



**ICON**  
**AGILITY SERVICES**

# Test-Driven Development Foundations: Scala v1.0

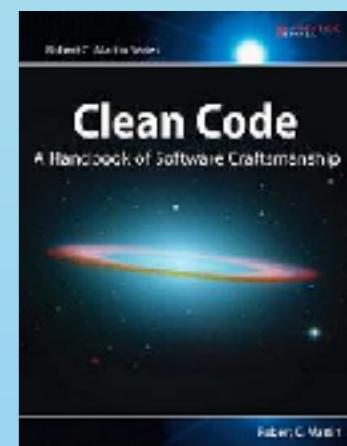
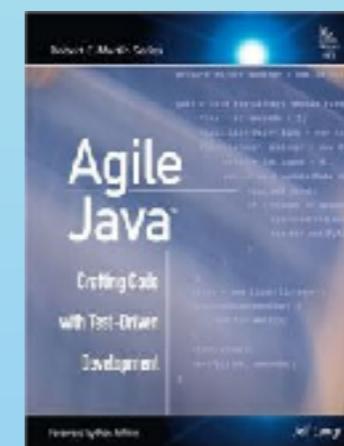
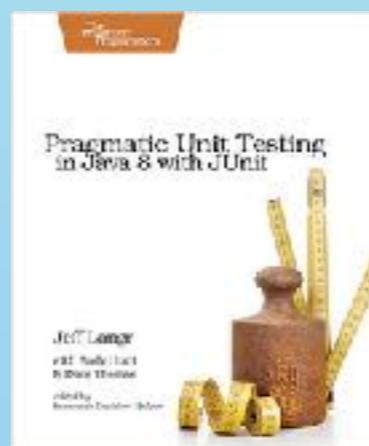
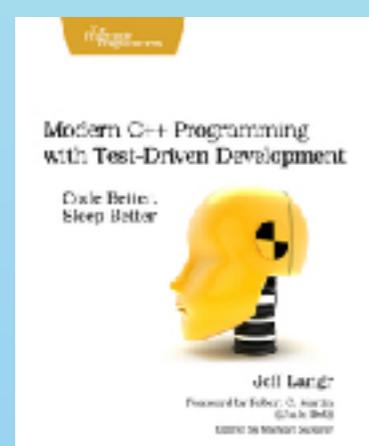
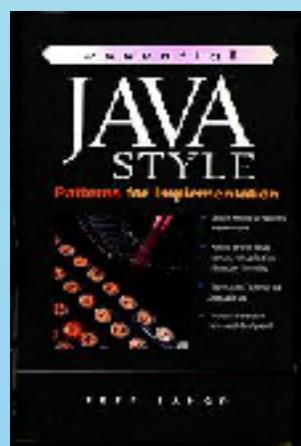
# Jeff Langr

[jeff@langrsoft.com](mailto:jeff@langrsoft.com)  
@JLangr



Langr  
*Software Solutions*

SOFTWARE TRAINING/CONSULTING



# What's a Unit Test Look Like?

```
describe("something") {  
    it("demonstrates some behavior") {  
        // Arrange  
  
        // Act  
  
        // Assert  
    }  
}
```

The AAA mnemonic is courtesy of Bill Wake.

# A Sample Unit Test

```
describe("an auto") {  
    val auto = Auto()  
  
    it("idles engine when started") {  
        auto.depressBrake()  
  
        auto.pressStartButton()  
  
        auto.RPM() shouldBe 1000 +- 50  
    }  
}
```

# What's an Assertion Look Like?

Some fundamentals:

```
someCondition shouldBe true  
someCondition should be (true)  
idleSpeed shouldEqual 1000  
idleSpeed should not equal 2000  
idleSpeed should equal (1000)  
idleSpeed should not be < (900)  
alphabetizedName shouldEqual "Schmoo, Kelly Loo"  
alphabetizedName should startWith regex ("S.*,")
```

See [http://www.scalatest.org/user\\_guide/using\\_matchers](http://www.scalatest.org/user_guide/using_matchers)

# What's an Assertion Look Like?

More interesting stuff:

```
someList should contain('0')
someList should contain theSameElementsInOrderAs
  Seq('0', 'T', 'T', 'F', 'F', 'S', 'S')
someList should contain inOrder('0', 'T', 'S')
someList should not be empty
someList should have length 7
forAll (someList) { c: Char => c.isLetter should be (true)}
someOption shouldBe empty
```

*You can also create custom matchers!*

See [http://www.scalatest.org/user\\_guide/using\\_matchers](http://www.scalatest.org/user_guide/using_matchers)

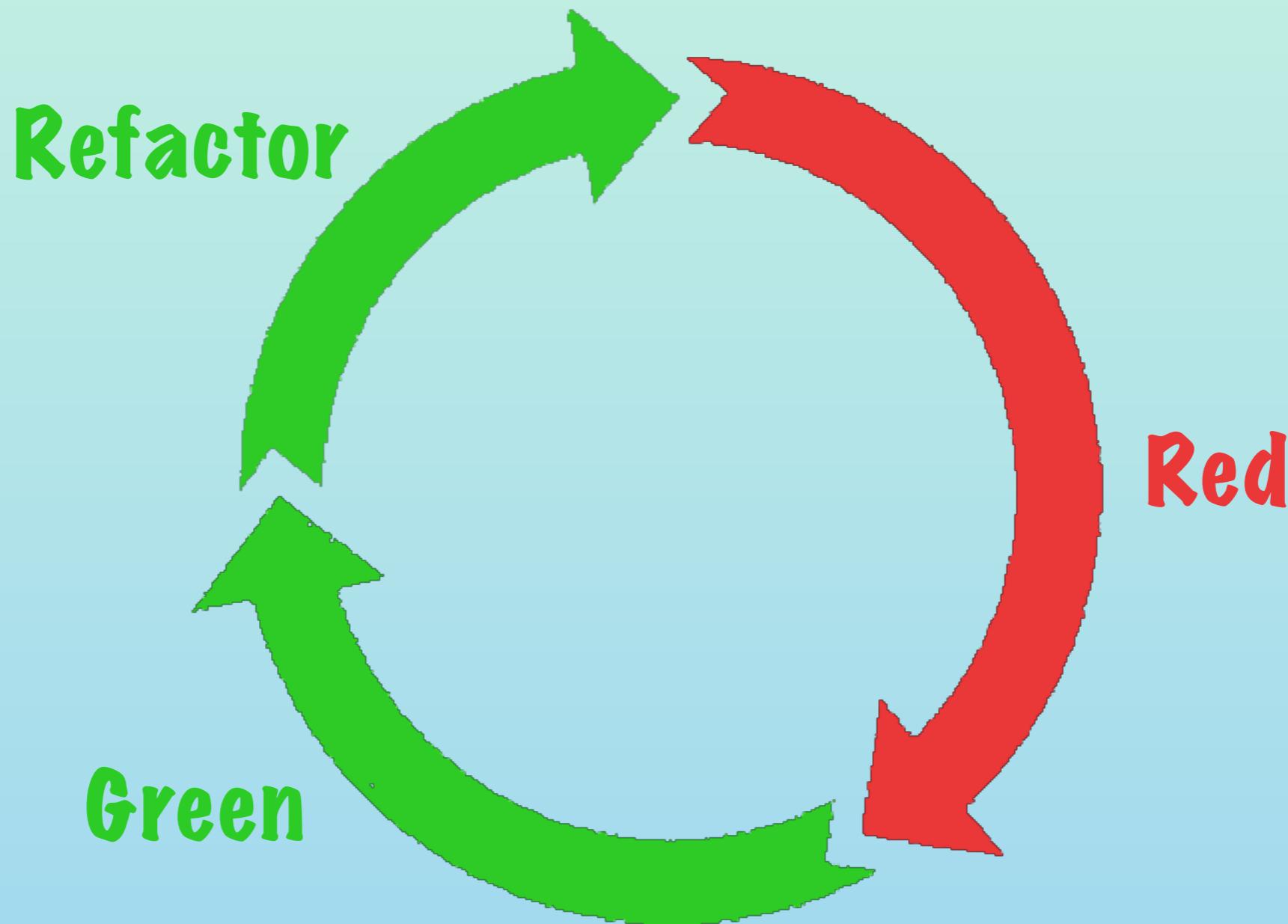
# Exercise: Simple Outcomes

---



Flesh out the tests in `com.langrsoft.util.BasicsTest`

# Test-Driven Development



*An incremental design technique*

# **Exercise: (Test)-Code-Refactor**

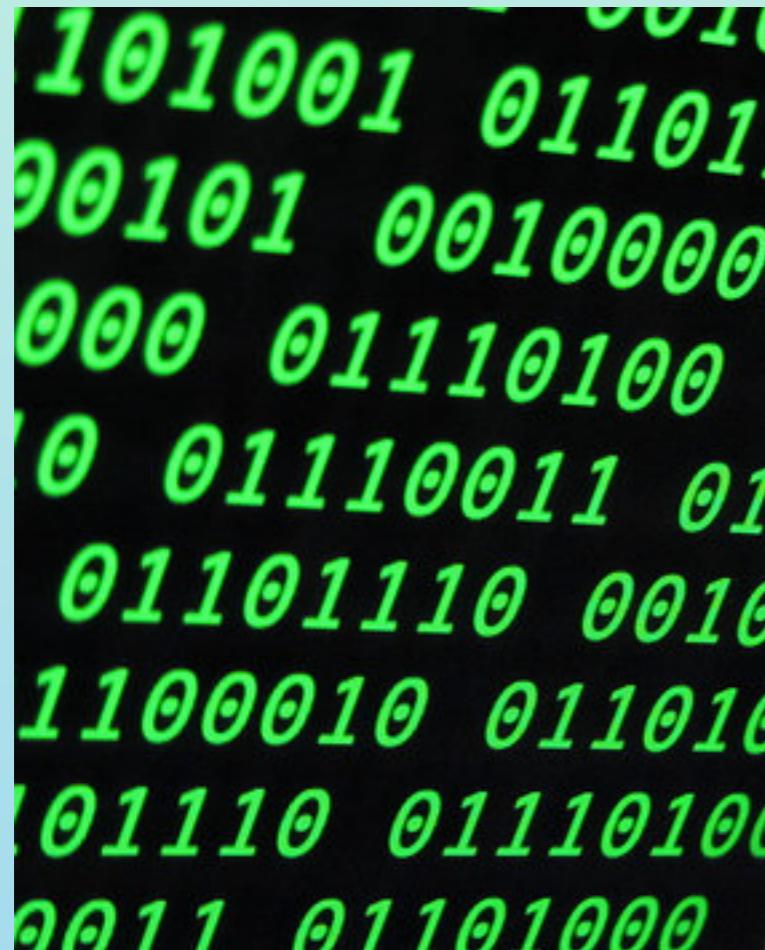
## TDD Paint by Numbers

---



See `com.langrsoft.util.NameNormalizerTest`

# Core TDD Themes



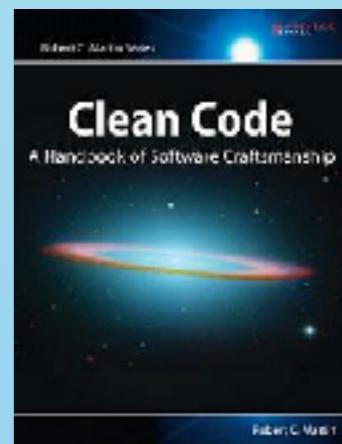
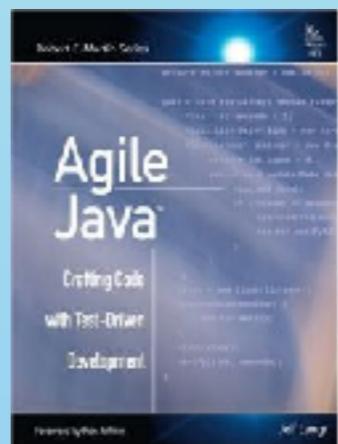
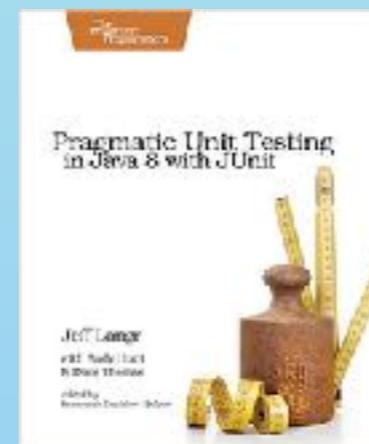
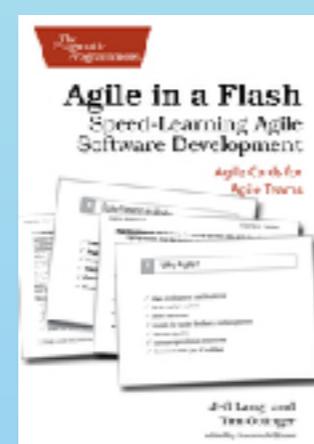
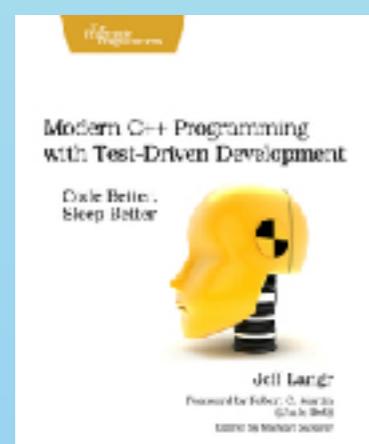
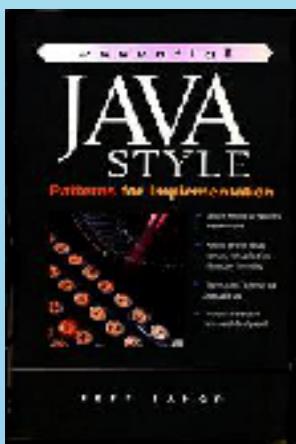
- Drive *behavior*
- Specification by example
- Small increments
- Stick to the cycle
  - Always see red
  - Always refactor

***It's just code!***

# Stepping Back: Writing Tests

Langr  
Software Solutions

SOFTWARE TRAINING/CONSULTING



# Recent File List

---

Create a list of recently-opened files  
(max: 5) for use in a GUI application.



What tests do we need?

