TDD, where did it all go wrong?

Kent Beck Style TDD

Who are you?

- Software Developer for 20 years
  - Worked mainly for ISVs
    - Reuters, SunGard, Misys, Huddle
  - Worked for a couple of MIS departments
    - DTI, Beazley
- Microsoft MVP for C#
  - Interested in OO design
  - Interested in Agile methodologies and practices
- No smart guys
  - Just the guys in this room

huddle™

Intelligent collaboration for the Enterprise

- The #1 SharePoint alternative in the cloud

Agenda

- The Problem
- TDD Rebooted
- Red-Green-Refactor
- Clean Code When?
- Refactoring to Patterns
- Ports and Adapters
- Behavior Driven Development
- [Mocks]
- [Test Data Builders]
- Q&A

THE PROBLEM

Some good developers don’t want to write tests.

- Why? They are not idiots.

When we change implementation details we break tests, often dozens.

- Didn’t refactoring promise change without breaking tests?

We often write more test code than implementation code.

- There may be a reason why customers complain that adding tests is robbing us of productivity

Why do approaches like “Programmer Anarchy” and “Lean Software Development” want to drop test first.

- They see Test First approaches as unproductive – as obstacles to development

Why is the ‘duct tape programmer’ winning?

- They deliver functionality to customers faster, and quickly abandon tests as ‘slowing them down’

When we return to our tests – because they are broken it is often difficult to understand their intent

- Is it broken because the behavior has changed? Where is that behavior expressed?

We have large ATDD suites written in Fit or Cucumber

- They spend much of their life red

Customers don't engage with our ATDD suites

- And yet we spend a lot of time maintaining them

Our ATDD suites are slow to run and increase the cost of the check-in dance

- Developers start ignoring red results
- And we have to down tools to fix them

Developers don't want to write them

- Because they can't see the value return on their effort
- They just want to build software

Where did it all go wrong?

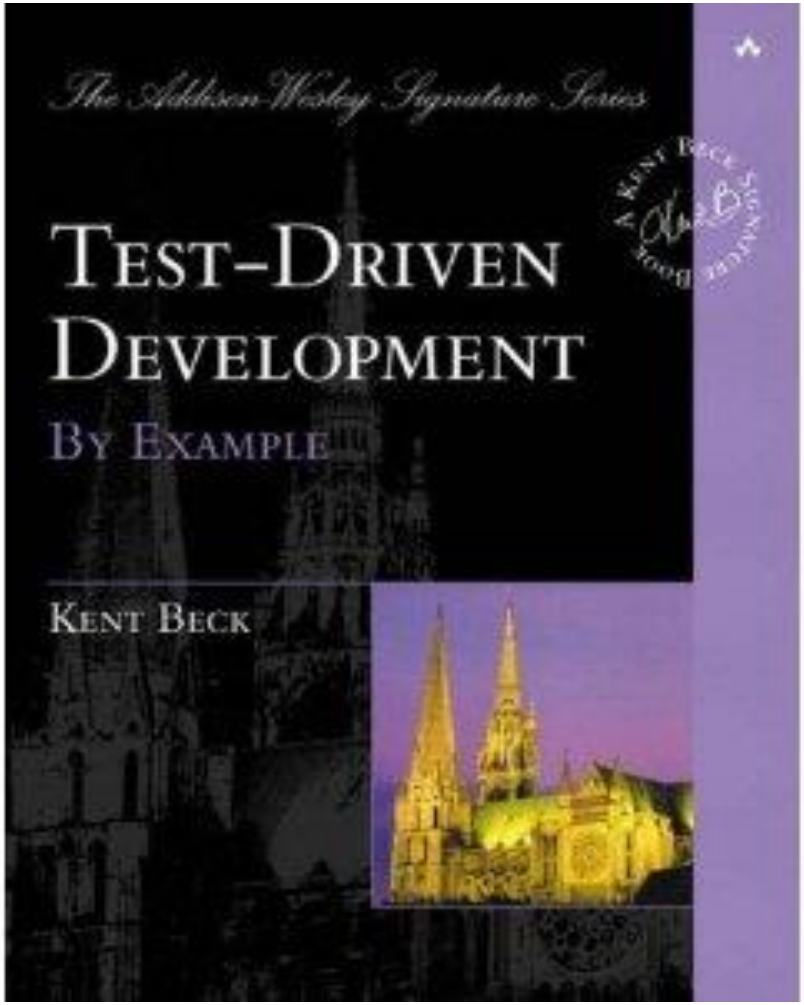
Let’s revisit the origins of our hero

TDD REBOOTED

Use The Source Luke!

A lot of folks have written TDD books. A lot of those books claim to be better introductions to TDD. For me however this remains the best text.

- It is simple and elegant.
- It covers the essentials of TDD, which have been misunderstood by many since.
- It covers more than just the basics with topics on patterns, smells, mocks etc.
- Good later work builds on it, but too much later work just misunderstands it
- The goal here is to go back and talk about it
- And then look at folks who built on it



The Zen of TDD

- Avoid testing implementation details, test behaviors
  - A test-case per class approach fails to capture the ethos for TDD. Adding a new class is not the trigger for writing tests. The trigger is implementing a requirement.
  - Test outside-in, (though I would recommend using ports and adapters and making the ‘outside’ the port), writing tests to cover then use cases (scenarios, examples, GWTs etc.)
  - Only writing tests to cover the implementation details when you need to better understand the refactoring of the simple implementation we start with.

