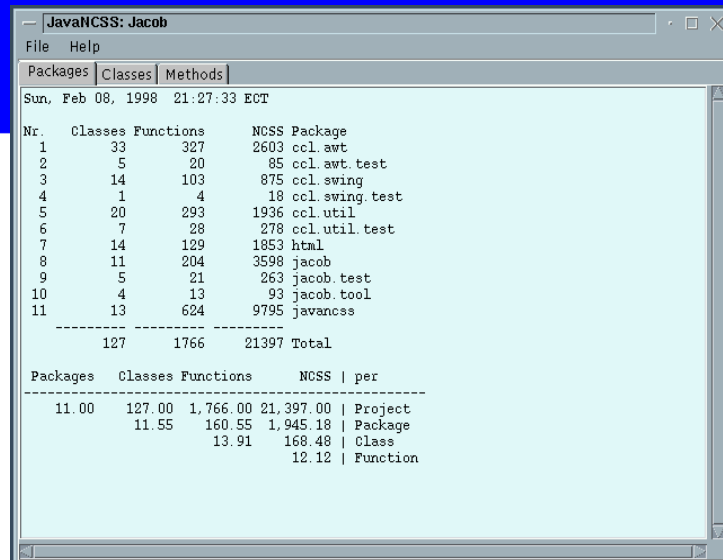


Establish Agenda- It's not about me it's about you.

- 1) What is Groovy – duh
- 2) What is LC – wait until next slides
- 2) What is you?

What is your longest class? You /should/ know. Be proud about reducing the line count.

Find people with legacy code and pester them with questions... will this work?



The screenshot shows a window titled "JavaNCSS: Jacob" with a menu bar (File, Help) and tabs (Packages, Classes, Methods). The main display area shows a table of code metrics for a project, dated "Sun, Feb 08, 1998 21:27:33 ECT".

Nr.	Classes	Functions	NCSS	Package
1	33	327	2603	ccl.awt
2	5	20	85	ccl.awt.test
3	14	103	875	ccl.swing
4	1	4	18	ccl.swing.test
5	20	293	1936	ccl.util
6	7	28	278	ccl.util.test
7	14	129	1853	html
8	11	204	3598	jacob
9	5	21	263	jacob.test
10	4	13	93	jacob.tool
11	13	624	9795	javancss

	127	1766	21397	Total

Packages	Classes	Functions	NCSS	per

11.00	127.00	1,766.00	21,397.00	Project
	11.55	160.55	1,945.18	Package
		13.91	168.48	Class
			12.12	Function

Java NCSS

<http://www.kclee.de/clemens/java/javancss/>

3

Java ncss - "Non Commenting Source Statements"

A fun diversion for a Friday. Find your longest class and longest methods.

Legacy Code is...

Stable?

Testable?

Tested?

Easy to change?

Defect free?

4

"Legacy code is code we've gotten from someone else". Feathers

LC is code is difficult to change.

LC is demoralizing and not fun to work with.

LC is a trap your resume will not survive.

"code without tests" is a flippant answer.

Code w/out tests is bad code, but is it legacy?

Properties of LC:

Massive classes. Tangled dependencies. Low test coverage.

Uses service location and direct constructor calls

Legacy Code lacks the proper design you would give it today.

If you were confident you could change the code w/out breaking it then you'd just do it. But you aren't. For that you need tests.

Legacy code is code you'd like to change but can't.

In practical terms, this means LC is code without good tests.

Good definition b/c it points to a solution.

Solving Legacy Code

Know where you want to go

Learn how to get there

Apply 51% Rule

5

Solving it does not mean eliminating all accidental complexity (sorry Stu)

Know where you want to go and then step in that direction. Don't solve all problems. Set and remember priorities. Can't solve every issue.

Learn how to get there – This presentation. WELC. Refactoring. Refactoring to Patterns. All good books.

51% Rule - If you know of an change you can make to improve the code base, and you're making that change 49% of the time, then your problem is getting worse. If your QA dept wants to do automated testing, they might hire an automation expert to reduce the time it take to to regression. So if there is one person writing automated tests cases, and 4 writing manual test cases, what is happening to the time it takes to do regression? It is increasing with each release of the software and your problem is getting worse. To improve a problem, you need to staff so that you're doing the right thing 51% of the time. Unit Test coverage isn't exactly addressed by this issue because quality is not directly correlated to test coverage. Be careful making this argument. Teams tend towards the extremes. a 49% practice wil tend towards 0 over time and a 51% will tend towards 100%. Don't need to get management to buy off on 100% improvement, just get them to sign off on 51% and let momentum do the rest.

Problems You'll Encounter

Cultural Issues

Glacial Pace


Cut and Run

6

Culture Issues - Not everyone on the team is going to be interested in modernizing the code. Some people are more than happy to continue what they're doing. What do you do about these people? Are there more of them than there are of you? That is a problem. What to do about crummy co-workers when you don't have firing power? I don't know.

Pace - The pace is glacial. It won't look like there is much progress but you're moving a glacier. How to not get discouraged? But, improvements are either incremental or excremental. There is a smaller scale pace that can improve: "Breaking down a big class into pieces can be pretty involved work unless you do it a couple of times a week. When you do, it becomes routine. You get better at figuring out what can break and what can't, and it is much easier to do."