# ZOMBIES!

- Zero
- One
- Many
- Boundaries
- Interface definition
- Exceptional behavior
- Simple Solutions & Scenarios



<http://blog.wingman-sw.com/archives/677>

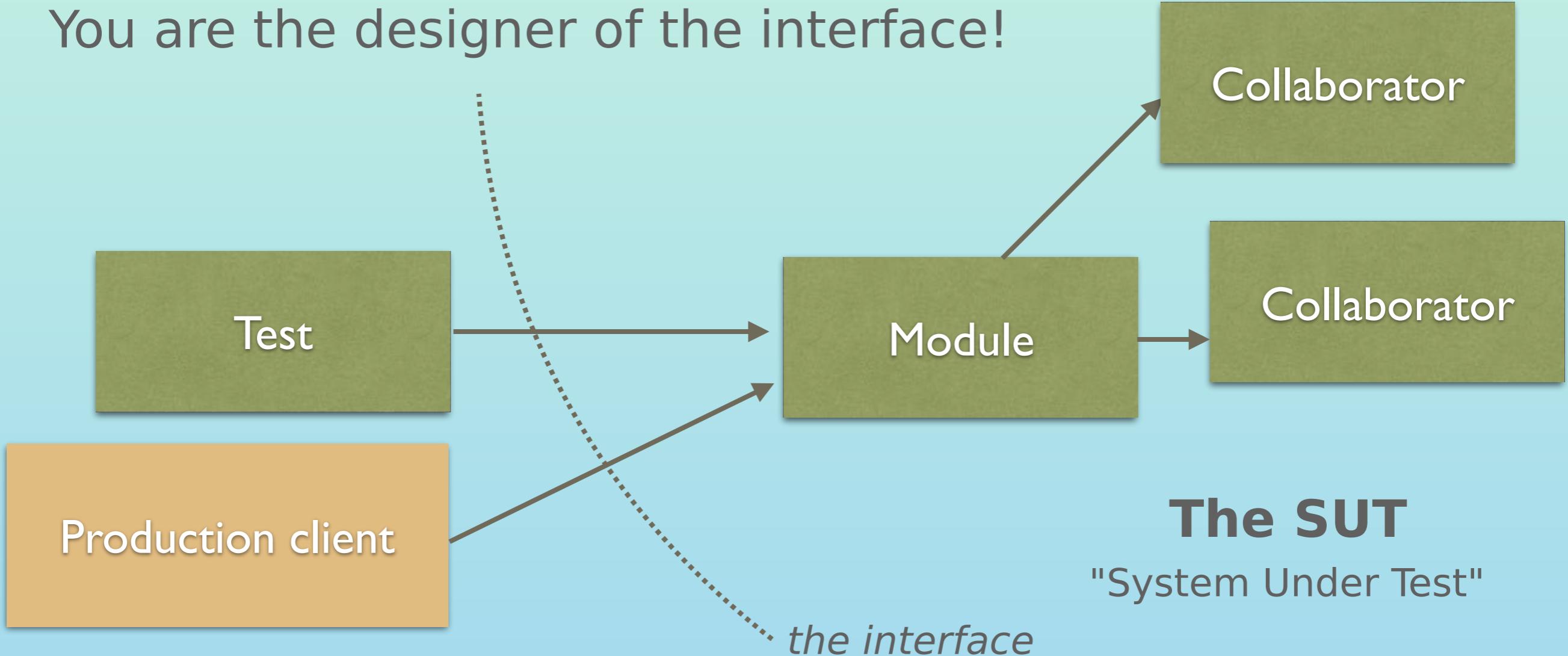
# Test Lists

---

- **Z:** No files opened
- **O:** One file opened
- **M:** Multiple files opened
- **B:** One more than capacity opened
- **E:** File opened twice

# Your Test Is the First Client

You are the designer of the interface!



# Structuring the Test

*"shifts duplicate filename to top of recently-used list"*

■ <b>Arrange</b>	Add file with name <code>opened-A.txt</code> Add file with name <code>opened-B.txt</code> Add file with name <code>opened-C.txt</code>
■ <b>Act</b>	Add file with name <code>opened-A.txt</code>
■ <b>Assert</b>	Expect recently-used list to equal: <code>opened-A.txt</code> <code>opened-C.txt</code> <code>opened-B.txt</code>

# Assert-First

---

```
it("should sift duplicate filename to top") {  
    // ?  
  
    recentlyUsedList.orderedFilenames shouldEqual  
        Seq(duplicateOfAddedFirst, "3rd", "2nd")  
}
```

Consider working from the outcome.

# Assert-First

---

```
it("should sift duplicate filename to top") {  
    // ?  
  
    recentlyUsedList.add(duplicateOfAddedFirst)  
  
    recentlyUsedList.orderedFilenames shouldEqual  
        Seq(duplicateOfAddedFirst, "3rd", "2nd")  
}
```

# Assert-First

---

```
it("should sift duplicate filename to top") {  
    recentlyUsedList.add("1st")  
    recentlyUsedList.add("2nd")  
    recentlyUsedList.add("3rd")  
    val duplicateOfAddedFirst = "1st"  
  
    recentlyUsedList.add(duplicateOfAddedFirst)  
  
    recentlyUsedList.orderedFilenames shouldEqual  
        Seq(duplicateOfAddedFirst, "3rd", "2nd")  
}
```

# Exercise: Stock Portfolio Module

In-memory only

- is it empty or not?
- what is the count of unique symbols?
- given a symbol and # of shares, make a purchase
- how many shares exist for a given symbol?
- given a symbol and # of shares, sell the shares
- throw an exception when selling too many shares



"Tokyo Stock Exchange," courtesy <https://www.flickr.com/photos/31029865@N06/>  
License: <https://creativecommons.org/licenses/by/2.0/>

\*\*\* Remember! \*\*\*

**Red:** Ensure test fails

**Green:** Implement no more code than necessary

**Refactor:** Clarify and eliminate all duplication

	09:00	11:30	12:30	15:00
TDK	3430	+85	トヨタ自	2522
キーエンス	18480	-80	ホンダ	2362
デンソー	2115	+38	スズキ	1613
ファナック	11790	+340	ニコン	1726
ローム	3565	+45	HOYA	1639
京セラ	6260	+120	キヤノン	3465
村田製	3960	+15	リコー	671
日東電	2819	+9	凸版印	575
三菱重	328	+2	大日印	756
日産自	696	+4	任天堂	10560
食品		+0.83	電力・ガス	56.43
エネルギー資源	113.35	+1.53	運輸・物流	
建設・資材		+0.62	商社・卸売	
素材・化学	97.74	+0.62	小売	
医薬品		+0.37	銀行	

# A Functional Implementation

## [[ TODO ]]

Store portfolio info in a state object:

```
{  
  holdings:  
  {  
    IBM: 20,  
    BAYN: 10  
  }  
}
```

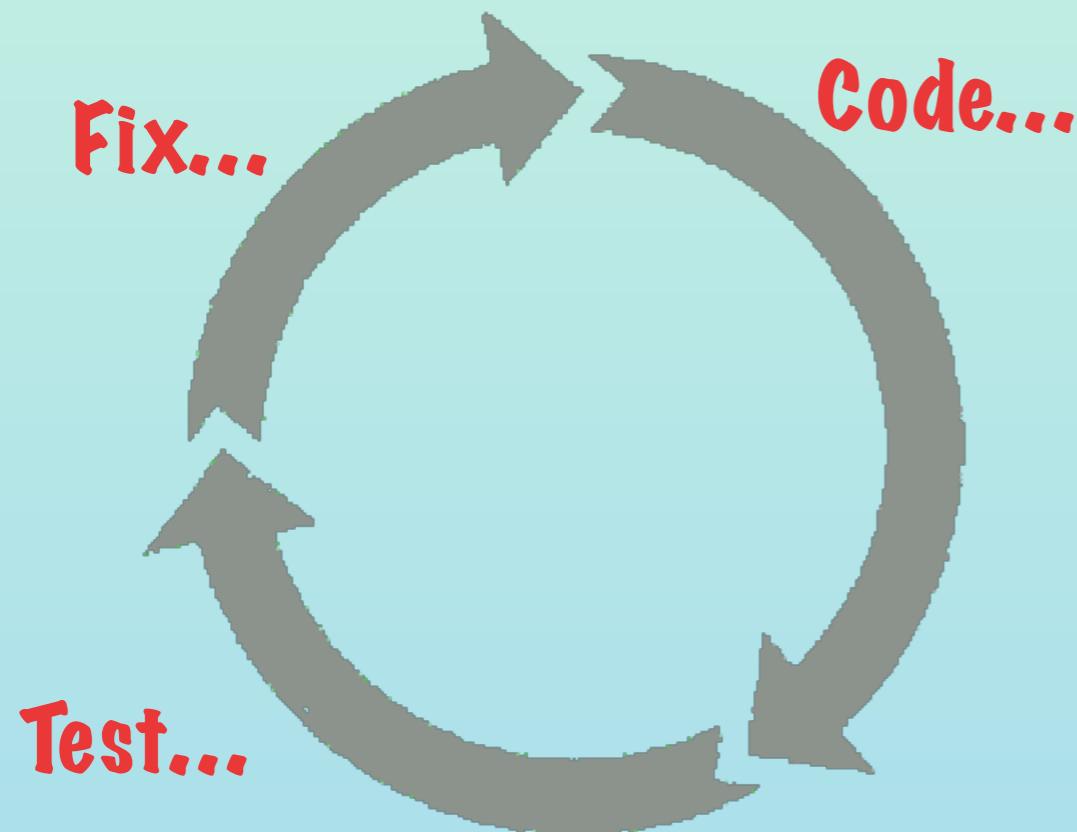
Pass state to each function, return updated state:

```
const portfolio = { holdings: [] }  
const newPortfolio = purchase(portfolio, 'BAYN', 42)
```

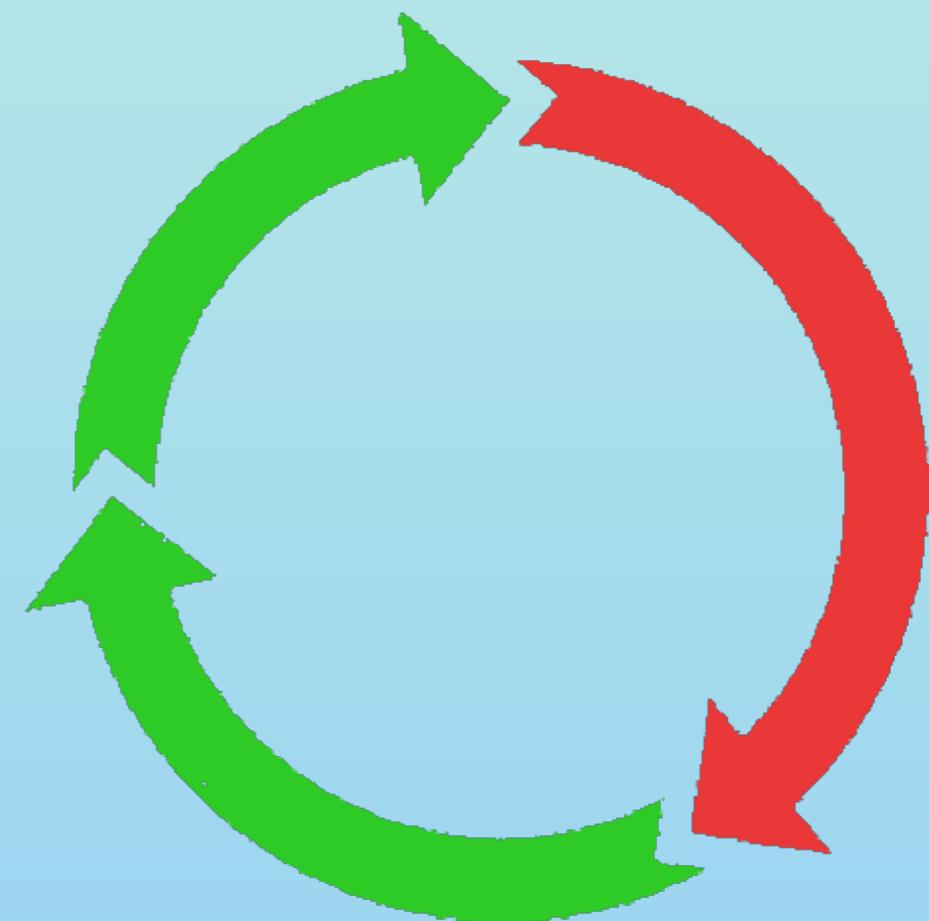
*Ensure your function creates no side-effects!*

I.e., **portfolio** should not change

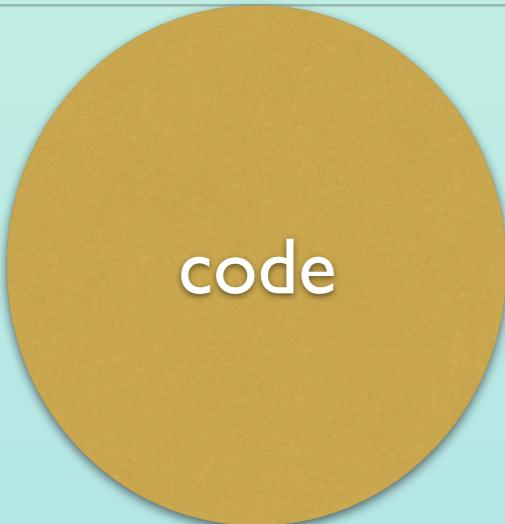
# TAD\*\* and TDD: What's the Difference?



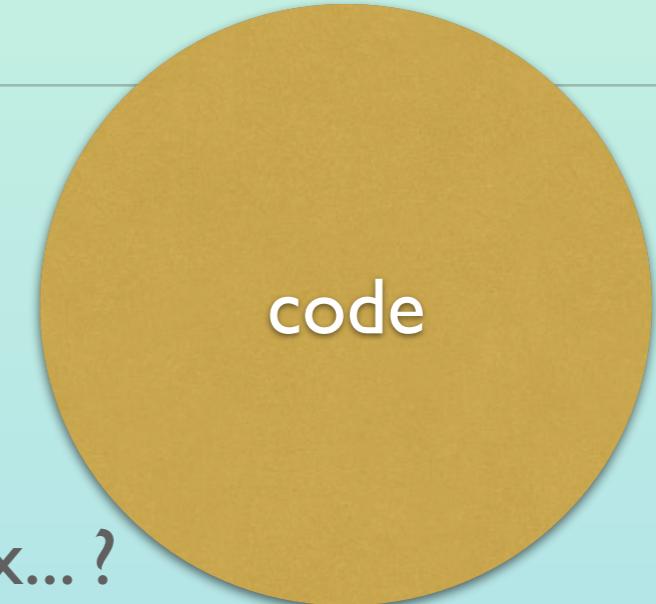
"Test-After Development"



# Small Increments



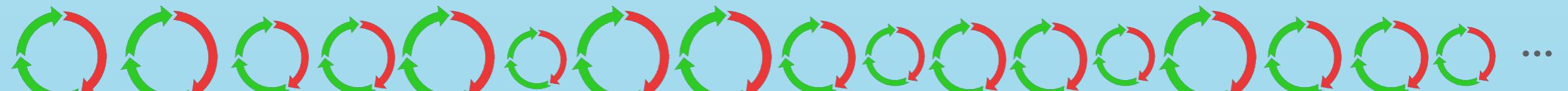
...fix... ?



...fix... ?