Dan North

Agile Expert and Originator of BDD

“When we write a test, we imagine the perfect interface for our operation. We are **telling ourselves a story about how the operation will look from the outside**. Our story won't always come true, but it's better to **start from the best-possible application program interface (API)** and work backward than to make things complicated, ugly, and “realistic” from the get-go.”

t

Kent Beck, TDD By Example

What is a unit test?

- For Kent Beck it is a test that ‘runs in isolation’ from other tests
  - Nothing more, nothing less
  - It is NOT to be confused with the classical unit test definition of targeting a module
  - We don’t touch file system, database, because these ‘shared fixture’ elements prevent us running in isolation from other tests.
- Explicitly writing tests that target a method on a class, is not a TDD unit test
  - TDD unit tests focus on a story
    - Use-case, scenario…pick your poison!
- In fact focusing on methods creates tests that are hard to maintain
  - We don’t capture the behavior we want to preserve

Writing sinful code

RED-GREEN-REFACTOR

Red-Green-Refactor

1. *1* ***Red*** *Write a little test that doesn't work, and perhaps doesn't even compile at first.*
2. ***2.Green*** *Make the test work quickly, committing whatever sins necessary in the process.*
3. ***3.Refactor*** *Eliminate all of the duplication created in merely getting the test to work.*

Or

1. *Write a test.*
2. *Make it compile.*
3. *Run it to see that it fails.*
4. *Make it run.*
5. *Remove duplication.*

The Simplest Thing

“Now I'm worried. I've given you a license to abandon all the principles of good design. Off you go to your teams—“Kent says all that design stuff doesn't matter.” Halt. The cycle is not complete. A four-legged Aeron chair falls over. The first four steps of the cycle won't work without the fifth. **Good design at good times. Make it run, make it right.**”

Kent Beck, TDD by Example

How Soon is Now?

CLEAN CODE WHEN

Clean Code Now

- The Refactoring Step is when we produce clean code.
  - It's when you apply patterns (see Joshua Kerievsky Refactoring to Patterns).
  - It's when you remove duplication
  - It's when you sanitize the code smells
- You do not write new unit tests here
  - You are not introducing public classes
  - It is likely if you feel the need, you need collaborators that fulfill a role.

Clean Code When?

- Don’t test internals.
- Don’t make everything public in order to test it
- Preserve implementation hiding by keeping a thin public API
- Refactor implementation details out, so that they do not need their own tests
- Continue to refactor implementation details over time, as you want
- Have expressive tests that tell you can read in the future

Kent beck on patterns

REFACTORING TO PATTERNS

Use of Design Patterns in TDD

Pattern	Test Writing	Refactoring
Command		X
Value Object		X
Null Object		X
Template Method		X
Pluggable Object		X
Pluggable Selector		X
Factory Method	X	X
Imposter	X	X
Composite	X	X
Collecting Parameter	X	X

Kent Beck, TDD By Example

Refactoring to Patterns

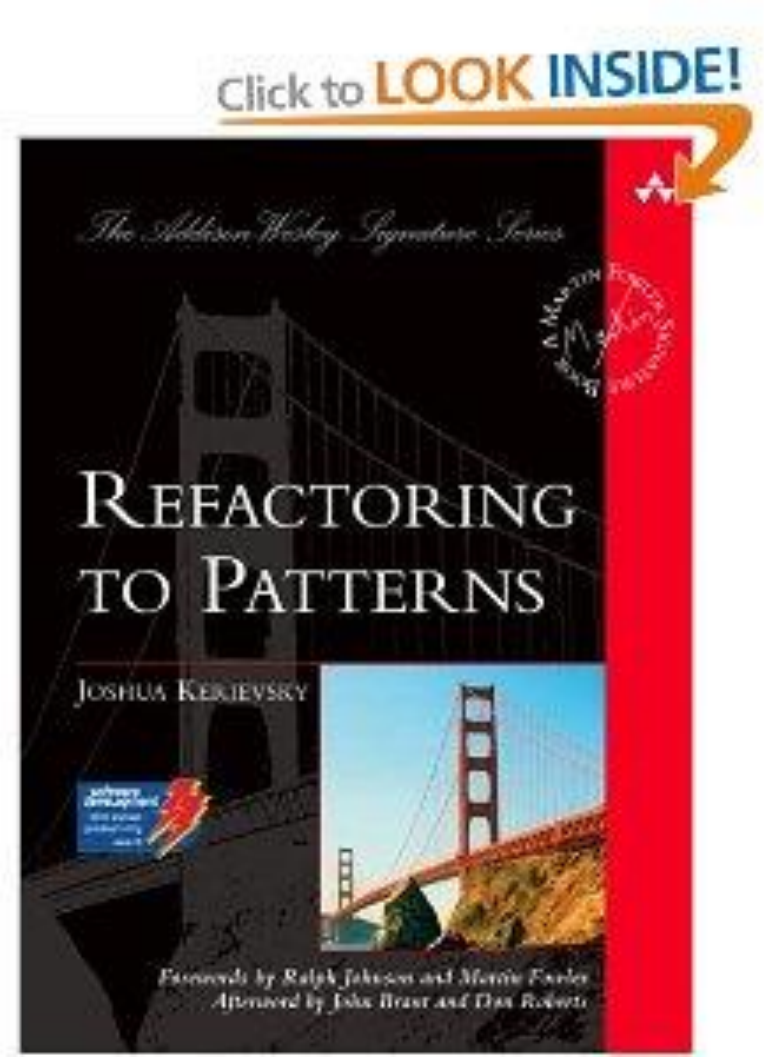
Joshua Kerievksy identified patterns as guidance for how we refactor.