Cut and Run - If you can have a high level, abstract discussion about what to do about legacy code (like this presentation) then you're probably good enough to find a different job. *Is your mistake staying at places too long, long after the learning curve flattened?* You need to find a way to stay motivated.



God grant me serenity to accept the code I cannot
change, courage to change the code I can, and
wisdom to know the difference.

Erik Naggum (1965-2009)

7

Applying yourself to every problem will frustrate and burn you out.

Find you what code needs to change and what code to leave well enough
alone.

This presentation will hopefully answer some of these questions for you.

Table of Contents

- 1) Better Testing with Groovy
- 2) Better Testing.
- 3) Changing Culture

TOC at page 8, not bad

Notice the full stop after #2

Let's write unit tests in Groovy!

Competing ideas:

- 1) "Testability in Groovy is easy"
- 2) Groovy is a "notational convenience"

9

What do you expect get from using Groovy? Are you introducing Groovy to aide in testing, or as a stepping stone towards using it in production? What do you expect to get out of using Groovy to unit test?

What is the reality? Next few topics will show.



A Success Story?

10

! means “Yeah Man”. ? means “No”.

Long project used Groovy for unit tests. In general, those that authored Groovy unit tests were pleased, those that didn't weren't. Coverage was very high. We didn't spend less time writing tests, but did write 3-4 times as many tests. Error conditions and edge cases were all better tested. Started adding Groovy tests to shared code and triggered an “enterprise review session”. It was decided to not proceed with any more Groovy development.

Next Slides are the advantages of using Groovy.

Is Groovy a Testing DSL?

```
def input      = ['Jane','Doe', 25.0, 4]
def expected   = ['Jane', 'Doe', 100.0]

assertTransformation(expected, transform(input))
```

11

Terseness of syntax inherent to Groovy language makes tests clearer:

Mapping design documents and requirements to test cases is simplified.

Does not require overhead of FIT or similar framework. It's just the language.

Don't underestimate Groovy's ability to remove the non essential from a test case. Groovy tests are just better.

Default Parameters

Unit testing is about building tons of data

```
JFrame f1 = makeLoginWindow( )  
JFrame f2 = makeLoginWindow( "username" )  
JFrame f3 = makeLoginWindow( "uname", "pword" )
```

12

Unit testing is about building tons of data, all in different configurations or states of construction, and then throwing that data at the system under test, observing and verifying the results. Unit tests look a lot different from production code. In production code, subroutines exist to perform smaller pieces of a larger, decomposed problem. In test code, subroutines exist to build objects. The design principles between production and test are different, and it makes sense that some language features might be very useful in test but of more limited value in production.

Default Parameters

```
JFrame makeLoginWindow(String username = null,  
                        String password = null) {  
  
    def userField = new JTextField()  
    userField.text = username ?: ""  
  
    def passwordField = new JPasswordField()  
    passwordField.text = password ?: ""  
  
    def frame = new JFrame()  
    frame.contentPane.add(userField)  
    frame.contentPane.add(passwordField)  
    frame  
}
```

13

There is a single source for all the test data configuration, and navigating (ie reading) the test case is made easier by having a single source. Conceptually, the Java version requires 3 added entities to the test case that didn't exist before. Navigating around labyrinths of production code is a necessary evil. There are other design constraints at play and stuffing everything into one place isn't always the right thing to do. But in the test tree, labyrinths of chained code are just evil, nothing necessary about it. Having to chase down dependencies undermines one the chief principles of testing: show clearly and simply how input is transformed to output.

Builders!

```
def element = new OMElementBuilder().root {  
  child(attribute: 'sample1')  
  child(attribute: 'sample2')  
}
```

14

Writing your own builder rocks.

Builder support in Groovy is a big win.

Builders?

```
OMElement e = AXIOMUtil.stringToOM(  
    "<root>" +  
        "<child attribute=\"sample1\" />" +  
        "<child attribute=\"sample2\" />" +  
    "</root>");
```

15

But is dynamic typing that much better than stuffing something in a String? Which is more type safe?

Your production code is written in Java, and presumably most your libraries. Making some creative factory methods for String and File types can go a long way towards cleaning up Java tests.

Builders really shine when mixed with iteration of list data.

Groovy Assertions

```
assert new File('foo.bar') == new File('example.txt')  
  
Caught: Assertion failed:  
assert new File('foo.bar') == new File('example.txt')  
      |                   |  
      foo.bar           example.txt  
                        false
```

16

This is better than stack trace (although that is still present). We've made mountains out of molehills debating alternatives to “assertEquals”. What is more natural than using the assert keyword?

Also, better stacktraces are probably on their way and already planned for Spock 0.2.

Cooler Tools?

Like easyb and Spock

But..

Behavior Driver Development is different
and Spock is very different

17

easyb won the Jolt award. It must be great, right?

BDD is maybe a great idea. Spock is maybe a cool framework.

I like easyb for heirarchical test fixtures, regardless of BDD claims.

Are these too radical of ideas? What problem will you solve by introducing these ideas? Are you just trying to get people to write plain old JUnit tests?