TAD

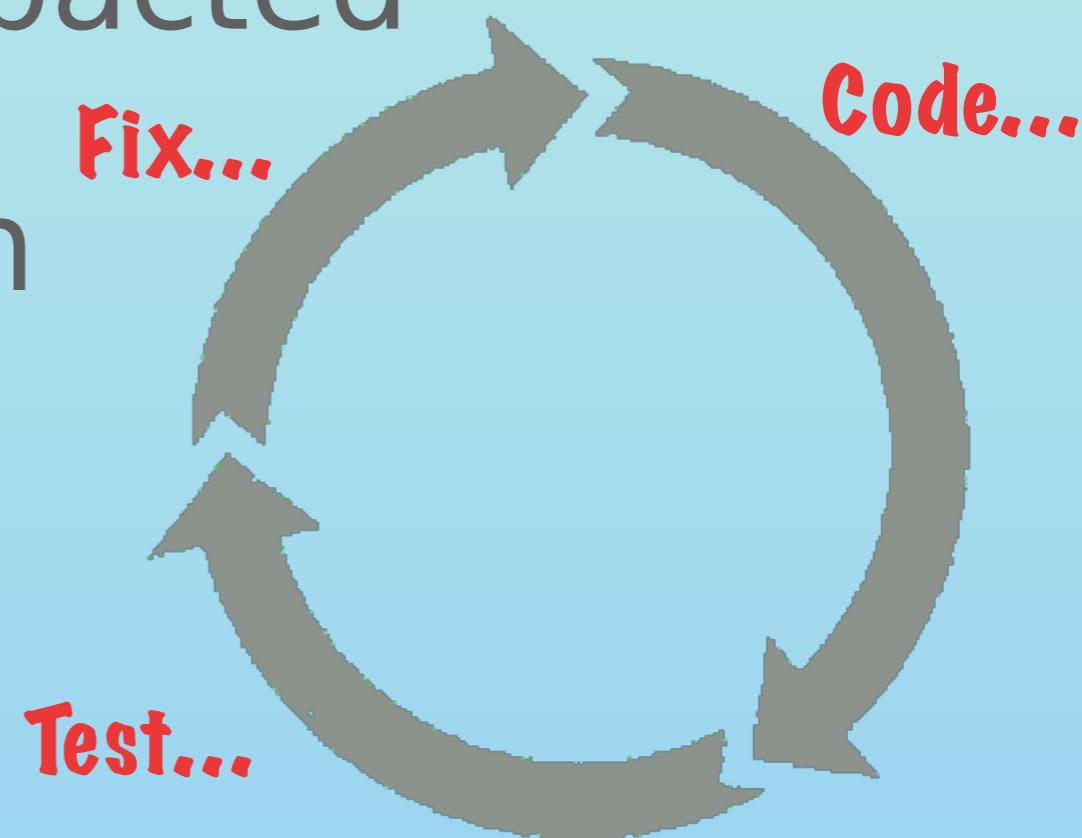
vs.



TDD

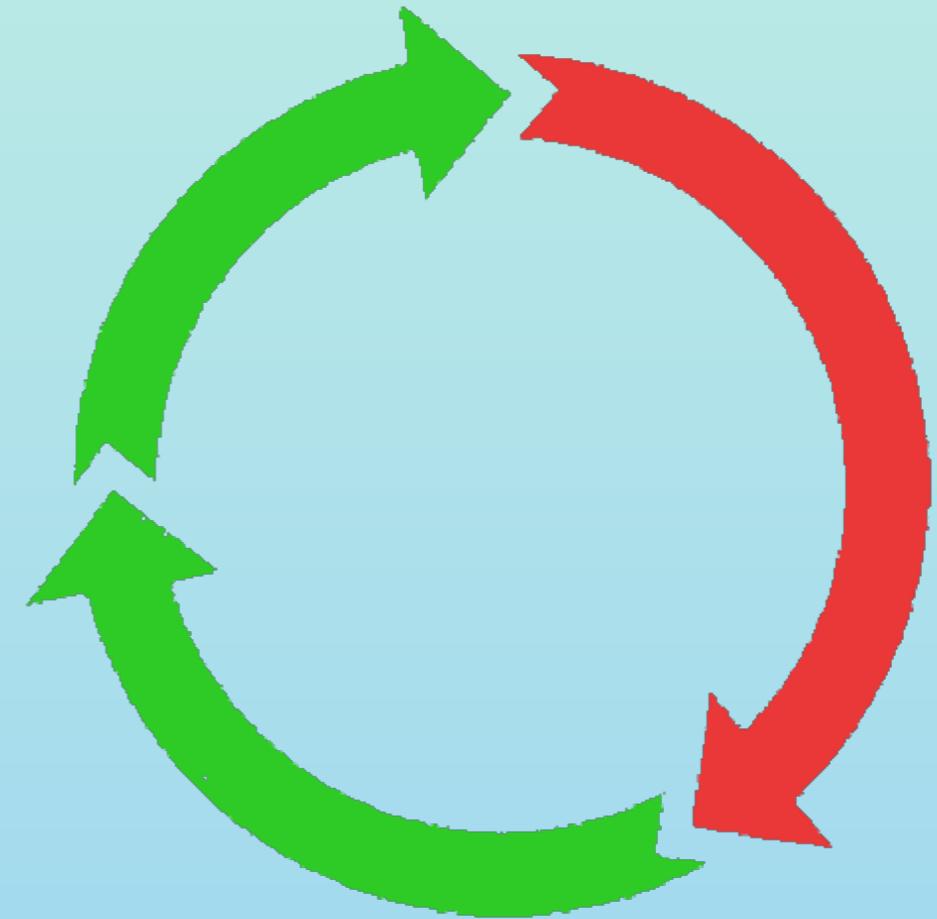
# Test-After Development (TAD)

- Some refactoring accommodated
- Coverage: ~70%
- Design not usually impacted
- Some defect reduction
- Separate task?

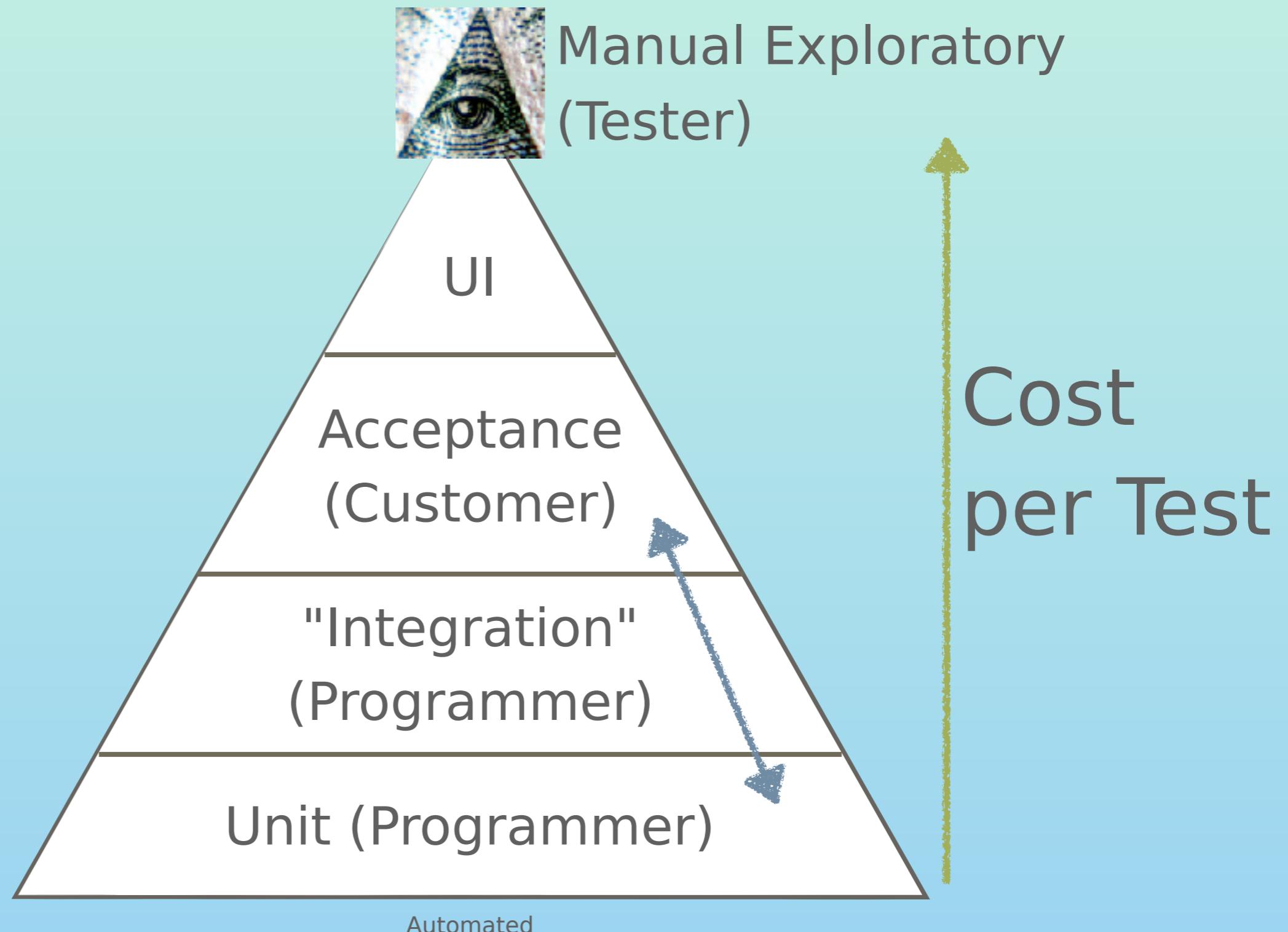


# Test-Driven Development (TDD)

- Continual refactoring
- Coverage for all intended features
- Incremental design shaping
- Significant defect reduction
- Minimized debugging
- Integral part of coding process
- Clarify / document needs / choices
- Continual forward progress
- Consistent pacing
- Continual feedback / learning
- Sustainable



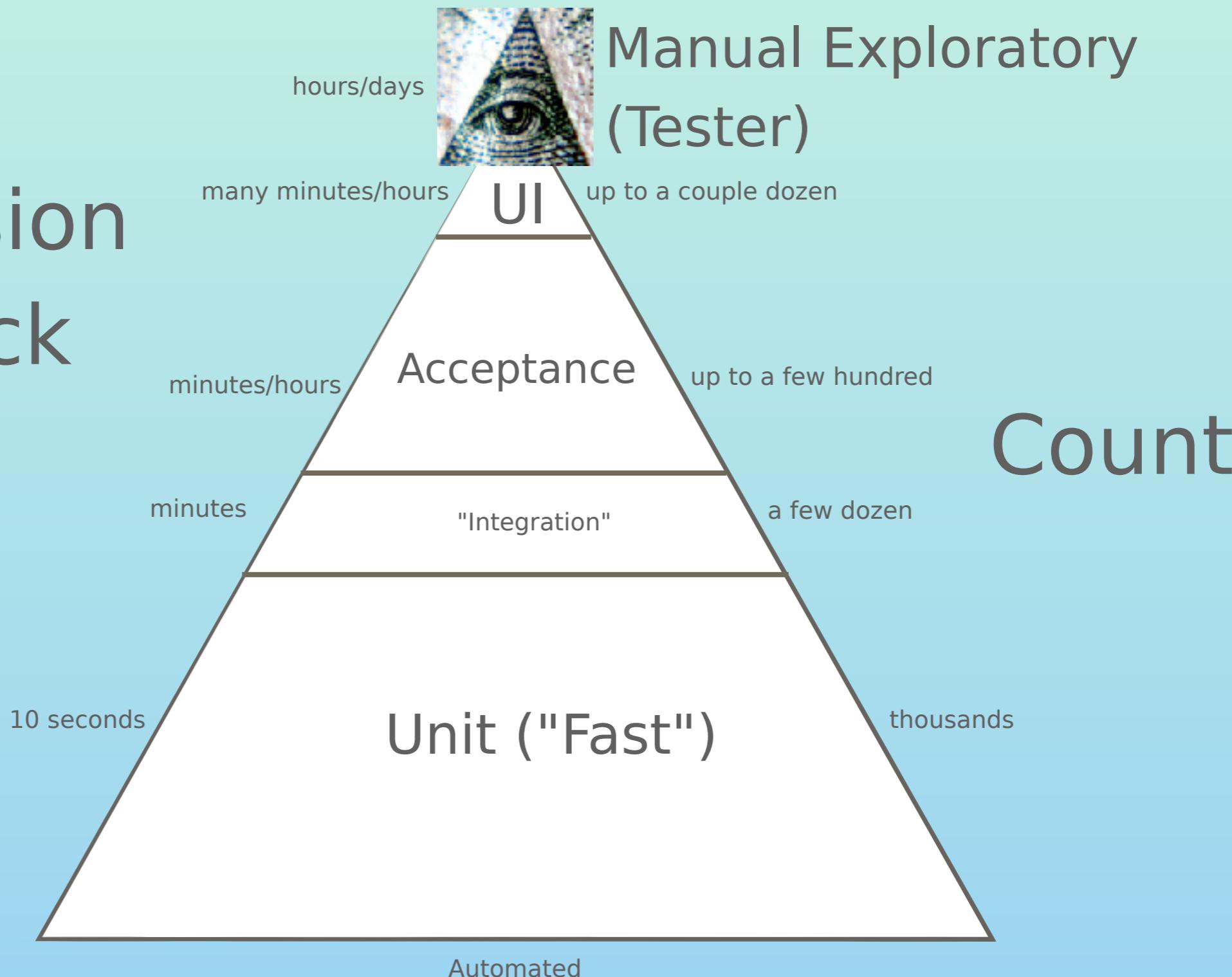
# Unit Testing: Insufficient!



# A Re-leveled Pyramid

Regression  
Feedback

Count



# Katas

---

<http://codekata.com>

## Other sites:

<https://exercism.io>

<http://cyber-dojo.org>

<https://sites.google.com/site/tddproblems/>

<https://www.codewars.com>

<https://github.com/jlangr/name-normalizer>

<https://github.com/emilybache>

<https://projecteuler.net/archives>



# Roman Numeral Converter

---

Given a positive integer  
from 1 up to 4000,  
answer its Roman equivalent



# Feedback Q's

■ :-)

What have I learned of value?

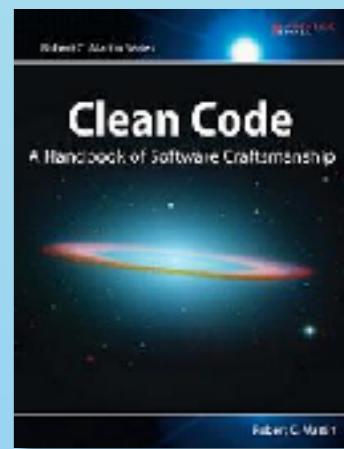
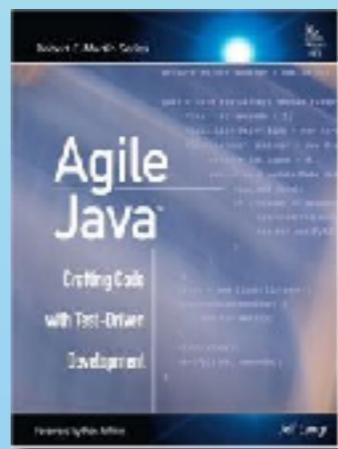
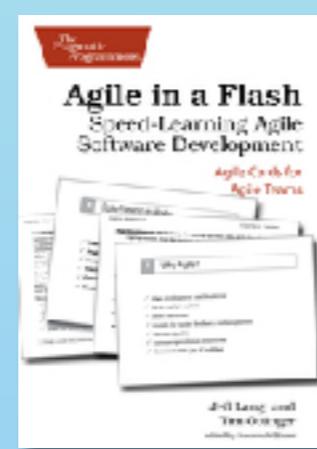
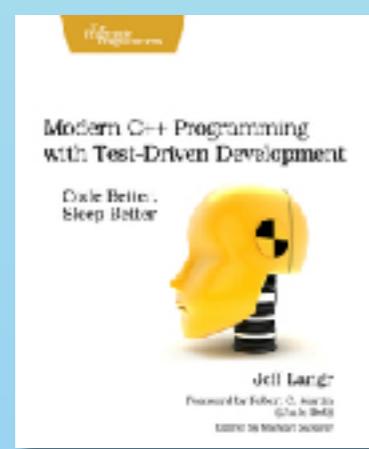
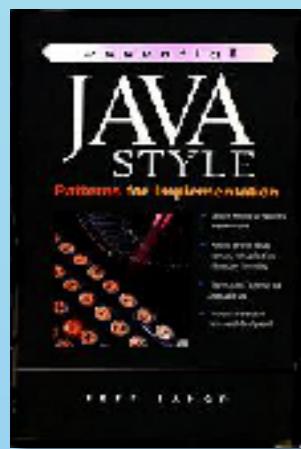
■ %-|

What, if anything, am I unclear about regarding the discussion / exercises?