We don't try to implement patterns in the SUT – the test focuses on requirements and is likely aimed at defining a contract.

We implement patterns as improvements to the code in the refactoring step.

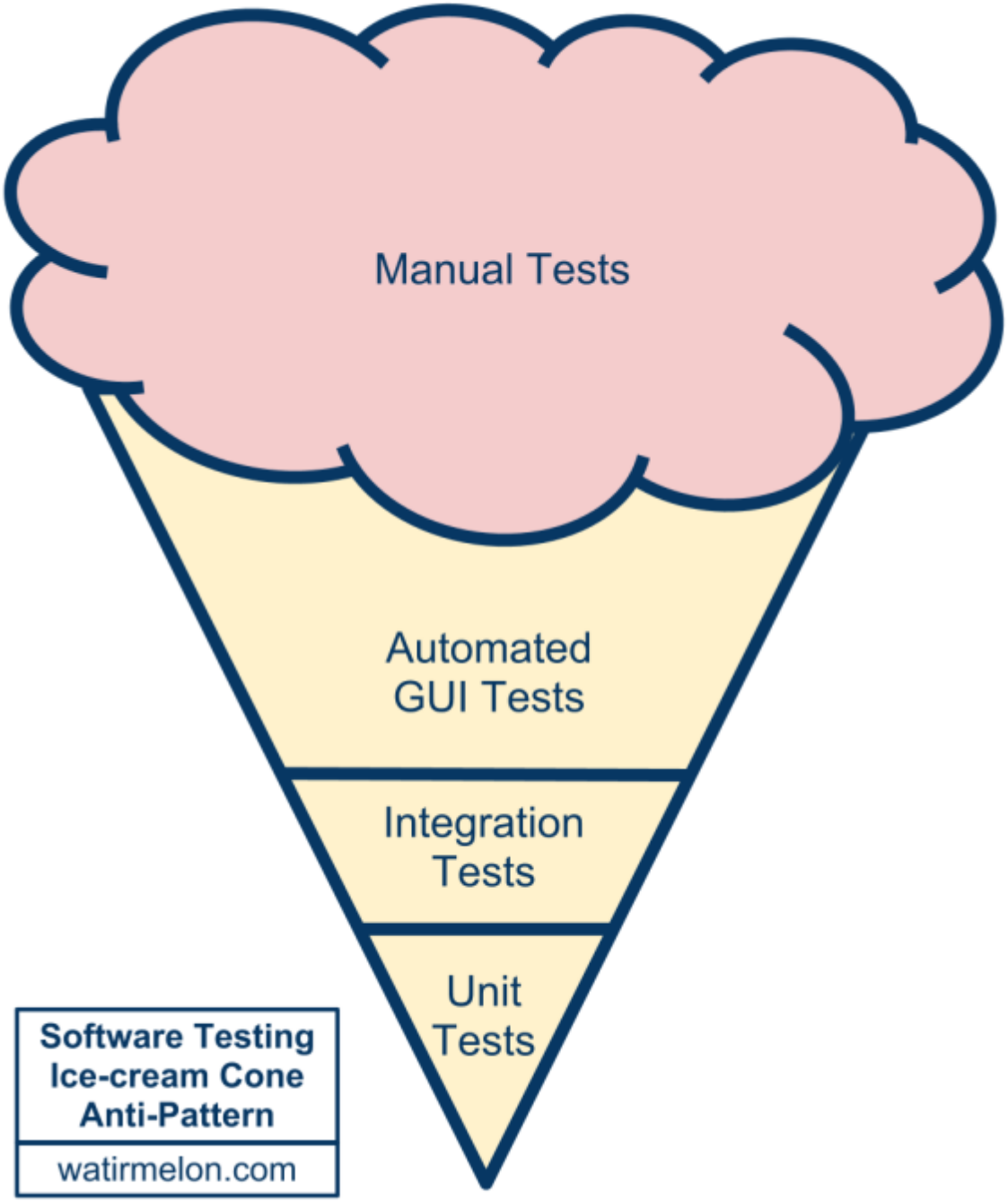
This prevents folks becoming 'pattern happy'. Patterns emerge as the solutions to refactoring sinful code into clean code.