Multiline Strings

```
def garage = new Garage(  
  new Car(CAR_NAME1, CAR_MAKE1),  
  new Car(CAR_NAME2, CAR_MAKE2),  
  new Car(CAR_NAME3, CAR_MAKE3)  
)  
  
def expected = """  
  <garage>  
    <car name="$CAR_NAME1" make="$CAR_MAKE1" />  
    <car name="$CAR_NAME2" make="$CAR_MAKE2" />  
    <car name="$CAR_NAME3" make="$CAR_MAKE3" />  
  </garage>"""  
  
assertXmlEqual(expected, serialize(garage, Garage))
```

18

There is symmetry and conciseness, and I find it pretty easy to see how the input relates to the expected output.

Multiline Strings

```
Garage garage = new Garage(  
    new Car(CAR_NAME1, CAR_MAKE1),  
    new Car(CAR_NAME2, CAR_MAKE2),  
    new Car(CAR_NAME3, CAR_MAKE3)  
);  
  
String expected =  
    "<garage>" +  
    "  <car name=\"" + CAR_NAME1 + "\" make=\"" + CAR_MAKE1 + "\" />" +  
    "  <car name=\"" + CAR_NAME2 + "\" make=\"" + CAR_MAKE2 + "\" />" +  
    "  <car name=\"" + CAR_NAME3 + "\" make=\"" + CAR_MAKE3 + "\" />" +  
    "</garage>";  
  
assertXmlEqual(expected, serialize(garage, Garage.class));
```

19

I can't see how the expected block lines up with the input anymore, it's lost in a sea of accidental complexity, back slashes, and quotes. The tests as documentation idea starts to suffer without multiline strings because the only thing that finds this format easy to read is the compiler. Speaking of documentation, how about copy and pasting the XML snippet into a user document or email? Or heaven forbid generating user documentation off the test case!

Partial Interfaces with Maps

Very useful for wide interfaces:

```
def x = [ mouseEntered : { ... } ] as MouseListener
```

20

Wish I had better example from JDK

This is insanely useful. Especially in a system with many wide interfaces.

Partial Interfaces with Maps

```
boolean wasCalled = false

def listener = [ mouseEntered : {
  wasCalled = true
} ] as MouseListener

// ... exercise system under test

assert wasCalled
```

21

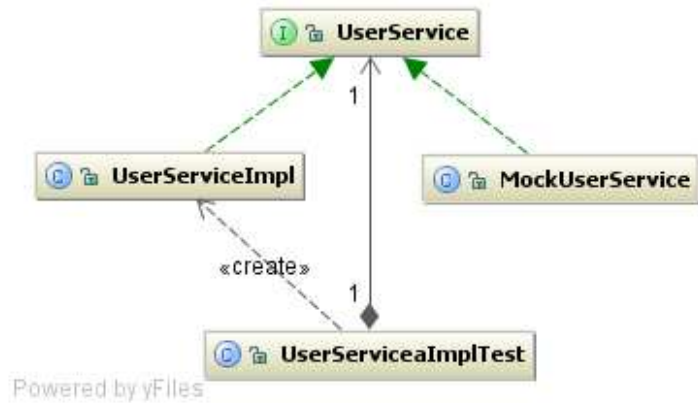
There is no verification/mock phase. This is best for stubs, in which it works extremely well.

Problems: Find usages don't always work.

Refactorings don't always work. Mocks lead to fragile tests. Mockist style is same as consultant style. Once you live with mock code for 6 months+ you feel the pain. Talk about mocks more.

Exceptions aren't always handled properly

Abstractions?



22

The quality of the feature is dependent on the quality of your abstractions. Extracting an interfaces on top of every class just to mock it out creates a meaningless abstraction.

Fake objects - is this an advantage of Groovy? Yes and No. Better for stubs, not for mocks.

Beware the meaningless abstraction.

Private Method Access

Insanely Useful?

Insanely Evil?

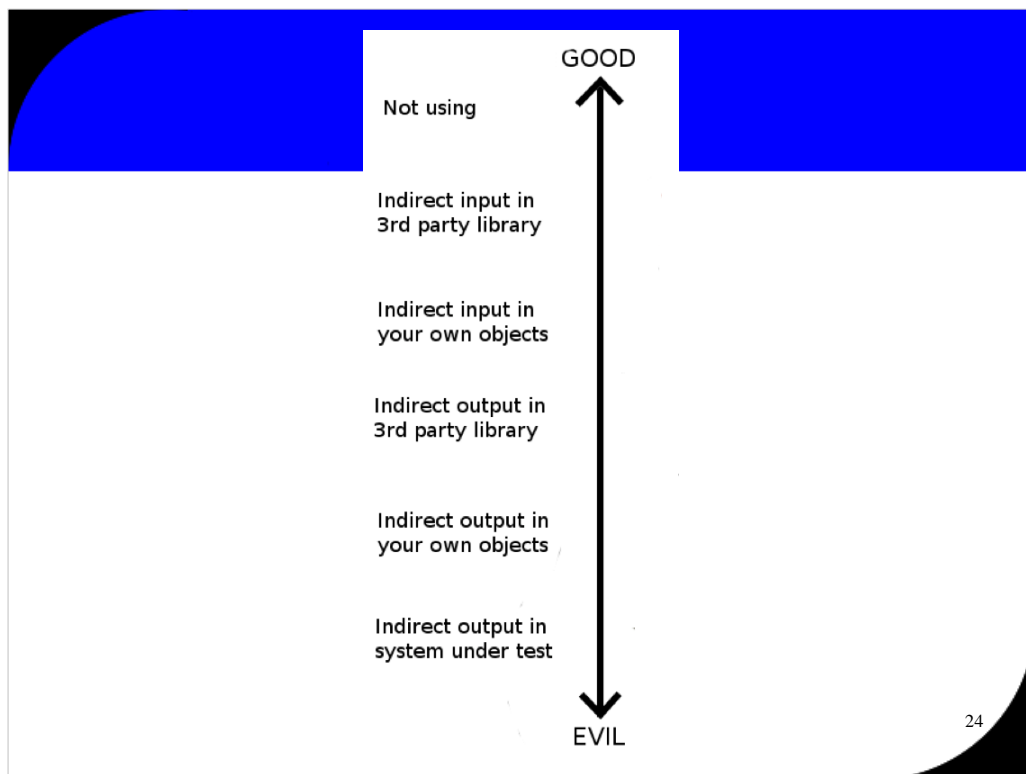
Understand direct vs. indirect input and output