■ ?:-/

What, if anything, did I find disturbing / disconcerting?

# Documentation



# Tests as Documents

what we're describing

```
describe("an unstarted auto") {
```

```
    val auto = Auto()
```

the generalized behavior it supports

```
    it("idles engine when started") {
```

```
        auto.depressBrake()
```

```
        auto.pressStartButton()
```

```
        auto.RPM() shouldBe 1000 +- 50
```

```
}
```

```
}
```

one example of that behavior

***Seek to first understand through the tests.***

# Behavior-Driven Structuring

```
describe("given some context") {  
    describe("when some event occurs") {  
        it("then can be verified") {  
            ...  
        }  
    }  
}
```

# BDD Structuring Example

```
describe("given a checked-out material") {  
    val material = new Material(AgileJava)  
    val dueDate = material.borrow(PatronId)  
  
    describe("when returned late") {  
        material.returnIt(dueDate.plusDays(2))  
  
        it("is marked available") {  
            material.isAvailable shouldBe (true)  
        }  
        it("generates late fine") {  
            material.fine shouldBe (2 * Material.FineAmount)  
        }  
    }  
}
```

# **[[ TODO ]] FunSpec Fixtures**

---

- <http://doc.scalatest.org/1.7.2/org/scalatest/fixture/FunSpec.html>

# Iterative Naming

---

- Re-consider name continually

```
it("does something")
```

```
it("verifies engine start")
```

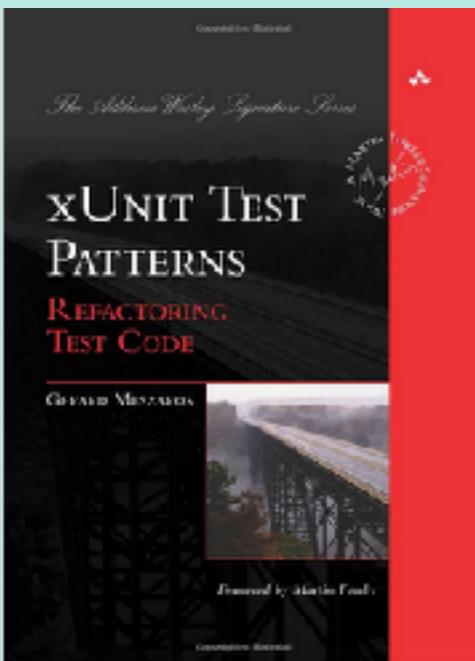
```
it("verifies engine start idle speed")
```

```
it("has low idle on engine start")
```

- Review often and holistically!

# Sustainable Tests

*The long  
answer:*



*A quicker route:*

- Single behavior tests
- AAA
- Correlate result with context
- Test abstraction

# Exercise: Test Smells

Find and fix test smells.  
Paraphrase cleaned tests  
to your pair.



Further instructions: com.langrsoft.pos.CheckoutTest

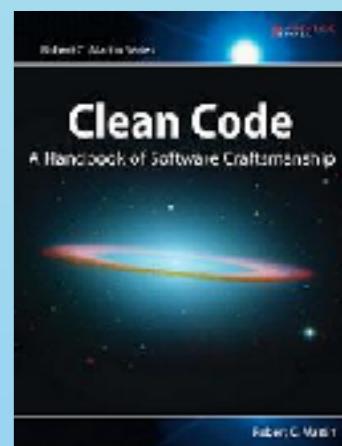
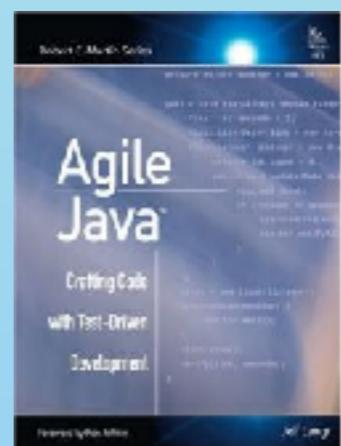
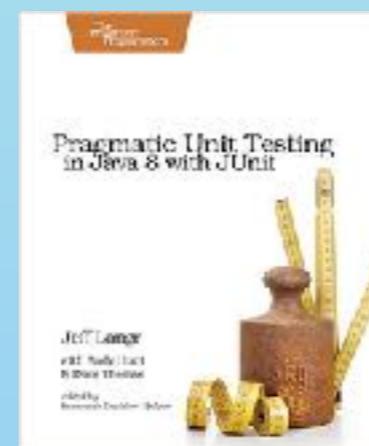
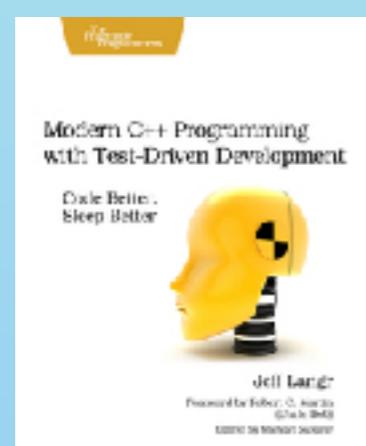
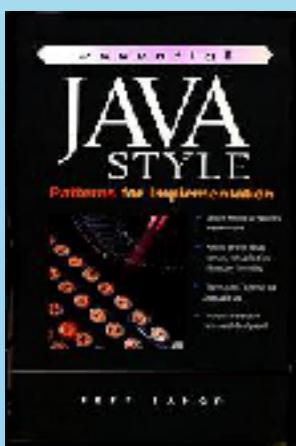
# Continual Design



**ICON**  
**AGILITY SERVICES**

**Langr**  
*Software Solutions*

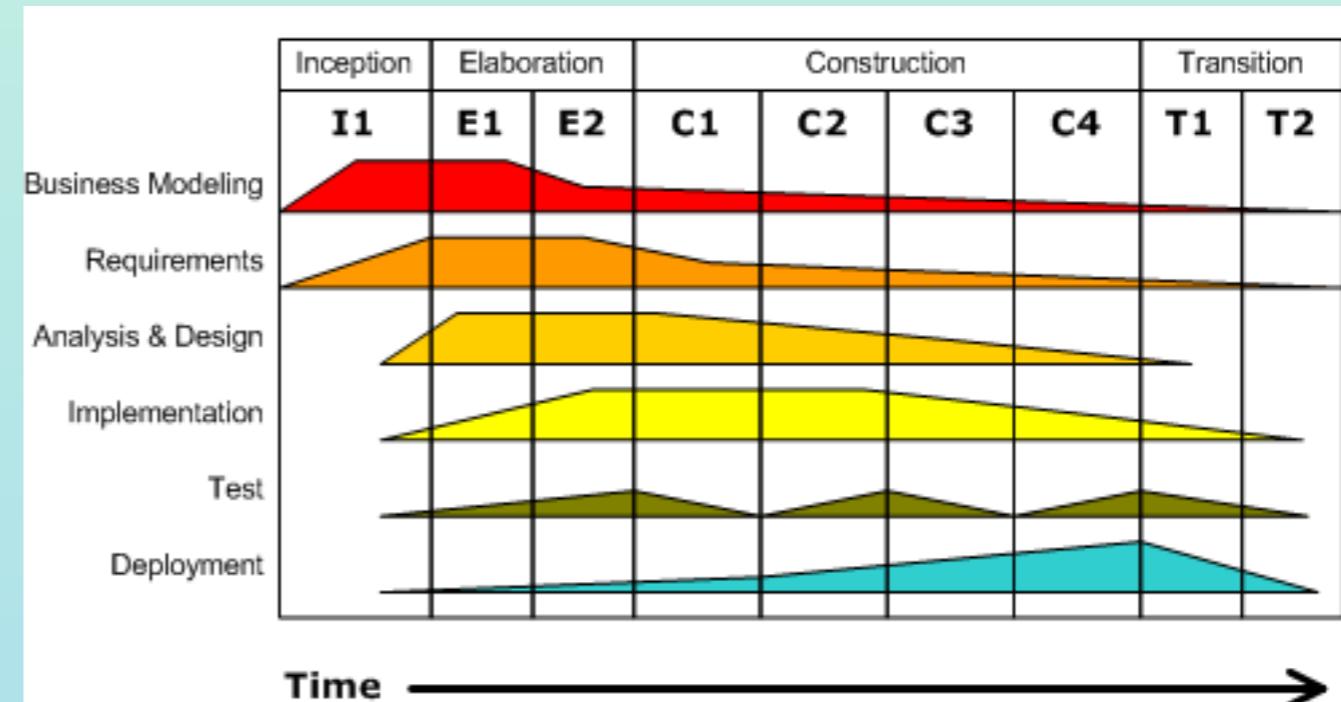
SOFTWARE TRAINING/CONSULTING



# Software Development

## ■ Activities:

- Analysis, design, coding, testing, review, documentation, planning, deployment, ...

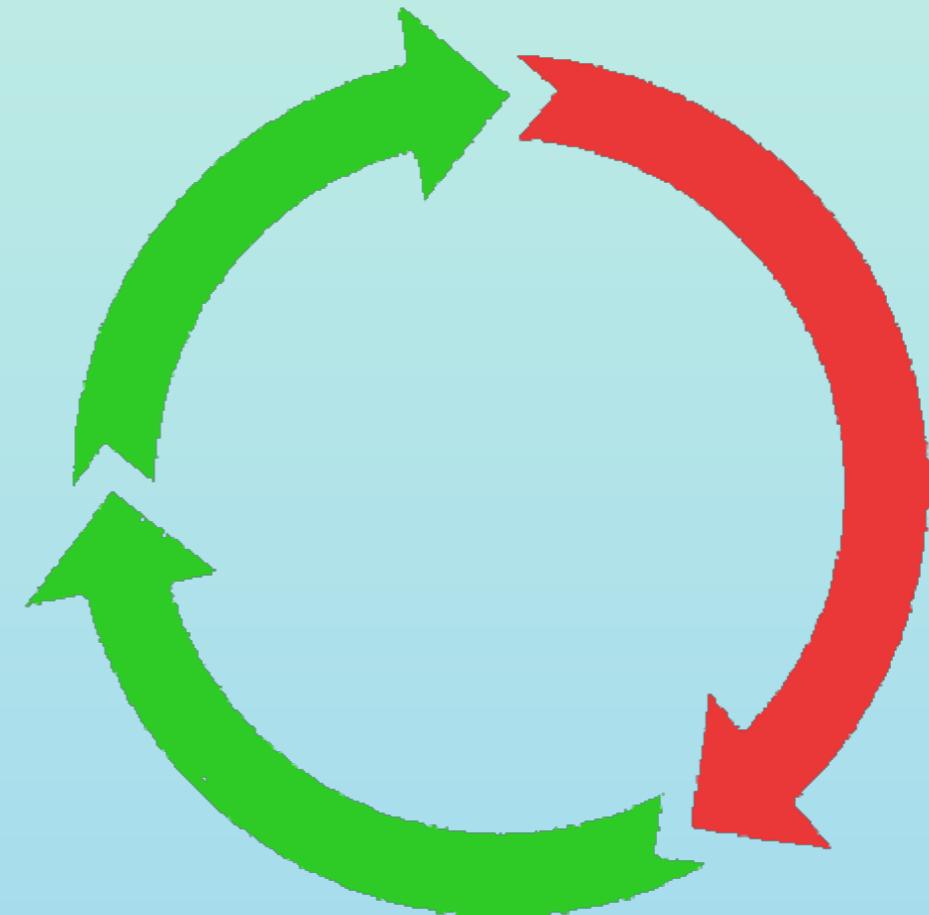


## ■ Agile:

- *All activities all the time*

# TDD: Continual ...

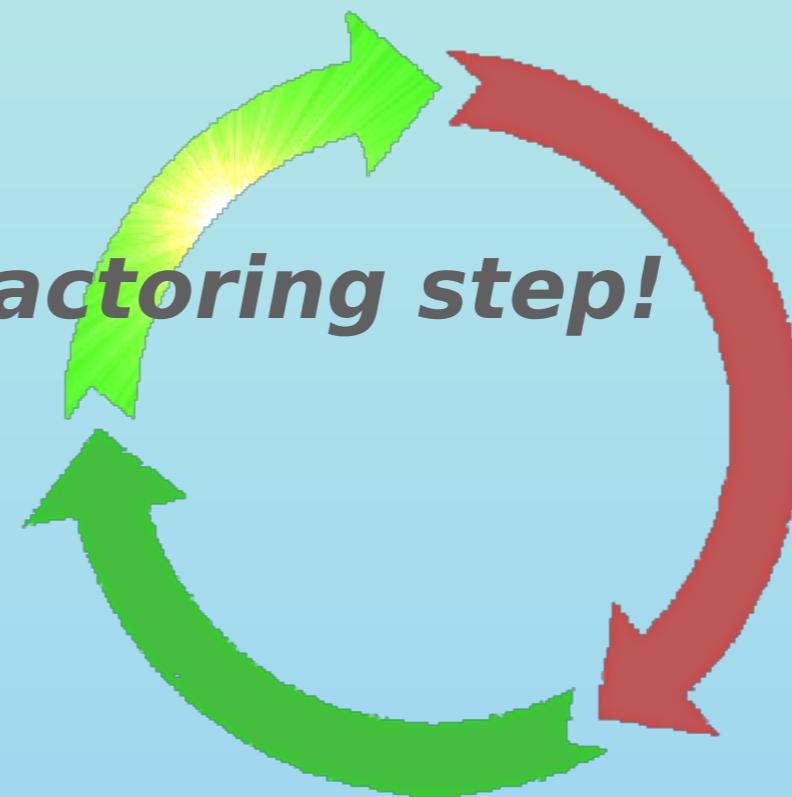
- Testing
- Coding
- Design
- Documentation
- Review