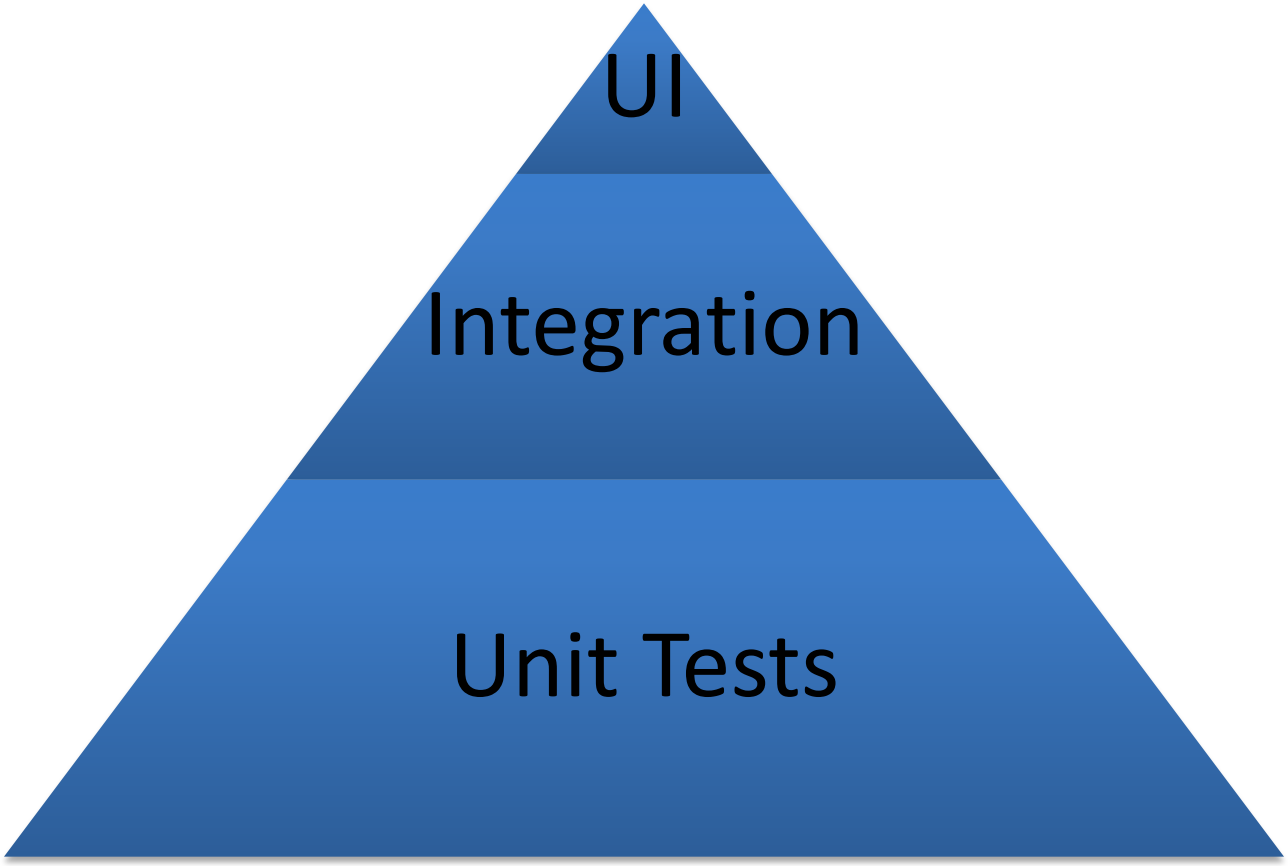
What would Jesus do? He'd sin then ask the god of patterns for forgiveness.