23

Groovy ignores the private access modifier in code
Insanely useful?

Insanely boring debate on whether you should test
private methods?

It's both useful and evil, depending on how you use it.
Define input/output and direct/indirect distinction.



I don't often ask myself "What would Jesus do?" I try to live my life more like Pilate. I try my best but, hey man, sometimes stuff happens

Next few slides: "Hitting the Java Wall" ... benefits start to disappear.

mockFor and stubFor

```
public class Service {
    static long getFileSize(String path) {
        return new File(path).length();
    }
}

def mock = new MockFor(Service)
mock.demand.getFileSize { -1 }
mock.use {
    def actual= Service.getFileSize('file.txt')
    assert -1 == actual
}
```

25

mockFor and stubFor demo well

Easily mock out static method calls and constructor invocations

MockFor and Stub for effect only Groovy classes. The required indirection does not exist in Java compiled classes. To mock out constructors and static method calls you'll need another tool like:

<http://code.google.com/p/powermock/>

<https://jmockit.dev.java.net/>

The Flow of Groovy

"I'd rather program Groovy in vi
than Java in IDEA"

- Neal Ford

"If vi is the answer, the question had
better be about Roman numerals"

- Me

26

But I see his point in pure Groovy projects. Doesn't carry into Java projects.

"Changes are often easier that you would expect because you can make changers to the tests and then make changes to the code, moving the structure along in small safe increments". Michael Feathers

Especially true in Groovy. You rarely needs more than 2 editor windows open at once... production and test. In Java, when you make a slight change, you need to fix every single instance before the compiler lets you even run a unit test. Not so in Groovy. Your workspace is conceptually 2 files: unit tests and class under test. Only when you decide to broaden the scope do the other tests get compiled and run.

In mixed Groovy/Java projects, all production code must compile. You're still stuck wading/mired in fixing all compiler warnings before getting test feedback on your changes.

Next Slides: Downsides of using Groovy

Downsides of Groovy

Lost benefits of TDD?

Questionable IDE support?

No Pain No Gain?

Is this a language decision?

27

Idea is possibly from Dierk König [cite needed]. Using TDD and Groovy will evolve your system to have a lot of functions that operate/return lists. And a lot of reuse of closure/runnable/callable types. This is a lot different than the API you'd produce by test driving the code with Java unit tests. The API design must end up being sub-optimal when called from other production code. Should you care?

IDEA has great support. But not perfect. Sometimes tests don't run. Sometimes your caches need to be cleared. Refactorings don't always work or you get a lot of false positives. Will the entire team have the patience to deal with this? JetGroovy is not perfect. And when it fails, Groovy will be blamed and people will come to you all a dither.

"It might sound kind of sadistic, but the pain that we feel working in a legacy code base can be an incredible impetus to change. We can take the sneaky way out, but unless we deal with the root causes, overly responsible classes and tangled dependencies, we are just delaying the bill" Michael Feathers. The risk is that Groovy (or a mock framework) cover the pain points to where your code sucks worse than it did before. IDEs in general are guilty of this. Massive projects don't exist with eMacs, pain is too great.

Is putting groovy.jar into your codebase a language decision? There is no real answer. In some ways yes, in some ways no. Some companies will shy away from Groovy claiming it is a language decision. Some companies will embrace it saying it is Java++ (this isn't my experience though). This is a big decision. Treating it like it isn't will not help you sell your idea.

What to make of Groovy...

Competing ideas:

“Testability in Groovy is easy”

Groovy is a "notational convenience"

28

Now you've seen the evidence. Which of these techniques is going to make your production code more testable? Any?

Is debating and investing in this the best usage of your time? Would you be better off doing a Refactoring book club? Or Refactoring to Patterns book club.

Either you want to use groovy or you don't. Is any of this really convincing?

Writing tests in Groovy is definitely more enjoyable. Writing complete tests is definitely easier. Getting the first test written on a monster object is going to be painful either way.

What's missing is the cultural benefits (more on this later).

Refactoring vs. Reengineering

The First Step in Refactoring: Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code.