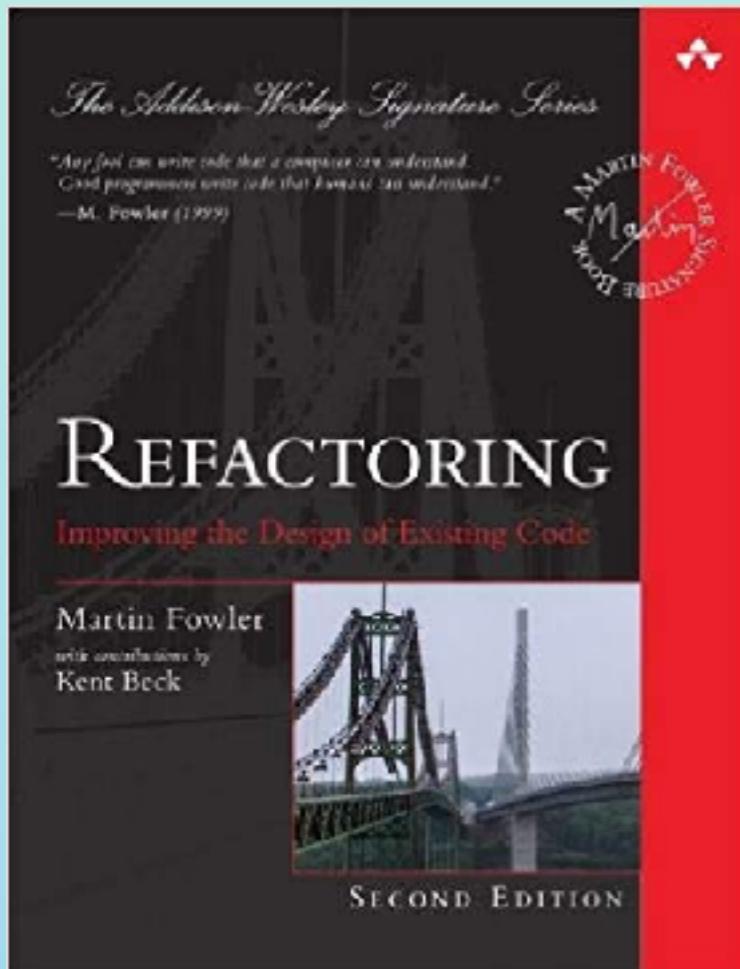
# Continual Design

- Always retain an optimal design
- Entropy is unavoidable

***Never skimp on the refactoring step!***



# Refactoring



Same "*externally recognized*" behavior

# Code Smells

- Comments
- Duplicate Code
- Feature Envy
- Large Class
- Long Method
- Long Parameter List
- Primitive Obsession
- Shotgun Surgery
- Speculative Generality
- Switch Statements
- . . . and many more



"A complete code smells reference:"  
<https://github.com/lee-dohm/code-smells>

# The Nature of the Smell

Aroma?



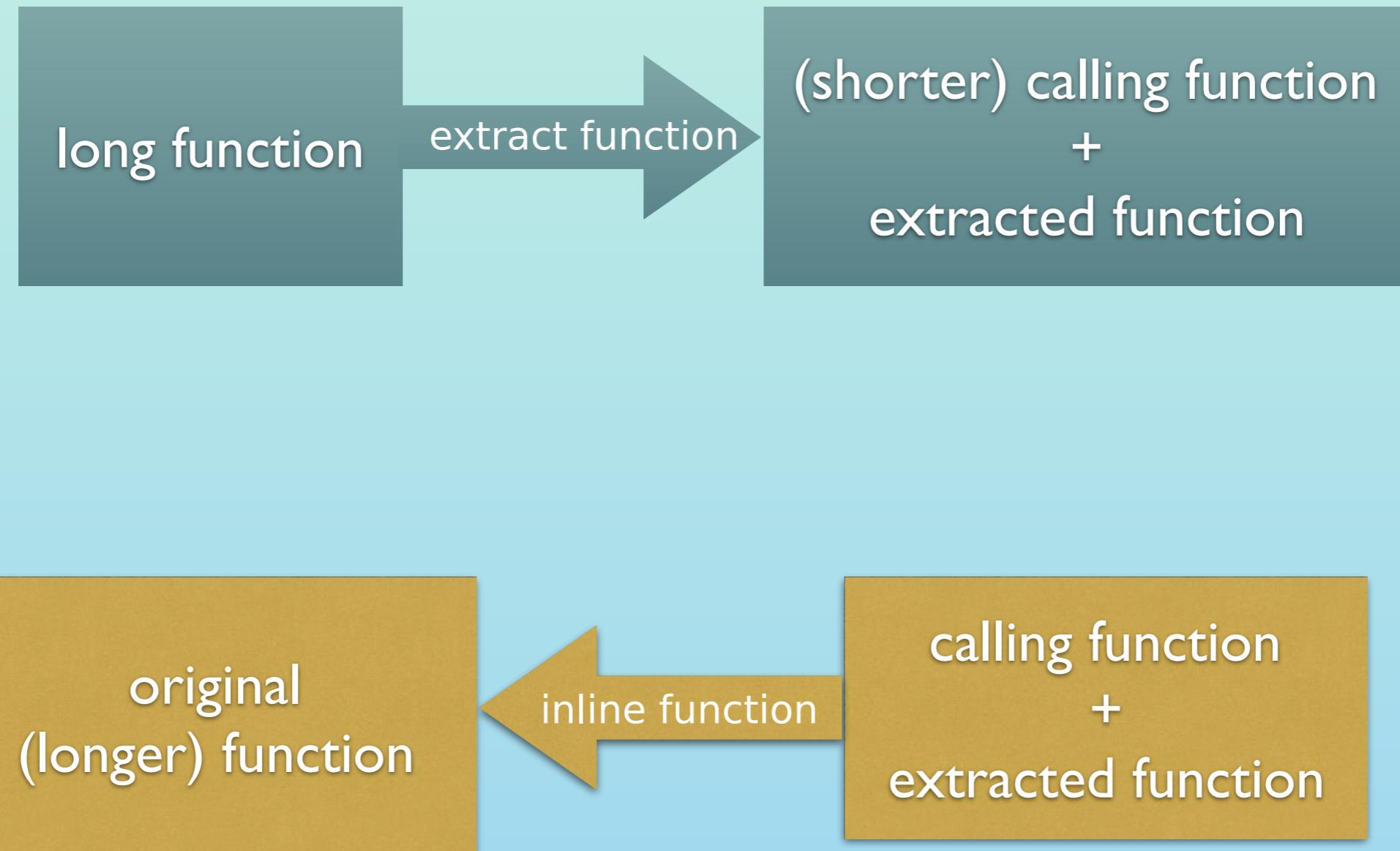
Stench?



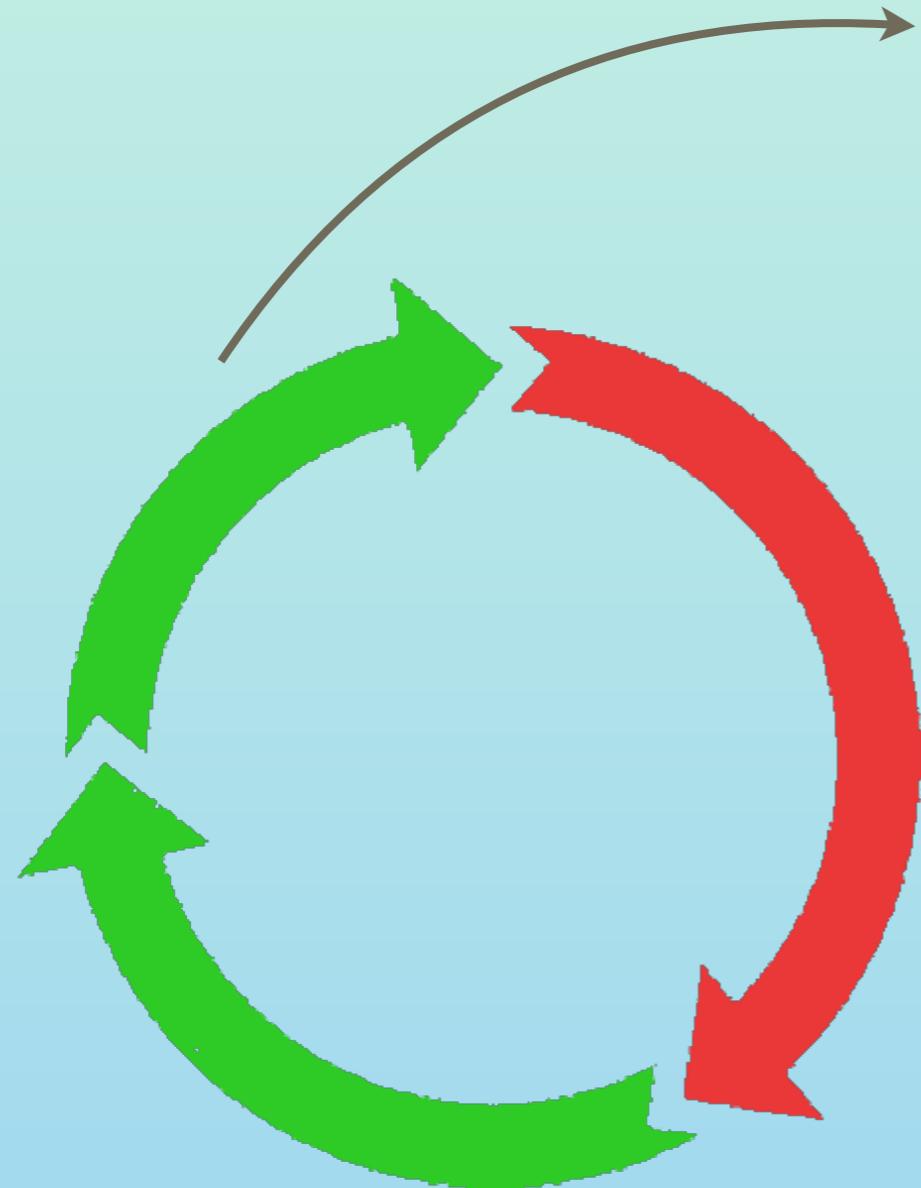
"beagle smelling roses," courtesy Grangernite, <https://www.flickr.com/photos/lidoty/2573672102>  
<https://creativecommons.org/licenses/by/2.0/>

adapted from "Yum, that smells good!", courtesy Tankboy,  
<https://www.flickr.com/photos/tankboy/284958387>; <https://creativecommons.org/licenses/by-sa/2.0/>

# For (almost) Every Transform...



# Refactoring Opportunities



small cleanup  
verify  
small cleanup  
verify  
...

**One thing at a time!**

Break things once in a while.

# Refactoring: Extract Function

```
def uglyFunction() = {  
    // stuff a  
    ...  
    // stuff b  
    ...  
    // some smaller behavior  
    doSomething()  
    doSomethingElse()  
    // ...  
    // stuff c  
    // ...  
}
```



```
def lessUglyFunction() = {  
    // stuff a    ...  
    // stuff b  
    ...  
    someSmallerBehavior()  
    // ...  
    // stuff c  
    // ...  
}  
  
def someSmallerBehavior() = {  
    doSomething()  
    doSomethingElse()  
}
```

## Why?

# Single-Line Extract?

```
catch {  
  case e: RuntimeException => {  
    val msg =  
      s"${new DateTime(): ${System}-${Module}} HIGH ${trunc(e.getMessage, 80)}"  
    logError(msg)  
    throw new RuntimeException(errMsg, e) } }
```



```
catch {  
  case e: RuntimeException => {  
    logError(format(e, "HIGH"))  
    throw new RuntimeException(format(e, "HIGH"), e)  
  }  
}
```

```
private def format(e: RuntimeException, severity: String) = {  
  s"${new DateTime(): ${System}-${Module}} $severity ${trunc(e.getMessage, 80)}"  
}
```

# Exercise: Extract Function

---



- In `com.langrsoft.pos.CheckoutRoutes`:
  - Start with a single-line function extract on `createReceipt`
  - Apply additional function extracts
  - Otherwise focus on renaming and eliminating lies

# [[ TODO ]] Refactoring-Inhibiting Temp

---

```
checkout.items.forEach(item => {
  let price = item.price;
  const isExempt = item.exempt;
  if (!isExempt && discount > 0) {
    const discountAmount = discount * price;
    const discountedPrice = price * (1.0 - discount);

    totalOfDiscountedItems += discountedPrice;

    let text = item.description;
    const amount = parseFloat(Math.round(price * 100) / 100).toFixed(2)
    const amountWidth = amount.length;
    let textWidth = LineWidth - amountWidth;
    messages.push(pad(text, textWidth) + amount);

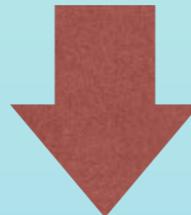
    total += discountedPrice;

    const discountFormatted = '-' + parseFloat(Math.round(discountAmount * 100) / 100).toFixed(2)
    textWidth = LineWidth - discountFormatted.length;
    text = ` ${discount * 100}% mbr disc`;
    messages.push(` ${pad(text, textWidth)}${discountFormatted}`);

    totalSaved += discountAmount;
}
```

# Refactoring: Replace Temp With Query

```
const discountAmount = discount * price;  
// . . .  
  
// discount line  
const discountFormatted = '-' + parseFloat(Math.round(discountAmount * 100) / 100).toFixed(2)  
textWidth = LineWidth - discountFormatted.length;  
text = ` ${discount * 100}% mbr disc`;  
messages.push(`${pad(text, textWidth)}${discountFormatted}`);  
  
totalSaved += discountAmount;
```



```
// discount line  
const discountFormatted =  
  '-' + parseFloat(Math.round(discountAmount(discount, price) * 100) / 100).toFixed(2)  
textWidth = LineWidth - discountFormatted.length;  
text = ` ${discount * 100}% mbr disc`;  
messages.push(`${pad(text, textWidth)}${discountFormatted}`);  
  
totalSaved += discountAmount(discount, price);  
// ...  
});  
  
const discountAmount = (discount, price) => discount * price;  
// . . . . .
```

# Replace Temp With Query

---

Steps:

- Make temp const
- Extract Function on the rhs
- Replace temp references with the query
- Remove temp

Why?

Concerns?

Can we go the other way?

# Exercise: Replace Temp with Query

---



- In com.langrsoft.pos.CheckoutRoutes
  - Apply Replace Temp with Query in postCheckoutTotal as appropriate
  - Extract as many additional functions as you can

# Code Smell: Feature Envy

```
class WrongPlace {  
    client() {  
        const x = enviousFunction()  
        ...  
    }  
  
    enviousFunction() {  
        const there = new GreenerGrass()  
        const result = there.doStuff(this.someValue)  
        there.doMoreStuff(result)  
        return there.answer()  
    }  
}
```

*Why is this a problem?*

# Refactoring: Move Function

```
class WrongPlace {  
    client() {  
        const x = new GreenerGrass().homebodyMethod(this.someValue)  
        ...  
    }  
}  
  
class GreenerGrass {  
    homebodyMethod(someValue) {  
        var result = doStuff(someValue)  
        doMoreStuff(result)  
        return answer()  
    }  
    ...  
}
```

*What must you also do when you move a function?*

# Exercise: Move Function

- In com.langrsoft.pos.CheckoutRoutes
- Move at least 2 envious functions to better (or new!) homes
- Ensure you've added documentation (i.e tests)!