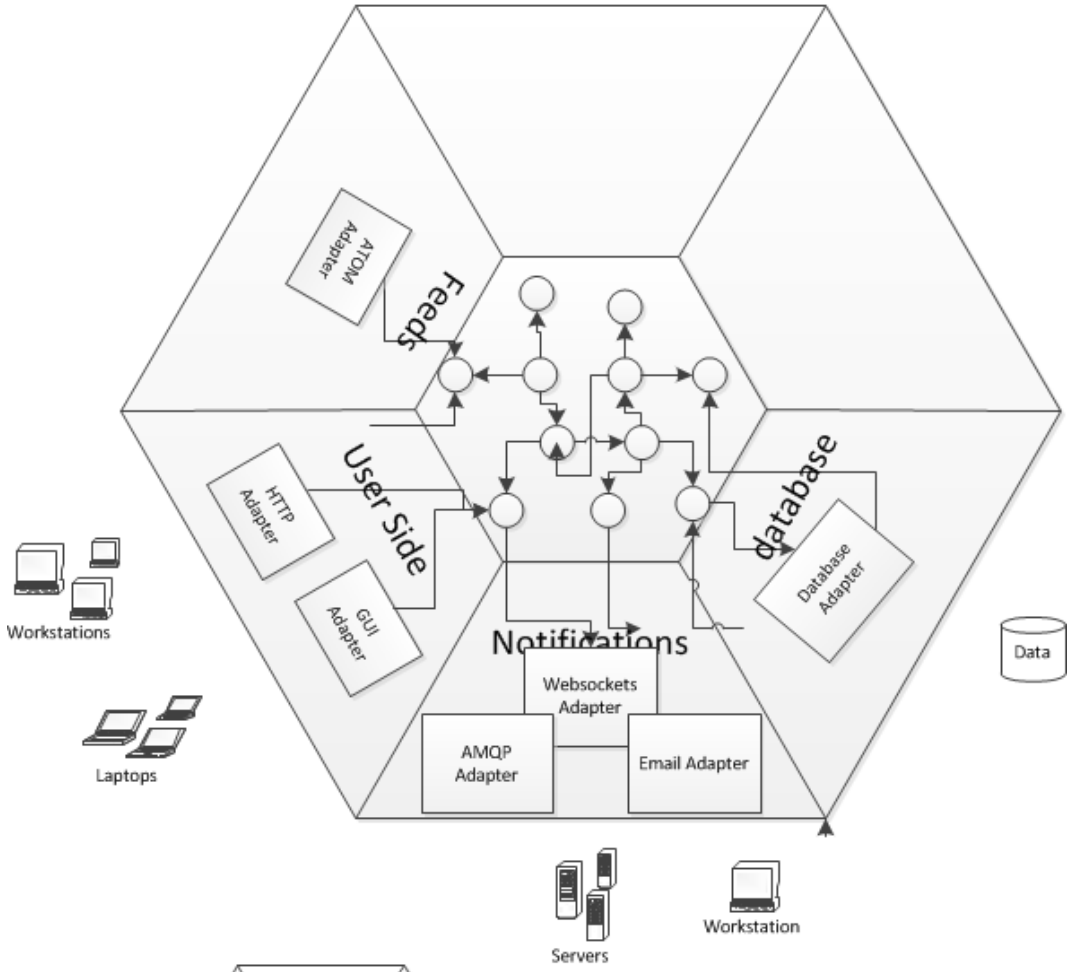
Hexagonal Architecture

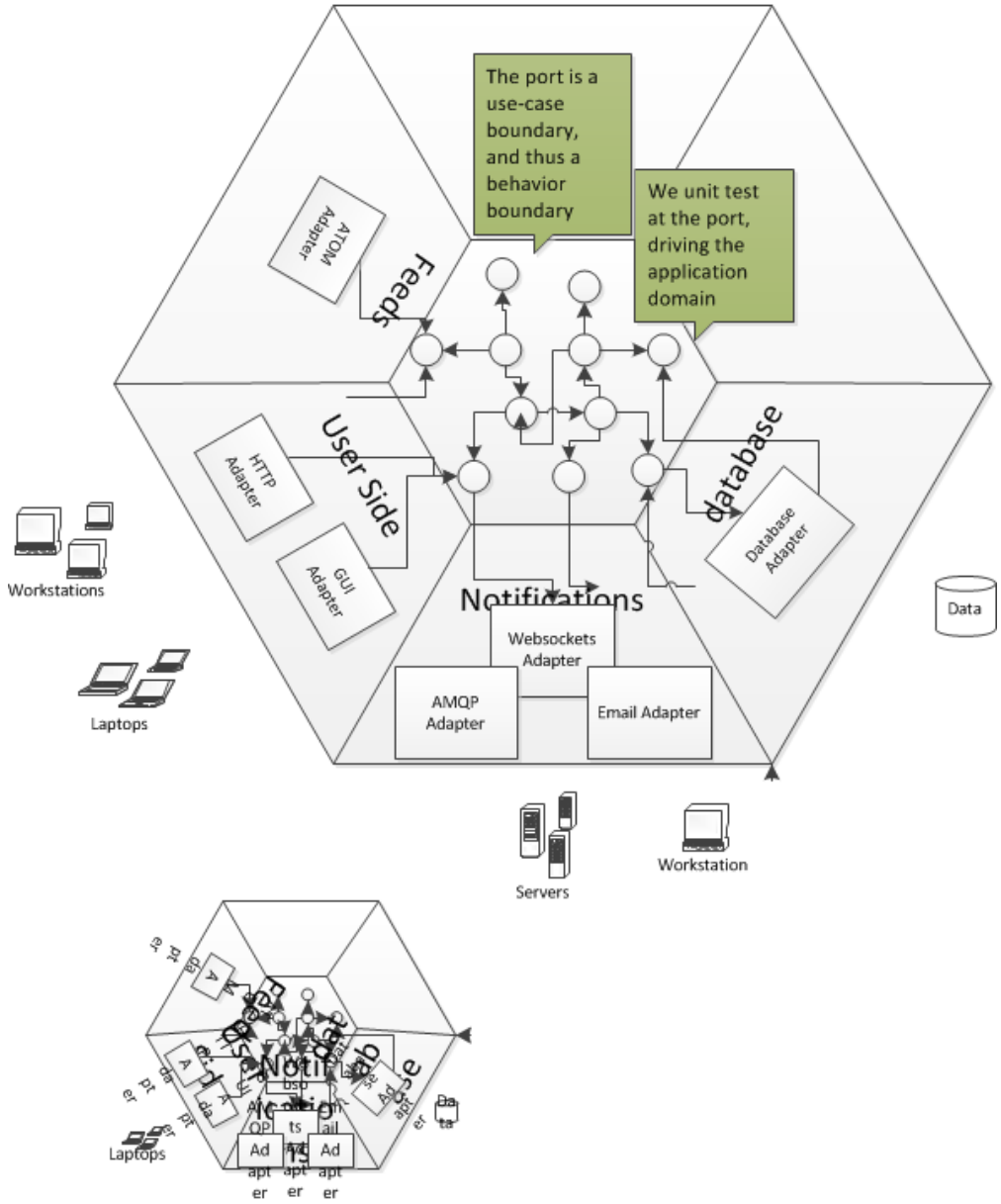
PORTS AND ADAPTERS

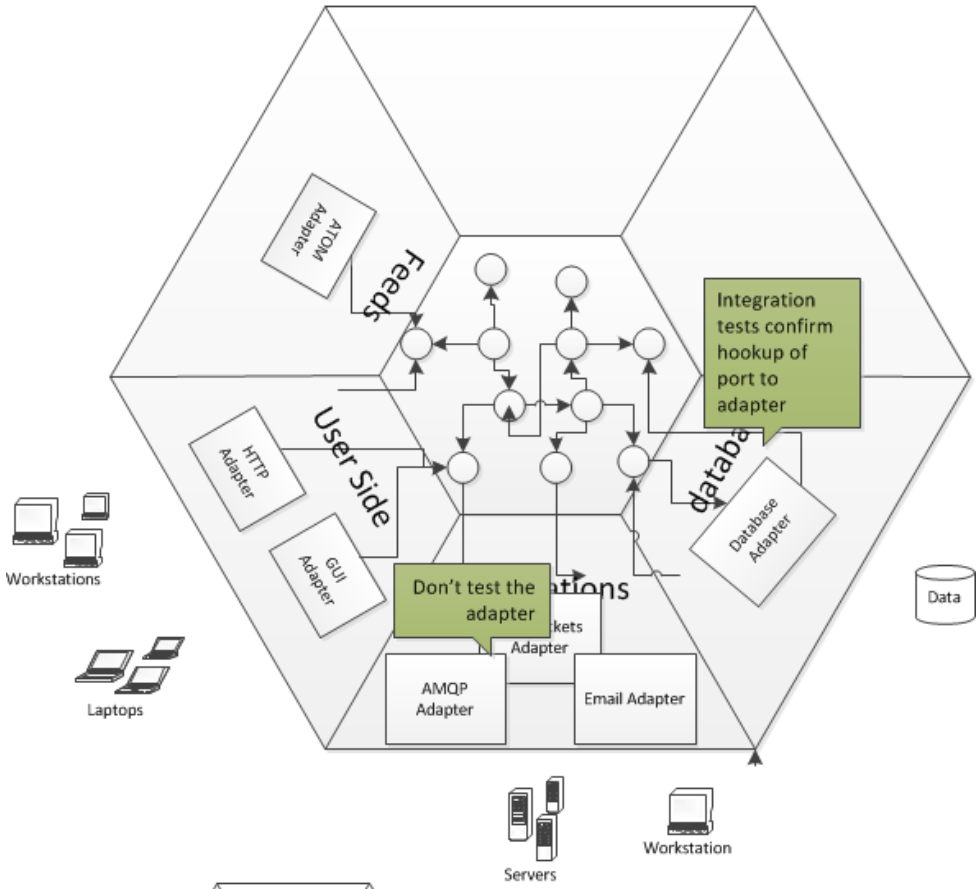

Manual Tests
Automated