Martin Fowler - Refactoring

I insist that you pair when you use the dependency-breaking techniques I've described in this book.

Michael Feathers - WELC

29

Edit and Pray vs. Cover and Modify

Refactoring by Martin Fowler - I still believe this is **the** book to read when starting out on the path to being an engineer. Refactoring laid out a vision of good, clean code that went beyond simply working correctly. The beauty of Refactoring was that it laid out specific instructions on how to make crummy code better. And those instructions were almost mechanical in their execution... first do this, then do this, then finish.

Why is refactoring a word you can barely mention to QA and Operations without horrified reactions? Because nobody starts with tests.

Lean on the Compiler

Legacy Code Dilemma:

Don't change code without tests
Can't write tests without changing code

30

Legacy Code Dilemma - When we change code, we should have tests in place. To put tests in place, we often have to change code. Michael Feathers defines it differently. Oh well.

"If a tool can extract methods for you safely, you don't need tests to verify your extractions." But can't reorder statements, especially instance methods. Make a series of changes with tools only and you'll be safe. There are a ton of changes you can make like this, including Extract Class the most important one.

"Judgement is a key programming skill, and we can get into trouble when we try to act like super-smart programmers." Feathers

Lean on the Tools

javac

Checkstyle

IDE Inspections

31

final local variables and final fields - this coding convention can be very revealing. Move towards these patterns for better composed methods. No more populate() methods.

Broken Windows - "If I can remove tiny pieces of duplication, I do those first because often it makes the big picture clearer." Michael Feathers
best to get code into state where there are no compile warnings.

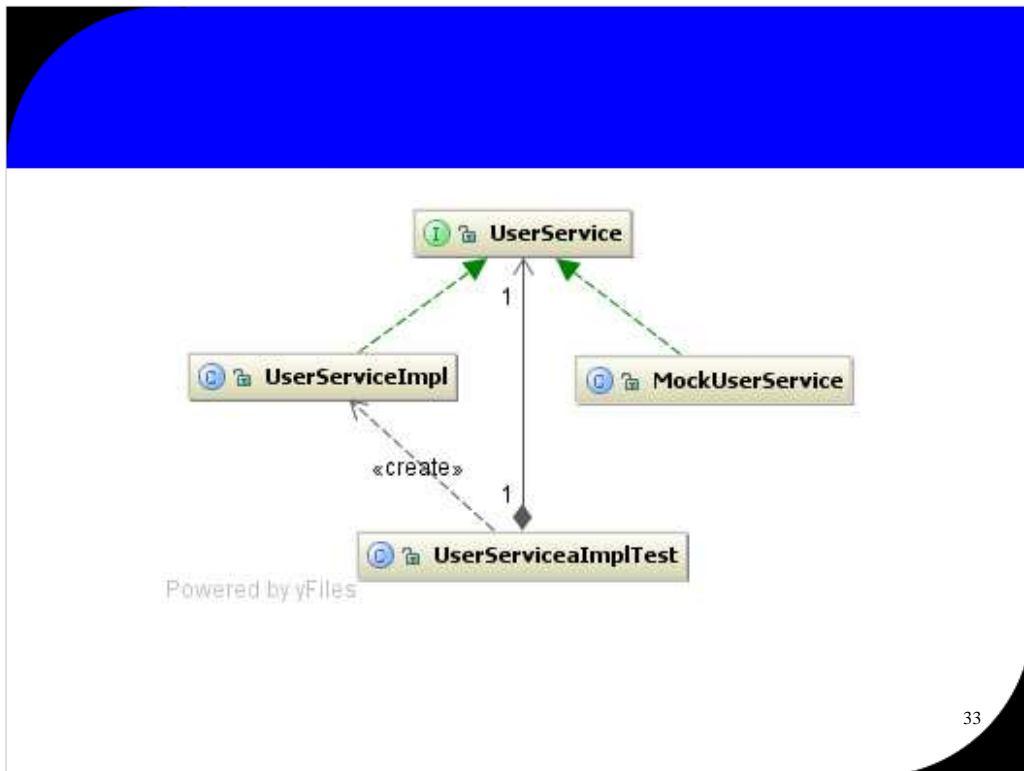
Testability vs. Readability

"...some of the attributes that support good testability also make for poor readability."

- Walter Harley

32

"...some of the attributes that support good testability also make for poor readability. For example, loosely coupled objects make it easy to supply mocks for testing; but they also make it hard to see what's going on in the production code."



33

what to do about it? The answer is all about abstractions. Are you creating meaningful or meaningless abstractions? A useful abstraction is not the same as a meaningful abstraction.

Tools like Class Extensions - is not a solution to interface/impl problem. What does it really add? A few less classes? Over time this is not the right choice.

Readability suffers in Java because of our culture of naming things, and we write our abstractions based on name instead of type. Readability suffers in Java because we have become supremely efficient and versed at producing meaningless abstractions.

Long term readability depends on finding type based abstractions, rather than name based ones, and then being diligent and intentional about coding to those abstractions.

Use appropriate data types, even at the micro level. Strings and ints are not Files and Enums!