# Macro Refactorings



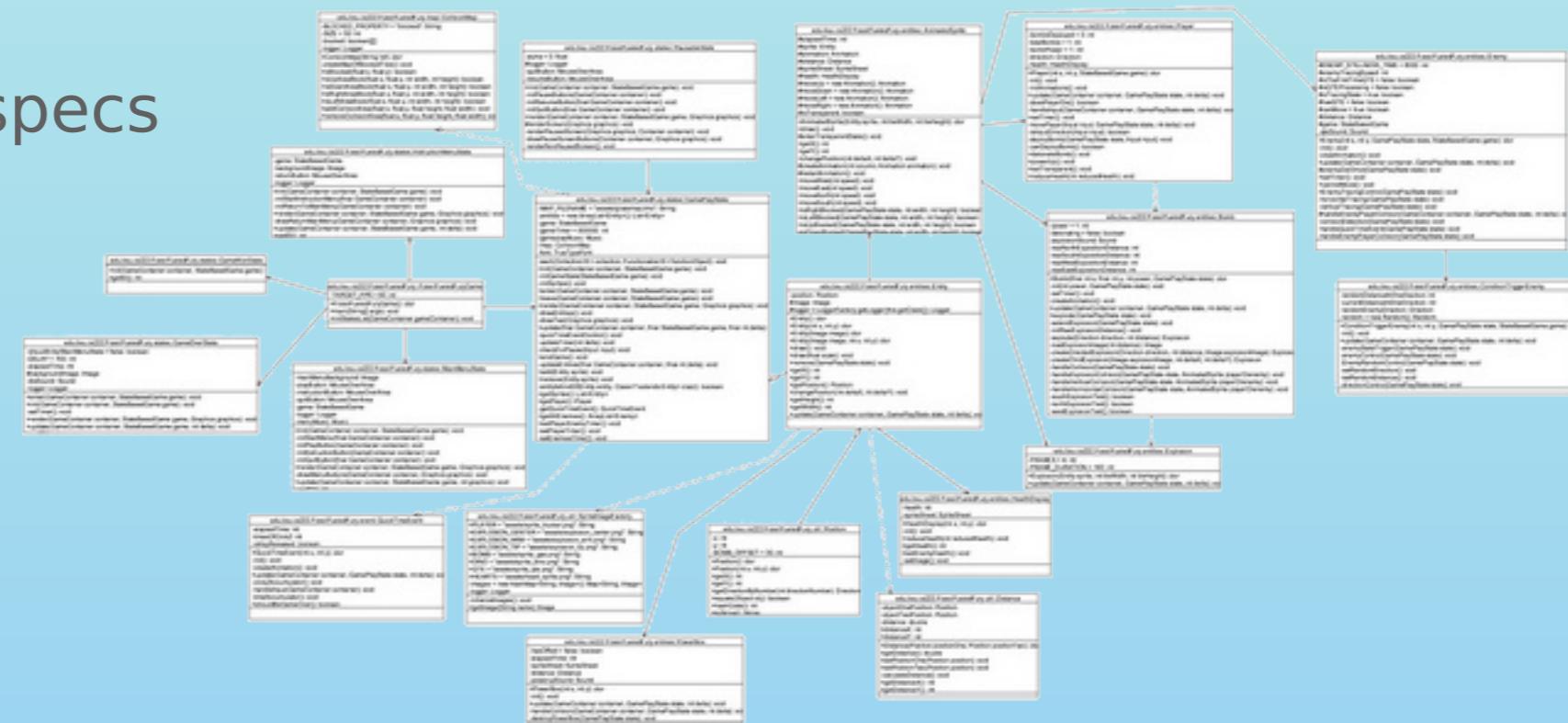
Might need to backtrack!

# Planning: Up-Front Design

- Not just once!
    - Planning: project, release, iteration, day, task, test
    - Any estimation

# ■ Minimal: sketches & conversations

- ## ■ Not detailed specs



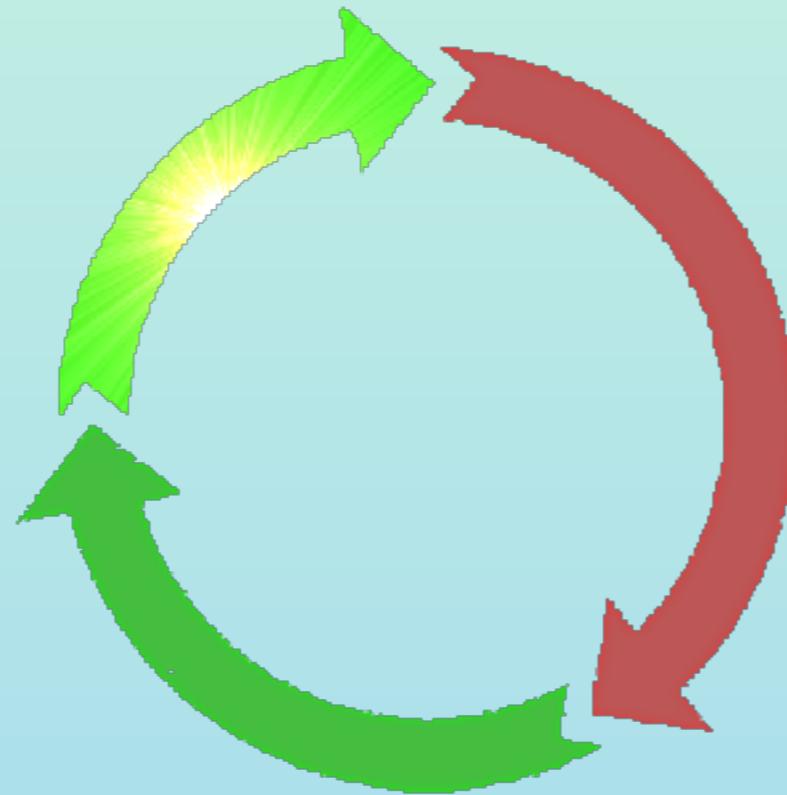
# Premature Generalization

- Need might never materialize
- Details might change
- Interim complexity \$



# Refactoring Guidelines

Single-goal



Run *all* tests

It's your responsibility

Never skip!

# Design Drivers / Guidelines

- Code smells
- SOLID
- Design patterns
- GRASP
- Simple design
- DRY
- ...

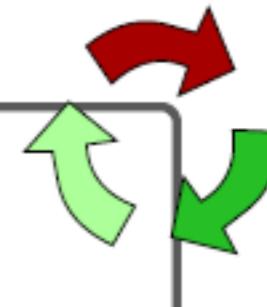


**It's all good!**

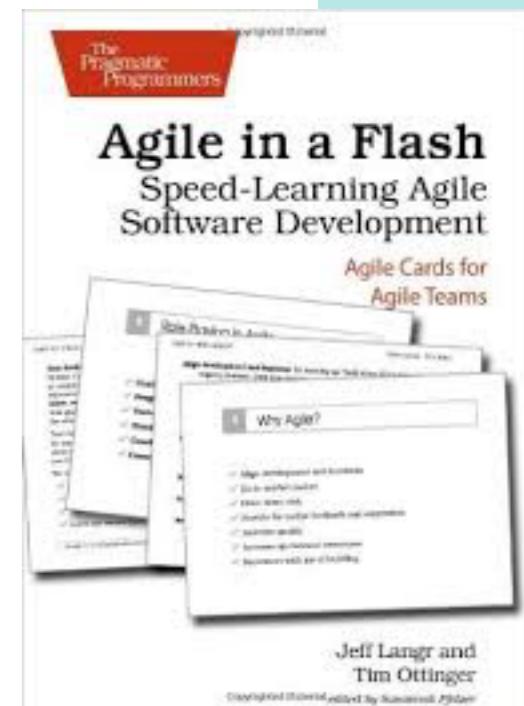
*Does everything point to the same place?*

41

## Build Superior Systems with Simple Design



- All tests must pass
- No code is duplicated
- Code is self-explanatory
- No superfluous parts exist



Kent Beck's (ordered) rules for emergent design

# Exercise: Simple Design Rules

- In com.langrsoft.pos.CheckoutRoutes
  - Stamp out duplication!
  - Strive for rapid readability
  - Have you gone too far?

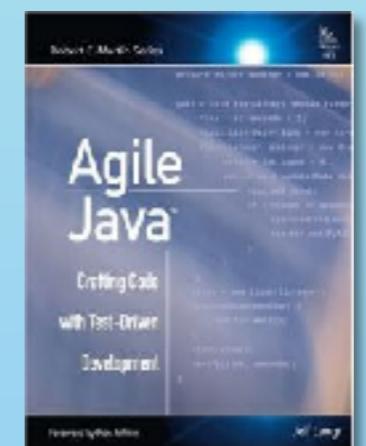
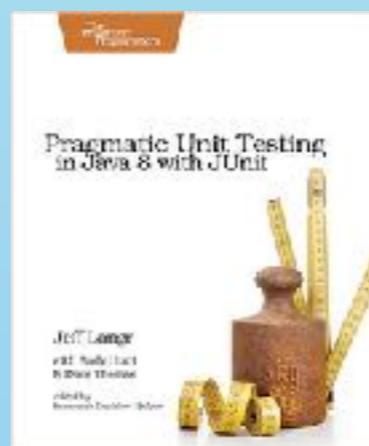
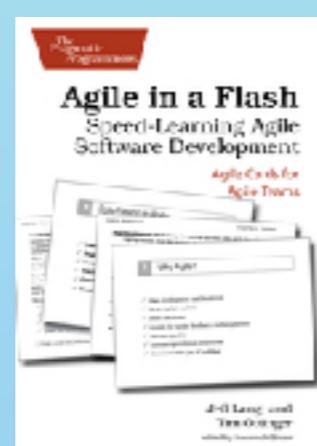
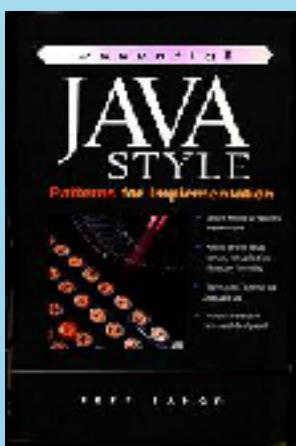