GUI Tests
Integration
Tests
Unit
Tests
Software Testing
Ice-cream Cone
Anti-Pattern
watirmelon.com

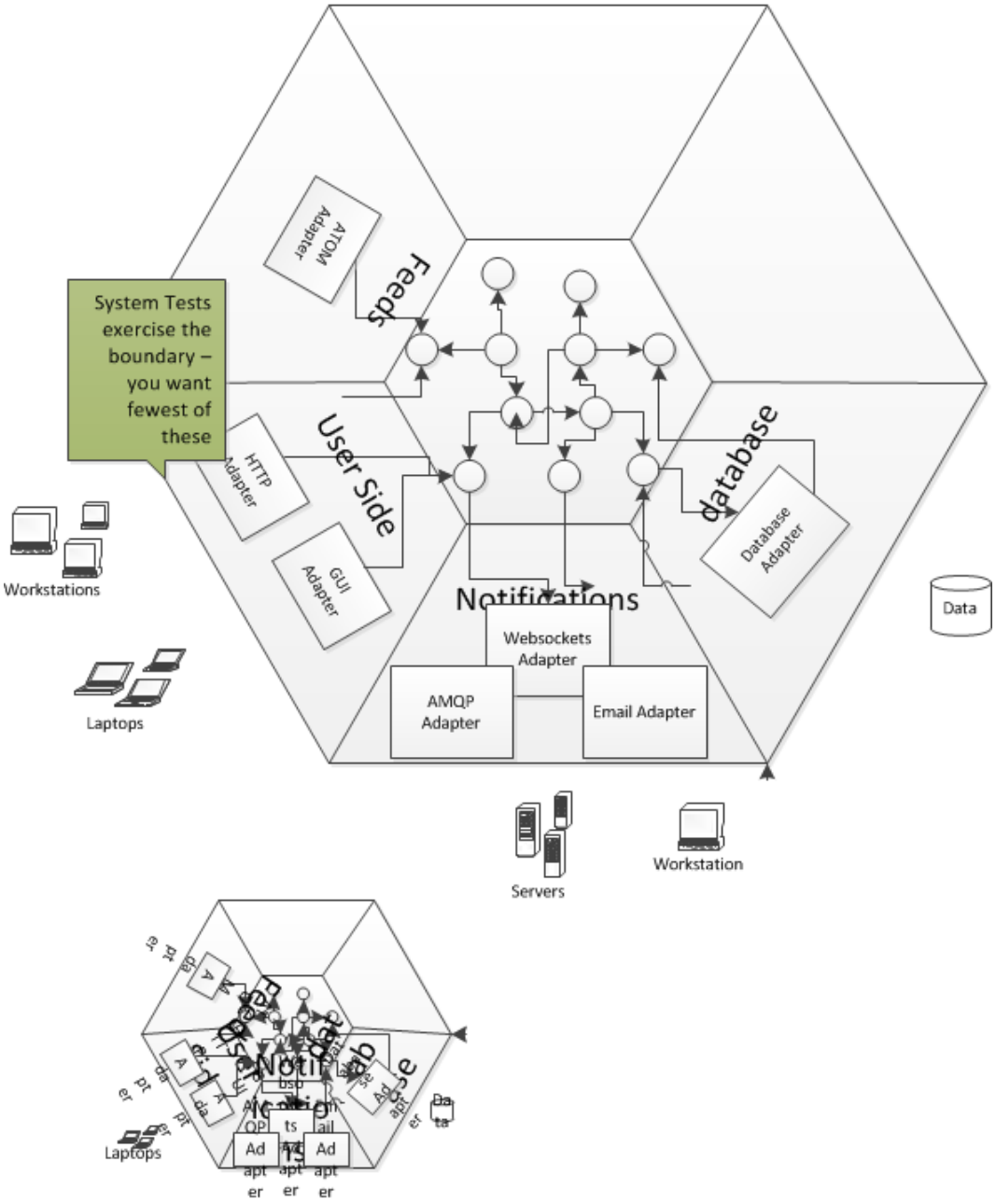


After Mike Cohn, Martin Fowler et al.