You Don't Need BPM

BPM == Big Procedural Methods

```
public void save() {  
    queryA();  
    performStepB();  
    queryC();  
    performStepD();  
    ...  
}
```

34

"There is something mesmerizing about large chunks of procedural code: They seem to beg for more" Michael Feathers

"one pervasive problem in legacy code bases is that there often aren't any layers of abstraction; the most important code in the system often sits intermingled with low-level API calls... Code is harder to understand when it is littered with wide interfaces containing dozens of unused methods. When you create narrow abstractions targeted toward what you need, your code communicates better and you are left with a better seam." Michael Feathers

Command/Query separation A method should be a command or a query, but not both. A command is a method that can modify the state of the object but that doesn't return a value. A query is a method that returns a value but that does not modify the object. Michael Feathers

Break Out Method Object

```
public void save() {  
    new MyObjectSaver(...).save();  
}
```

35

Break Out Method Object - Ward Cunningham - Move big method to it's own class.

Allows you to test the container of the save() method by mocking out the MyObjectSaver. In general, pass more on the constructor and less on the save() method (maybe one object at most, the entity being saved). This will aide later in creating narrow interfaces. This is about bringing the class with save() method under test.

Extract Class

```
public void save() {  
    new StepASaver(...).save();  
    new StepBSaver(...).save();  
    new StepCSaver(...).save();  
    ...  
}
```

36

This helps you bring each step under test more than the host.

Is this a great domain model, no. Is it better than alternative? Quite likely. Are these meaningful abstractions? No. How do we fix that? This does increase conceptual complexity. Programmers have developed a sense of how things work correctly that you are breaking.

Extract Class

- 1) Extract Method until the BPM is composed
- 2) Extract Parameters until each method is static
- 3) Create a new static inner class
- 4) Move methods to inner class
- 5) Make methods non static
(compiler will tell you if you error)
- 6) Elevate class to top level and new package

37

This is a completely automated refactoring.

Meaningful Abstractions and BPMs

```
public void save() {  
    List<Saver> steps = ...  
    for(Saver step : steps) {  
        step.save();  
    }  
}
```

38

Grouping things together because they happen at the same time is a weak relationship. The next step from Extract Class on BPM could be this example.

Each step can implement a narrow interface. This can be meaningful. Then the save methods can be a general composite. As more save methods follow this general composite you have another meaningful abstraction. Now you've separate /how/ an object is saved from what it does.

Maniacal Focus on Dependencies

```
private void createWheel();  
private void makePedal();  
public void drive();  
public static Car build(DataSource);  
private void buildBody();
```

Maniacal Focus on Dependencies

```
private static Wheel createWheel(DataSource);  
private static Pedal createPedal(DataSource);  
private static Body buildBody(DataSource);  
public static Car build(DataSource);  
public void drive();
```

40

2nd Example points to a CarBuilder object more easily than the first.
Add in 20 other methods and you'll never see the dependencies.

static methods declare their direct dependencies in their type signature. This is good.

static methods can be moved to a new class with one keystroke. This is good.

static methods declare, instance method hide.

Finding hidden classes: Look at names and type signature. Must have naming convention and type ordering convention.

Moving towards this is set of automated refactorings.

Make the methods **functions** - functional functions make caching/optimization/benchmarks trivial.

Maniacal Focus on Dependencies

static methods declare their direct dependencies
in their type signature. This is good.

static methods can be moved to a new class with
one keystroke. This is good.

static methods declare, instance method hide.

Dependencies, Dependencies, Dependencies

Inversion of Control is your friend

- Passing only what an object needs
- Works against using the appropriate type

42

Inversion of Control is your friend

Break dependencies by passing only what an object needs, not a wrapper with a getter

In extreme, works against using the appropriate type

Sweet spot is declaring dependencies on something other than primitives

Strive for system that requires Stubs to be passed, not mocks. Indirect input OK, indirect output (side effect) needs to be limited to certain areas.

Snipping Dependency Trees

- How do you introduce Spring into legacy code?
- Should each class load an ApplicationContext?
- Is Guice a better choice?
- Why not a Singleton ApplicationContext?

43

Your object graph is usually a tree. Chose point to start DI and just run with it. Somehow incent people to do this. Pick a branch and say "Everything below branch x is DI, everything above it is Service Location".

- 1.Extract Field on heavyweight local variables in class, instantiating them in the constructor.
- 2.Extract parameter in constructor
- 3.When constructor parameters become unweildy, introduce DI container
- 4.Move globals to Spring with factory-method beans

Step 2 : Over time constructor parameter lists will become huge as large classes have their dependencies exposed list this. This is symptom of problem, not problem itself. As part of the feature work, working from the bottom, strive to place an application context as high up the dependency tree as you can, and snip those dependencies.

Snipping Dependency Trees

- Look for reflection in code
- Look for interfaces with tons of subclasses
- Do **not**: write ApplicationContext for unit tests.

44

Stake out areas that have natural entry points. Is there an interface with 40 subclasses, all which are instantiated using reflection? Look for reflection and insert Spring here. I don't mind seeing class instantiation through reflection because it usually means that I can easily convert it to use Spring.

Spring configuration for unit test a la UnitTestInitializer

- Not a good idea in the long term.