# Test Doubles

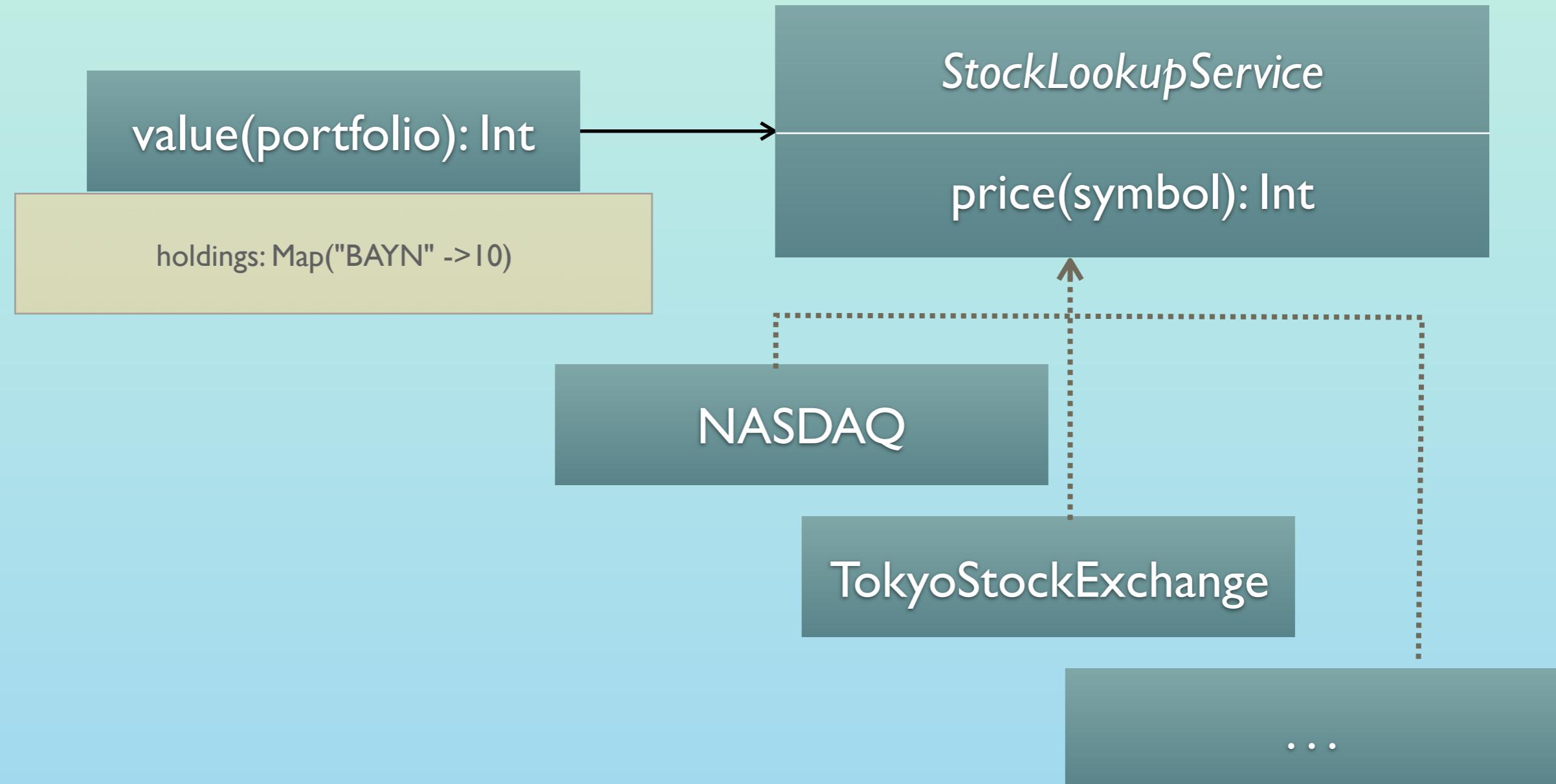


Langr  
Software Solutions

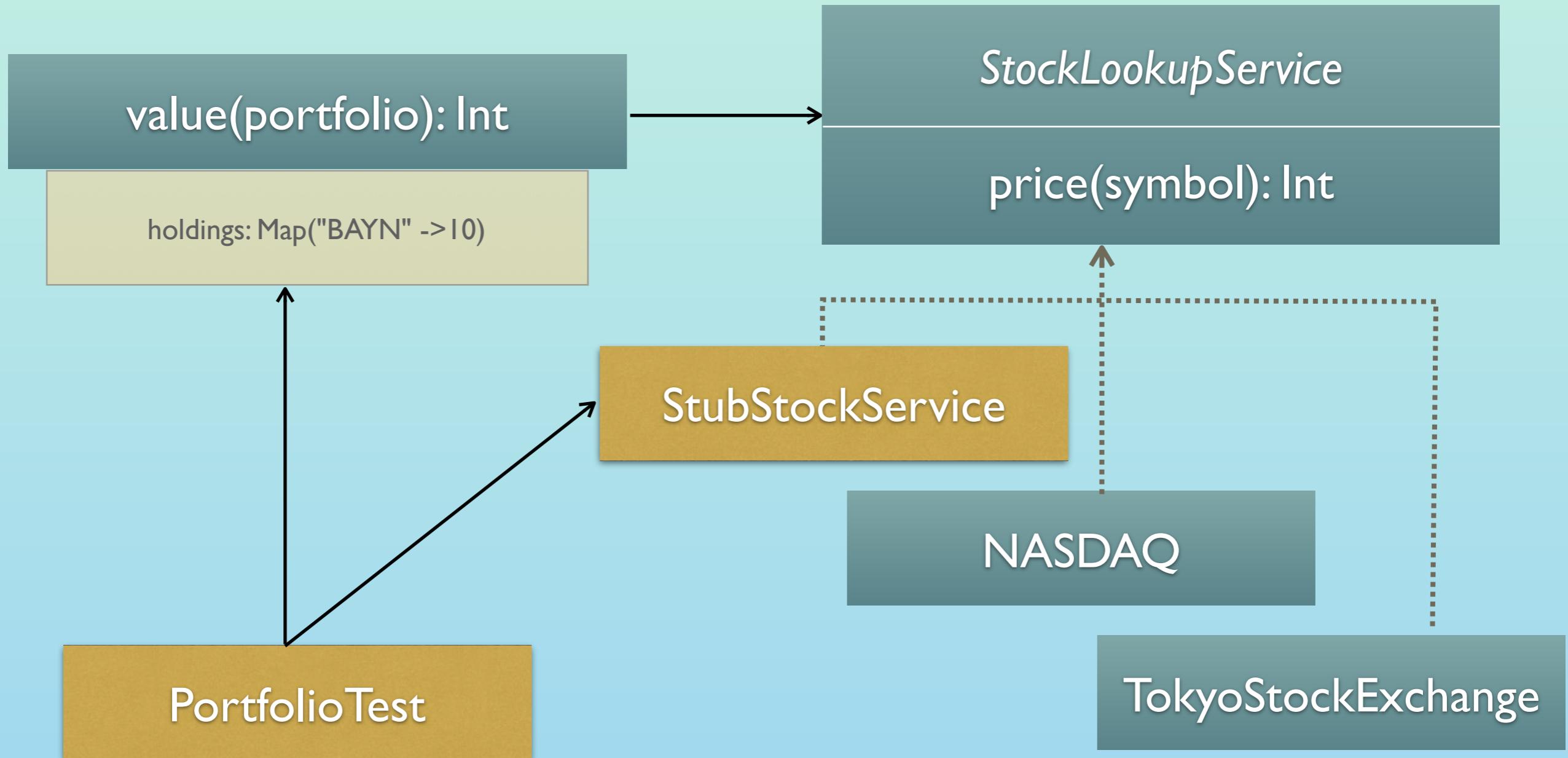
SOFTWARE TRAINING/CONSULTING



# Testing Challenge: Portfolio



# Using a Test Double



# Portfolio Value...

StockService:

```
trait StockService { def price(symbol: String): Int }
```

```
it("is share value after purchase single share") {
    val stockService = new StockService {
        override def price(symbol: String): Int = BayerPrice
    }

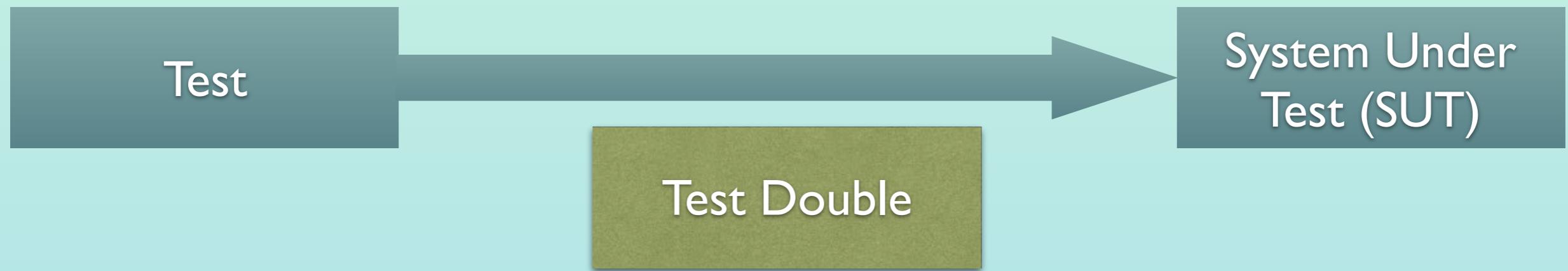
    portfolio.purchase("BAYN", 1)

    portfolio.value(stockService) shouldBe BayerPrice
}
```

Incremental implementation in Portfolio:

```
def value(service: StockService) = {
    val soleSymbol = symbols.keySet.head
    service.price(soleSymbol)
}
```

# [[ TODO ]] Test Double Injection Techniques



Function argument  
Constructor  
Factory method override  
Cake pattern  
Thin cake pattern  
Structural typing  
Implicit declarations  
Using the reader monad

Guice  
MacWire  
Mockito  
Spring

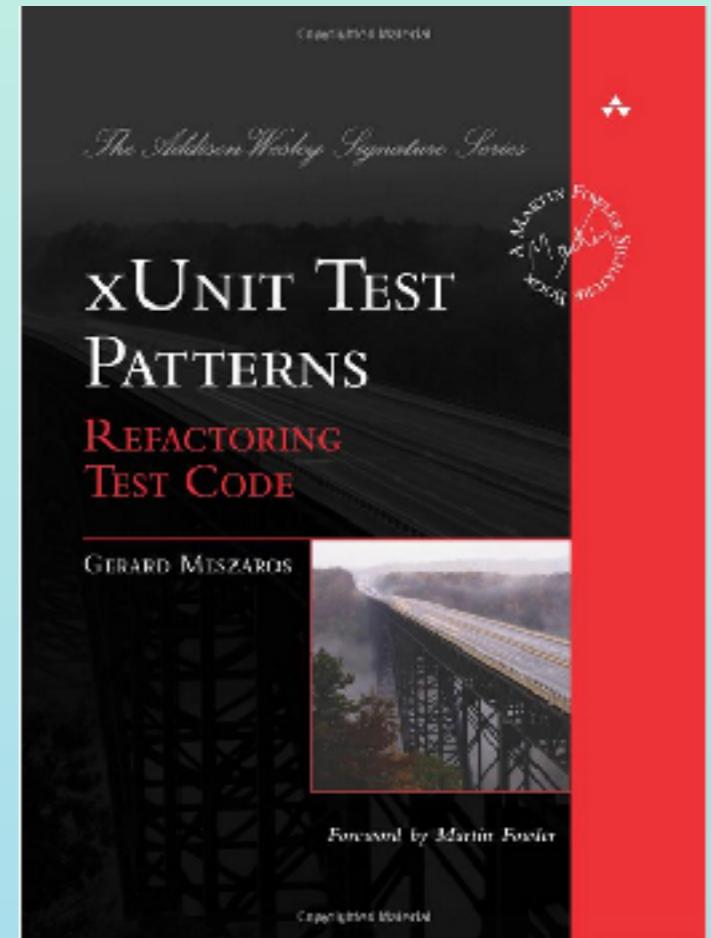
# Test Double Terms

**Stub:** dumb emulation

**Spy:** captures values to verify

**Mock:** self-verifies

**Fake:** whole collaborator emulation



# Mock Tools



(and *mockito-scala*)

ScalaMock (paul butcher)

EAS~~Y~~MOCK

jMock



JMockit

*History suggests more & changes coming!*

# Mockito / Mockito-Scala

<http://site.mockito.org>

<https://github.com/mockito/mockito-scala>



# Mockito-Scala: A Simple Stub

```
class PortfolioTest extends FunSpec with IdiomaticMockito
// ...
```

```
val stockService = mock[StockService]
val portfolio = new Portfolio(stockService)
stockService.price("BAYN") shouldReturn BayerPrice
stockService.price("IBM") shouldReturn IbmPrice
```

```
portfolio.purchase("BAYN", 10)
portfolio.purchase("IBM", 20)
```

```
portfolio.value shouldBe BayerPrice * 10 + IbmPrice * 20
```

<https://static.javadoc.io/org.mockito/mockito-core/2.7.7/org/mockito/Mockito.html>

# Exercise: Mockito

---



Replace your hand-crafted test double  
with a Mockito-defined stub

# Mock Tools Are Trackers

---

**In the debugger:**

- .mockitoInterceptor
- .handler
- .mockHandler
- .invocationContainer
- .registeredInvocations
- .invocations

# Constructor Injection

```
val stockService = mock[StockService]  
val portfolio: Portfolio = new Portfolio(stockService)
```

```
class Portfolio(stockService: StockService) {  
    // ...  
}
```

# Mockito Injection

```
import org.mockito.{InjectMocks, Mock, MockitoAnnotations, // ...}  
// ...  
class PortfolioTest extends FunSpec // ... {  
  @InjectMocks var portfolio: Portfolio =_  
  @Mock var stockService: StockService =_  
  
  before {  
    portfolio = new Portfolio  
    MockitoAnnotations.initMocks(this)  
  }  
  // ...
```

```
class Portfolio {  
  var stockService: StockService =_  
  // ...
```

# Mockito: Exceptions

```
it("sets price to 0 when price lookup throws") {  
    stockService.price("IBM") shouldThrow(new RuntimeException)  
  
    portfolio.purchase("IBM", 20)  
  
    portfolio.value shouldBe 0  
}
```

# Exercise: Exceptions

---



When calculating the portfolio value,  
use zero dollars for the current stock price  
if the symbol lookup throws an error.

# Verifying a "Tell"

```
def purchase(symbol: String, sharesToBuy: Integer) =  
  sharesToBuy match {  
    case n if n > 0 =>  
      symbols += symbol -> (sharesToBuy + shares(symbol))  
      auditor.audit(  
        s"Purchased $sharesToBuy shares of $symbol")  
      // ...  
  }
```

How to test-drive the call to audit?

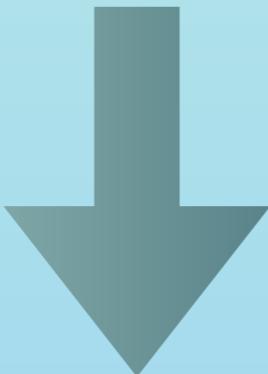
# Spying on the Audit Call

```
it("audits on purchase") {  
    val stockService = mock[StockService]  
    val auditor = mock[Auditor]  
    val portfolio = new Portfolio(stockService, auditor)  
  
    portfolio.purchase("BAYN", 10)  
  
    auditor.audit("Purchased 10 shares of BAYN") was called  
}
```

# Argument Matching

```
trait Auditor {  
    def audit(symbol: String, timestamp: DateTime)  
}
```

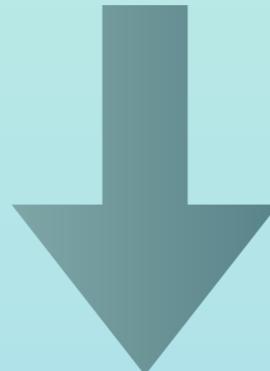
```
auditor.audit("Purchased 10 of BAYN",  
any[DateTime]) was called
```



```
auditor.audit(s"Purchased $sharesToBuy of $symbol",  
new DateTime)
```

# Argument Matching... Hmm.

```
auditor.audit("Purchased 10 of BAYN",  
any[DateTime]) was called
```



```
auditor.audit(s"Purchased $sharesToBuy of $symbol",  
null)
```

This passes!

# Argument Capture

```
val captor = ArgCaptor[DateTime]
val now = new DateTime

portfolio.purchase("BAYN", 10)

auditor.audit("buy: 10 of BAYN", captor) was called
secondsBetween(captor.value, now) should be < 1
```

# Exercise: Spies

Verify that sell transactions are audit-logged with:

- date-time
- transaction description ("sell")
- symbol sold
- # shares sold



```
trait Auditor {  
    def audit(  
        symbol: String,  
        shares: Int,  
        transactionType: String,  
        timestamp: DateTime)  
}
```

# Mockito-Scala: Still More!

---

- Expect # of invocations or "never"
- Spies of real objects / partial mocks (avoid!)
- Verification in order
- Mocking consecutive calls
- Support for by-name arguments

# Cake Pattern

```
trait StockServiceComponent {  
    val service: PriceService  
  
    trait PriceService {  
        def price(symbol: String): Int  
    }  
}
```

```
trait ProdStockServiceComponent  
    extends StockServiceComponent {  
    val service = new ProdPriceService()  
  
    class ProdPriceService() extends PriceService {  
        def price(symbol: String): Int = { /* */ }  
    }  
}
```

```
class Portfolio {  
    this: StockServiceComponent => // "self type;" allows mixing-in a stock service  
  
    // ...  
  
    def value =  
        symbols.keysIterator.foldLeft(0) {  
            (total, symbol) =>  
                total + shares(symbol) * service.price(symbol)  
        }  
}
```

# Testing Via the Cake Pattern

```
class PortfolioTest // ...
{
  trait TestStockServiceComponent extends StockServiceComponent {
    val service = new TestPriceService()
    class TestPriceService() extends PriceService {
      def price(symbol: String): Int = {
        symbol match {
          case "BAYN" => BayerPrice
          case "IBM" => IbmPrice
        }
      }
    }
  }

  var portfolio = new Portfolio with TestStockServiceComponent
```

# ... Using Mockito

```
trait MockitoTestStockServiceComponent extends StockServiceComponent
  with IdiomaticMockito {
  val service = mock[PriceService]
  service.price("BAYN") shouldReturn BayerPrice
  service.price("IBM") shouldReturn IbmPrice
}
```

# Cake Is a Mess...

# Debatable.

- Violates ISP, SRP, OCP
  - Uses "is a," not "has a" inheritance



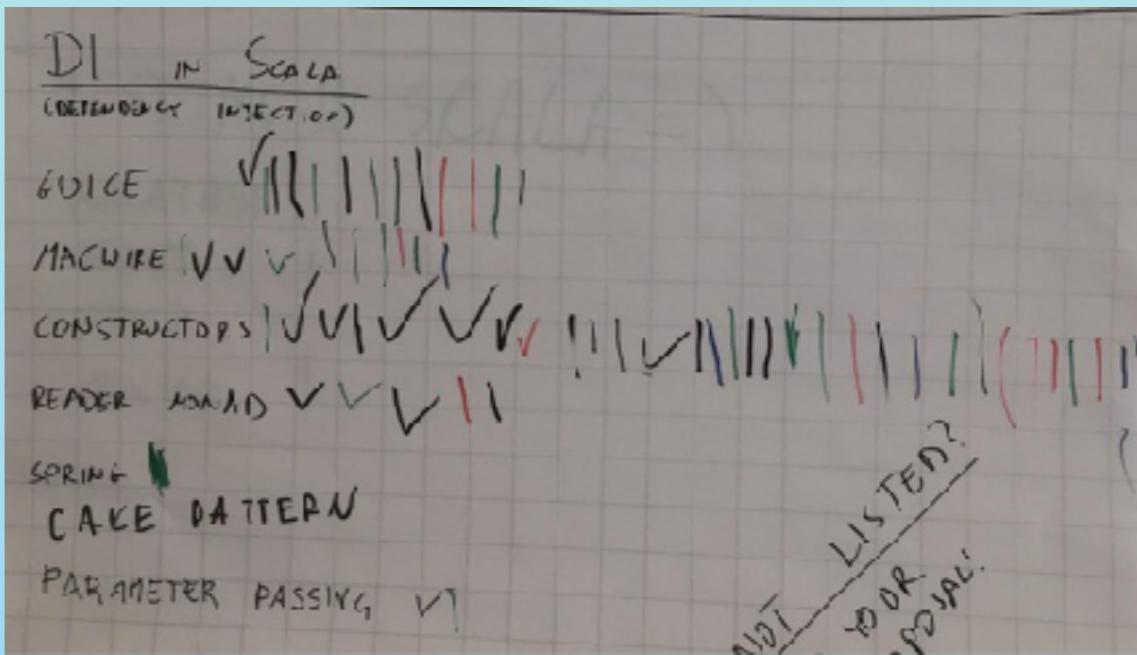
via Bartosz Mikulski:*The Cake Pattern Is a Lie:*