Feeds
ATOM Adapter
User Side
HTTP Adapter
GUI Adapter
database
Database Adapter
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Data
Servers
Workstation


The port is a use-case boundary, and thus a behavior boundary
We unit test at the port, driving the application domain
ATOM Adapter
Feeds
HTTP Adapter
User Side
GUI Adapter
database
Database Adapter
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Data
Servers
Workstation


Feeds
ATOM Adapter
User Side
HTTP Adapter
GUI Adapter
Integration tests confirm hookup of port to adapter
Database Adapter
Data
Don't test the adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Servers
Workstation


System Tests exercise the boundary – you want fewest of these
ATOM Adapter
Feeds
HTTP Adapter
User Side
GUI Adapter
database
Database Adapter
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Workstations
Laptops
Data
Servers
Workstation

Driving with a shift stick

GEARS


OFF
1
3
5
2
4
6

ACCEPTANCE TEST-DRIVEN DEVELOPMENT

- I’ve changed my mind on this over time.
- I used to believe in the tools, I believed that having to pass those tests kept developers honest, they could not declare done when they felt like it. Gojko Adzic quoted me on that too.
- But I think now the problem was how they wrote their unit tests – we were compensating by introducing ATDD tools to make them hit the bar. Instead, we should have been fixing the problem at source – write unit tests that focus on behaviors and thus can be used for acceptance.
- If you need to surface the tests to customers, process the tests a la MSpec or SpecUnit etc .to allow customer to read
- Hire QAs who are SDTs. Have them engage with the test code

BEHAVIOR DRIVEN DEVELOPMENT

“BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.”

Dan North, 2009 Agile specifications, BDD and Testing eXchange

- BDD starts with a similar insight – that we misunderstood TDD
  - It creates tools like RSpec and Jbehave
  - It realizes that specifying scenarios is the key to driving test-driven development
  - It evolves into a methodology for facilitating the transmission of requirements from customer to developer through scenarios that can be automated

This presentation is about ‘fixing’ TDD, but if you believe in the story here – you might want to take that narrative further by looking into BDD.

- However, I wanted to get back to basics here – and discuss what is wrong with TDD as learned by many, as I see even people claiming to practice BDD writing poor ‘unit tests’