Busy Constructors

```
public class UserService {  
    private final DataSource dataSource;  
    private final UserSettings settings;  
  
    public UserService() {  
        dataSource = DataSourceFactory.getDefault();  
        settings = new UserSettingsImpl();  
    }  
}
```

Busy Constructors

```
public class UserService {  
    private final DataSource dataSource;  
    private final UserSettings settings;  
  
    public UserService() {  
        this(  
            DataSourceFactory.getDefault(),  
            new UserSettingsImpl());  
    }  
  
    UserService(DataSource dataSource,  
                UserSettings settings) {  
        this.dataSource = dataSource;  
        this.settings = settings;  
    }  
}
```

46

Adding constructor parameters - overload constructor and chain the two so that other clients don't need to be recompiled. **THIS IS GOOD IDEA!** and alternative to parameterize constructor. In Groovy, which has default argument values, there is a simpler way to support extract parameter in methods.

More Static Dependencies...

```
public class UserService {  
    private DataSource datasource;  
  
    public void create(UserDTO input) {  
        User user = new User(datasource);  
        user.setFirstName(input.getFirstName());  
        user.setLastName(input.getLastName());  
        user.save();  
    }  
  
    // read, update, and delete methods omitted  
}
```

47

You could introduce a `UserFactory` for this, but that is overkill and a meaningless abstraction. You could use `MockFor` to test this, but only if the source is Groovy. Instead subclass and override:

More Static Dependencies...

```
public class UserService {  
    private DataSource datasource;  
  
    public void create(UserDTO input) {  
        User user = makeUser();  
        user.setFirstName(input.getFirstName());  
        user.setLastName(input.getLastName());  
        user.save();  
    }  
    protected User makeUser() {  
        return new User(datasource);  
    }  
}
```

48

This is good work around for using constructors in classes and you don't need factory objects. Now you can create a test specific subclass in your test and just test using that. In this case... **the system under test is a fake object**. That is OK.

Undetermined Effects

“But we can't make changes because it might break dynamic class loading!”

- NoSuchMethodException
- ClassNotFoundException
- Ever heard of JUnit?

49

What keeps breaking? Is it the same weird edge case? Can you write a really unique unit test for it? Can you test the package and class name of every class loaded through reflection? Can you use ASM/BCEL to make sure dependencies don't change? Did this with InvocationAnalyzer.

Using reflection? Write a unit test to make sure you don't get tripped up.

A Different Type of Testing

"I write unit tests for one reason: so my coworkers
don't **** up my code."

David Angry



50

Testing supposedly makes things easier to change
JUnit tests often make things harder to change
Is your goal to make future changes easier, verify your
own changes, or stop backwards regression?

A Different Type of Testing

Unit tests on legacy code are different

Is the answer functional tests?

51

Unit tests on legacy code are different:

Often test what system does instead of what it is supposed to do

Often requires extensive mocking b/c of side effects

Produces fragile tests in which refactoring breaks tests.

Pro: Solidifies changes so production code can't be reverted back to inferior state. Once a test is written, no one will revert code back.

Often the case w/ inner classes and stuff.

Is the answer functional tests?

Functional tests will stand up to crazy big refactorings

But DB tests don't scale. Best use is get in, lay coverage, refactor, retest, rip out.

But you will never want to throw test code away!

As test bed grows, failures will grow more frequent.

Functional tests won't dig you out of hole alone.

Spring test runner is cool... but... use should be focused and rare.

For both approaches: Write tests for the "area" where you will make your changes, not on the function to change.

A characterization test is a test that characterizes the actual behavior of a piece of code.

BDD seems like a pipe dream when you're asking yourself "how the heck am I ever going to test this?"

Scratch Refactoring

- Set a timer (1-2 hours?)
- Tag your code (git, Local History)
- Refactor without tests
- Step back and analyze... is it better?
- Revert to previous save point if not

52

The best technique for learning about code is refactoring... Scratch refactoring is a good way to convince yourself that you understand the most important things about the code, and that, in itself, can make the work go easier.

git or IDEA Local History greatly helps with experimentation and rollback.

Experiment: When things do suck then just wack them. Be ruthless: Don't hold on to something because it took you a long time to write.

Get the Most from Code Reviews

- Analyze import list
- Before and after class length
- Incent reducing lines count in large classes
- Keep notes and post results

53

Code Reviews: Get People to start doing them. Some will, some won't. It won't bring the bottom people up, but it will bring the intermediate people up to a higher level

Import List - What is being imported? What data types are your model, File? What data types are your view, JPanel and JTextField? Is the file mixing the two? The imports clearly show this on the screen without scrolling.

Class Size Delta - look at before and after class length. Did the person's change just raise the linecount from 5500 to 6000? Is this something you can enforce as a metric that can't climb. ncss is a tool too look at this. JavaNCSS is a fun toy to look at.

How long should a class be? Who cares. You know your classes are too long, so just make them smaller.

Avoiding the Big Band Aid

- Do the right thing incrementally
- Avoid “Patterns of Enterprise Cruft”
- Have a 2 year plan for the codebase
- Avoid “Death by 1,000 Clever Ideas”

54

Writing big band aids is fun, like a Generic Object Factory. It's not fun if 5 teams come up with a slightly different band aid. Death by a thousand clever ideas. (Clever is never a compliment. People like ideas that they should have thought of not ideas the never would have thought of. Repurposing existing objects is clever, simplifying the object model is intelligent.). Things like generic object factory aren't moving you towards solution, just towards short term testability.