Removing shard fixture

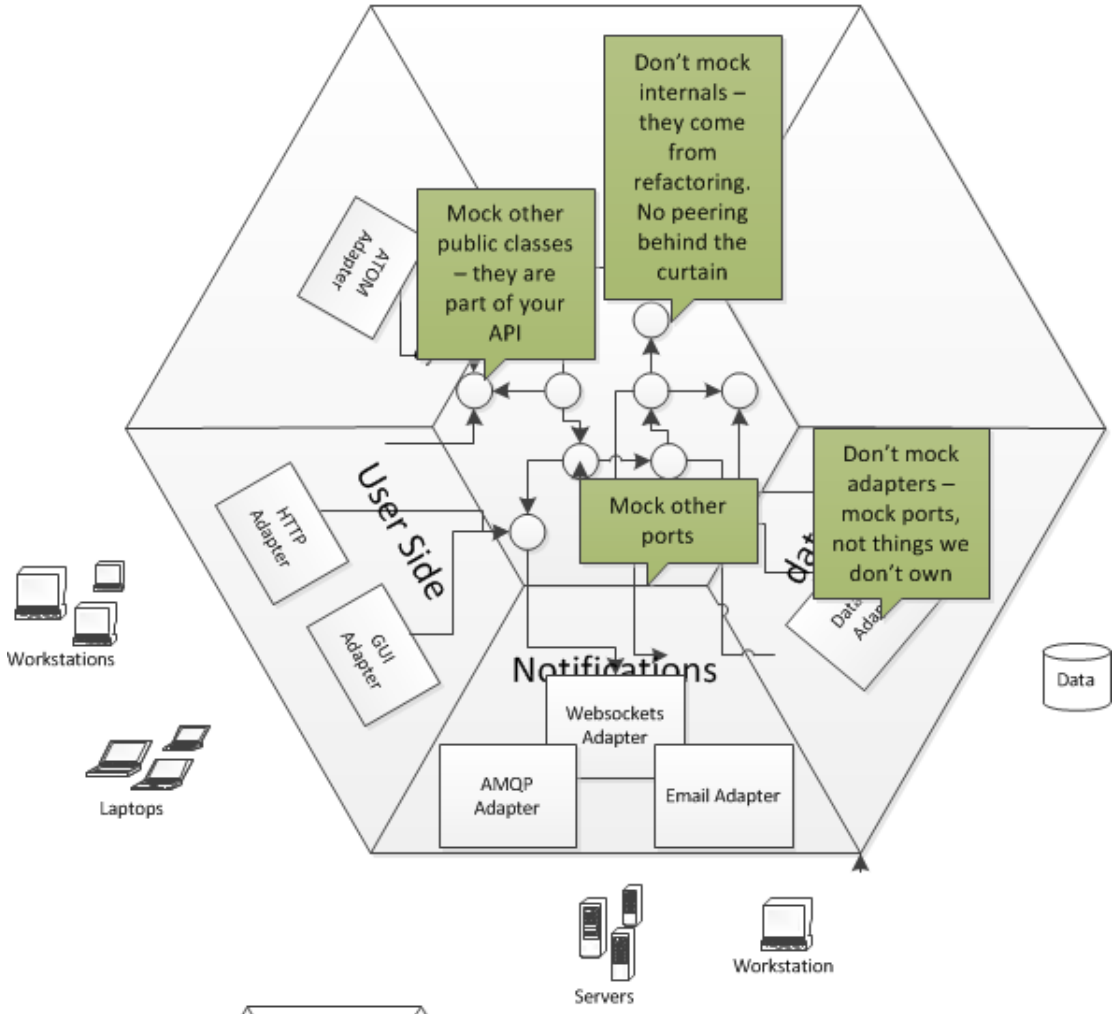
MOCKS

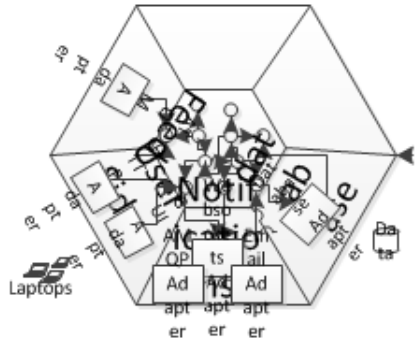
Mock Object

- Classic example is database
  - Long time to start
  - Difficult to keep ‘clean’
  - Tie tests to physical location on network
  - Write tests against something that acts like Db
- Mocks can improve readability by making contents of read rows explicit (evident data)
- Mocks hinder the use of singletons
  - Which may be a good thing
- Mocks make us consider issues around coupling
  - We learn to inject dependencies
- Mocks risk that the real system will not perform in the correct way

Forces

- Two forces:
  - We want to remove ‘hard-to-test’ dependencies
    - Database, UI, network, file system, configuration files etc.
    - Depending on these makes our test fragile
  - We want to use Responsibility Driven Design and mock collaborators
    - Tell Don’t Ask
    - But don’t mock internals


Don't mock internals – they come from refactoring. No peering behind the curtain
Mock other public classes – they are part of your API
ATOM Adapter
HTTP Adapter
User Side
GUI Adapter
Mock other ports
Don't mock adapters – mock ports, not things we don't own
Notifications
Websockets Adapter
AMQP Adapter
Email Adapter
Data
Workstations
Laptops
Servers
Workstation



Test Data and Evident Test patterns

OBJECT MOTHER AND TEST DATA BUILDER

Object Construction

- Tests do a lot more setup of data than normal code
  - They don't use repositories etc
- That makes tests very noisy and leads to long setup methods.
  - The Test Data has a low signal to noise ratio.
  - We have a DRY issue. Changing our domain can ripple out across the test codebase forcing us to change a lot of tests

Object Mother

- An early solution was object mother
  - A class that contains static factory methods used to create objects for use in tests
  - The name of the method helps identify the stereotypical object being created
  - The conceit was that developers would become familiar with the objects and use them in scenarios

```
Invoice invoice1 =
   TestInvoices.newDeerstalkerAndCapeAndSwordstickInvoice();
Invoice invoice2 =
   TestInvoices.newDeerstalkerAndBootsInvoice();
```

Problems with Object Mother

- Variations in testing mean we end up with many different factory methods
  - Object Mother soon becomes bloated
- The variables affecting the test are not evident, but hidden behind the creation mechanism
  - If you don't know the Object Mother test data, tests are obscure
- The factory methods become shared fixture, which means that changes to construction ripple out

Test Data Builders

- A solution is to use the Builder Pattern. For each class you want to use in a test, create a Builder for that class that:
  - Has an instance variable for each constructor parameter
  - Initialises its instance variables to commonly used or safe values
  - Has a `build` method that creates a new object using the values in its instance variables
    - We can do a neat trick with conversion operators too
  - Has "chainable" public methods for overriding the values in its instance variables.

```
public class InvoiceTestDataBuilder {
   Recipient recipient = new RecipientTestDataBuilder().build();
   InvoiceLines lines = new InvoiceLines(new InvoiceLineTestDataBuilder().build());
   PoundsShillingsPence discount = PoundsShillingsPence.ZERO;

   public InvoiceBuilder WithRecipient(Recipient recipient) {
      this.recipient = recipient;
      return this;
   }

   public InvoiceBuilder WithInvoiceLines(InvoiceLines lines) {
      this.lines = lines;
      return this;
   }

   public InvoiceBuilder WithDiscount(PoundsShillingsPence discount) {
      this.discount = discount;
      return this;
   }

   public Invoice Build() {
      return new Invoice(recipient, lines, discount);
   }
}
```

Builders and Evident Data

- The use of the fluent interface – the withFoo() syntax helps highlight data affecting the test
  - To create the fluent interface, return the Builder itself by using *return this;* in the WithFoo() method
  - You can also use a specification, to allow you to separate the data used to construct instances from the builder
    - MyObjectBuilder(IAmASpecification mySpec)
  - Allows repeat creation of objects, variation between to be highlighted
  - Also allows default parameters on the specification

In case you slept through it

SUMMARY

- The reason to test is a new behavior, not a method on a class
- Write dirty code to get green, then refactor
- No new tests for refactored internals and privates (methods, classes)
- Both Develop and Accept against tests written on a port
- Add Integration tests for coverage of ports to adapters
- Add system tests for end-to-end confidence
- Don’t mock internals, privates, or adapters

If in doubt just ask

Q&A