Beware solutions that require heavy mocking to test

Unsolved Mysteries

- Where do you start?
- Unreasonable use of instance fields
- Database compatibility
- Tumbleweeding
- Your problem at work

55

Where to start – start at the most unstable code... the class that has the highest revision # in version control

Unreasonable means beyond the scope of human or computer reason. Such as impossible to reason about the usage of instance variables. In a giant method, set a field to a value at start, then null at end, then refer to it every else. Horrible Practice. What to do?

What to do about the database? You can't usually change it. Reverse compatibility issues?

Tumbleweeding – Prevented by accountability: pair programming & daily standup. Working remotely and working alone will get you in trouble.

My Co-Workers are Idiots

I assure you, they are not

Term	IQ Range
Moron	51-70
Imbecile	26-50
Idiot	0-25

Assume positive intent

You probably just don't know what motivates them.

We Need to Restart

- Big Rewrite vs Big Refactor vs Big Rehire
- Change is either incremental or excremental
- I'd rather staff to 51% for incremental changes

57

You'll wait forever to get funding for something.
Although being involved in these restarts is always fun at first... just don't stick around until the end.

I Hate My Job

- Legacy code is demoralizing
- Promote positive change
- Do something uselessly cool outside work
- Use Groovy

58

Need to fight resignation.

Can you promote testability? We had a 100% Club and Shanghai'd to Excellence promotion. Build trophy is shame and negative, so pick something positive. Make it public. This isn't about creating change as it is about creating a shared project mythology. Changing legacy code is a long slog. Manufactured events provide events of shared experience. Much like a family vacation... it can suck for 7 days straight, but one drunken bonfire makes the trip memorable for the next 40 years.

Pictures: Build Trophy, Shanghai'd, 100% Club.

Morale ebbs and flows. Some months are better than others. Do something radically different outside work. TDD outside work to get in flow. Find way to bring it back.

Design By Resume - Using Groovy in order to design by resume. Is that always a bad thing? "Reversing rot is not easy, and it's not quick.... take work, time, endurance, and care". How do you hire great people into a legacy code shop? How do you keep good people? Groovy in job description is exciting.

Shanghai'd to Excellence



Avast, Athena Team!



*The Scourge of the Black Seas known as
vue.exam be approachin' 1000 unit tests.*

*Pirate's bounty awaits the lucky buccaneer
who checks in the 1000th unit test.*

*And a keel haulin' awaits all if we don't get
there by the New Year.*

Consider ye self warned, matey.

**You wrote
some ill
unit tests
Son.**



**Athena
Thanks
You**



I Can't Do it Alone

- No, you can't
- Requires large commitment
- Or no commitment at all

61

Contractors are your friend. It looks good on resumes and they don't get in trouble to bend rules.

Can you tell contractors what the expectation is for code coverage and then hold them to it? They are much more likely to do it.

If your code has cross team dependencies then you can't make a decision for that other team.

Getting classes into test harnesses requires a commitment. Who is committed? You, managers, team mates? Everyone?

We Can't Get Traction

- Start with TDD for defects?
- Define code coverage minimums?
- Find a new team?

62

TDD for defects? Not a bad way to start. Writing tests for legacy code are the hardest tests to write. If your team has never written tests before then they will decide that testing is not worth the effort. Testing for defects in legacy code are some of the hardest tests to write.

Defining Coverage Minimums - Is spending effort on getting a tool in place really worth your time? Wouldn't you be better off adding coverage blindly because you know you need it everywhere? Where does management stand on this? If people aren't writing unit tests then why aren't they on an improvement plan? Does management really want to add more coverage? What problem are minimums fixing? And will it really work? This seems unlikely to work to me.

If you team is staffed so that only 1 of 5 developers writes unit tests then the situation is hopeless. Find the other craftsmen and work to get on a team together.

They Won't Let Us

- “Don't change production code to enable testing”
- “The changes you're making aren't designed well”
- “You're creating too many types”

63

At what point is testability a first class design concern? Do your co-workers agree with you? Where did the idea of not changing production code to enable testing come from? Who? TDD is kind of an elaborate way around this. If you've dug a legacy hole, you'll need to violate this to dig yourself out.

Goal isn't to create great design... goal is a design good enough to use as a wedge to break dependencies and allow us to test as we move forward

“Too many types” - Myth: These techniques are not increasing the number of types in your system. They are increasing the number of names in your system. The types were all there to begin with they were just all mixed into a God object that hid that from you.

Contract Driven Environment - arbitrarily enforcing only some rules is most problematic: no DAO but final methods. Blocks to mocking.

What you can do:

Consistency – learn the tradeoff between the value of consistency and the value of experimentation. Experimentation needs to be done invisibly... not in shared objects! And you need to be responsible when things go awry.

You can grow high-quality areas, but don't be surprised if some of the steps you take to make changes involve making some code slightly uglier. This is great ammunition for the people that like legacy code.

Summary

"The last hazards are fear and resignation: fear that we can't change a particular piece of code and make it easier to work with, and resignation because whole areas of the code just aren't getting any better."

Michael Feathers