## Some Resources:

<http://di-in-scala.github.io>

<http://jonasboner.com/real-world-scala-dependency-injection-di/>

<https://www.originate.com/thinking/stories/reader-monad-for-dependency-injection/>



# Schools of Mock

Classic  
("Detroit")

algorithmic approach

verification of state

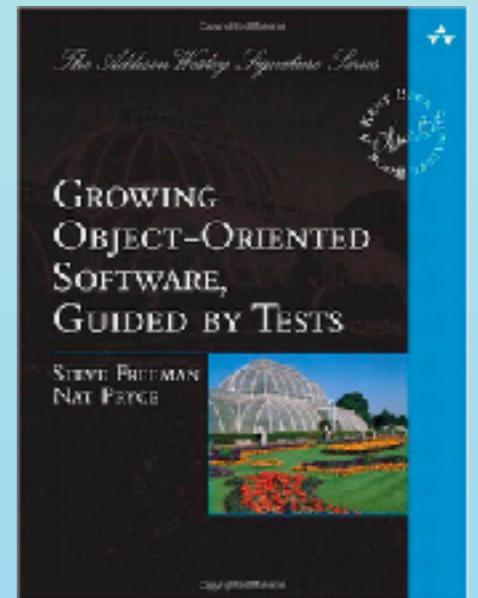
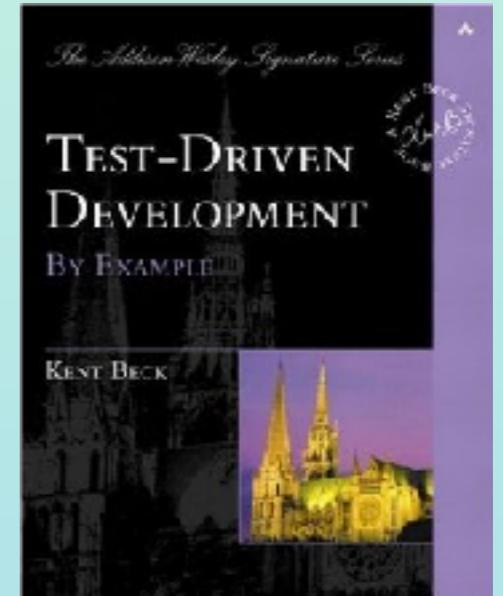
tests drive code from specific to general

London

roles, responsibilities, and interactions

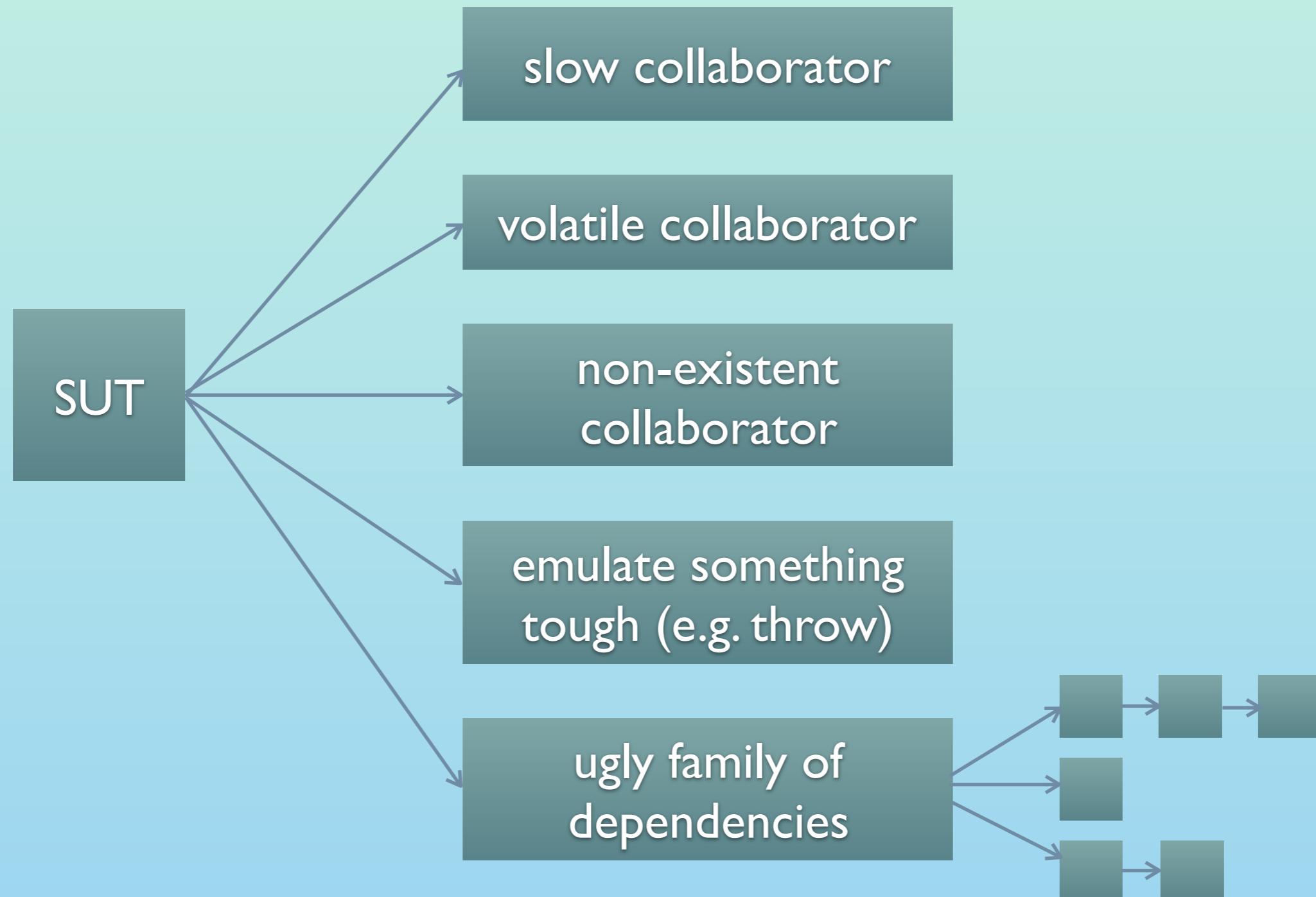
"Fake it until you make it"

drive incrementally from the outside in



***Read Martin Fowler's "Mocks Aren't Stubs"***

# Classic School: When to Mock



# When Not to Mock?

---

Can you inject the data or a generator instead?

# The Pragmatic Mocker

- Writes integration tests where mocks are used
- Watches coupling between tests & implementation
- Seeks to mock only direct collaborators
- Avoids mocking what they don't own
- Avoids fakes (and if necessary, test-drives 'em!)
- Knows their mock tool in & out
- Isolates, minimizes the use of test doubles
  - But prefers testability



# Wrap-Up Exercise



- Support voids on checkout



Work outside in:

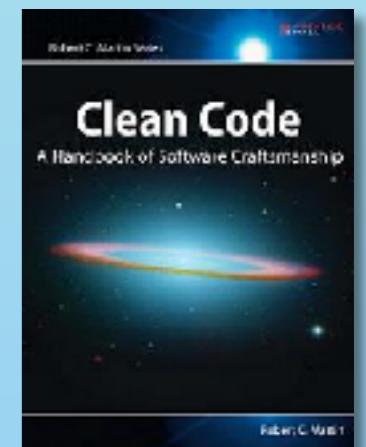
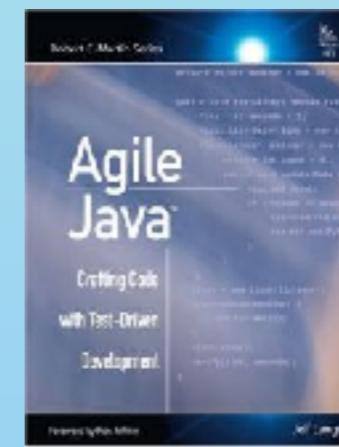
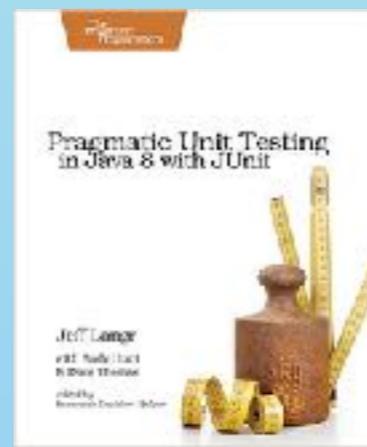
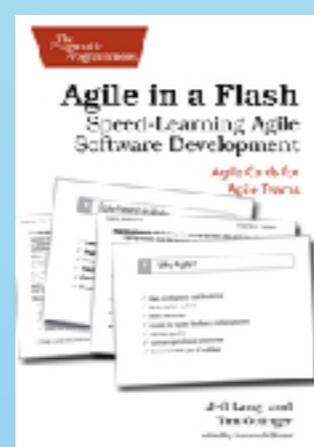
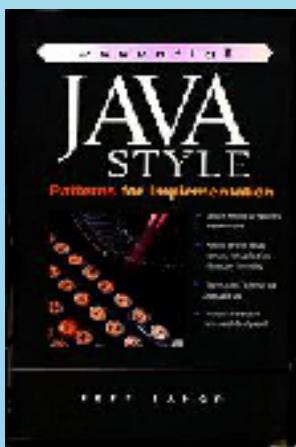
- Stub collaborators until ready to build them out

# Growing & Sustaining TDD

ICON  
AGILITY SERVICES

Langr  
Software Solutions

SOFTWARE TRAINING/CONSULTING



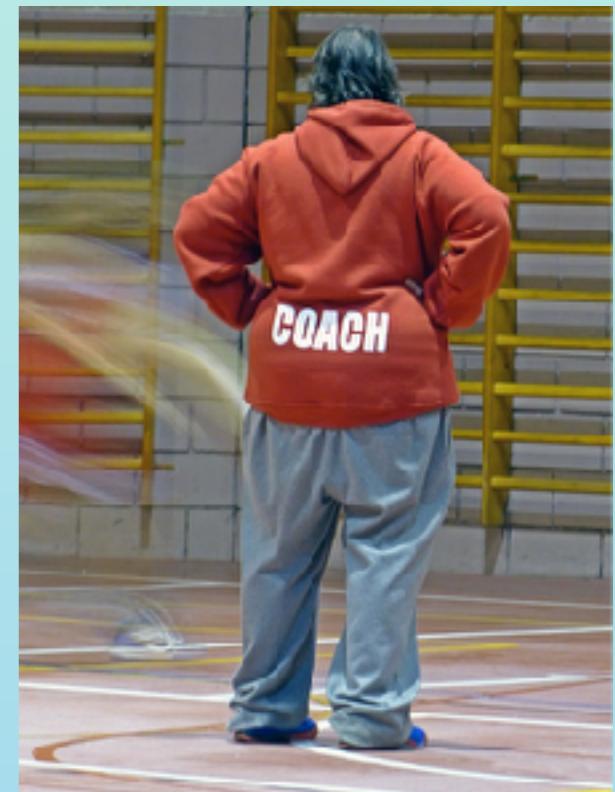
# Next Week



- Create team policies/standards around TDD
- Ensure your build pipeline supports it
- Reviews must include the tests
- Pairing / mobbing helps quite a bit!
- Focus on problem areas first
- Adopt on smaller scale if necessary

# Growing TDD

- Education / sharing sessions
- Challenges / contests / randoris
- Coach / champion
- "Stop the line" mentality
- Changing metrics for education
- Pair & paraphrase
  - *Three+* sets of eyes!
- Mobbing



# Sustaining TDD

Everything under "Growing TDD," plus:

- TDD-specific retrospectives
- Daily stand-down
- Regular reminders as to "why?"
- Revisit standards regularly
- Keep the tests fast
  - Segregate (and fix) slow tests
- Always use tests as an entry point
- Refactor, refactor, refactor
  - Tests too!



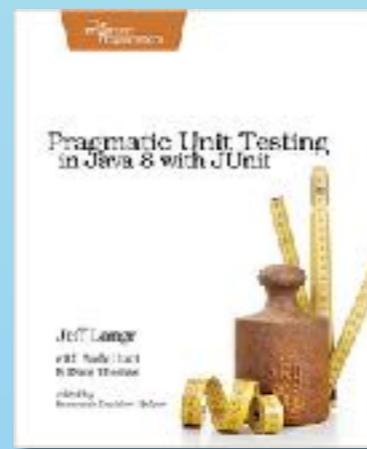
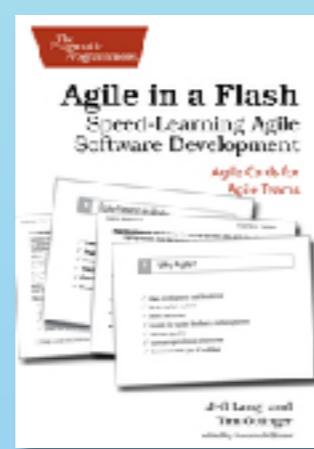
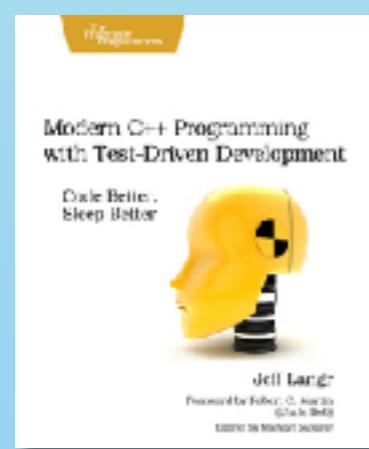
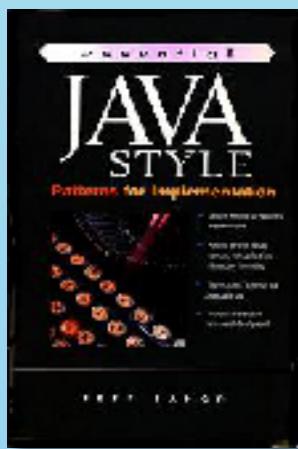
# Mastering TDD

TDD is a skill.



*Practice, practice, practice.*

# Miscellaneous Topics

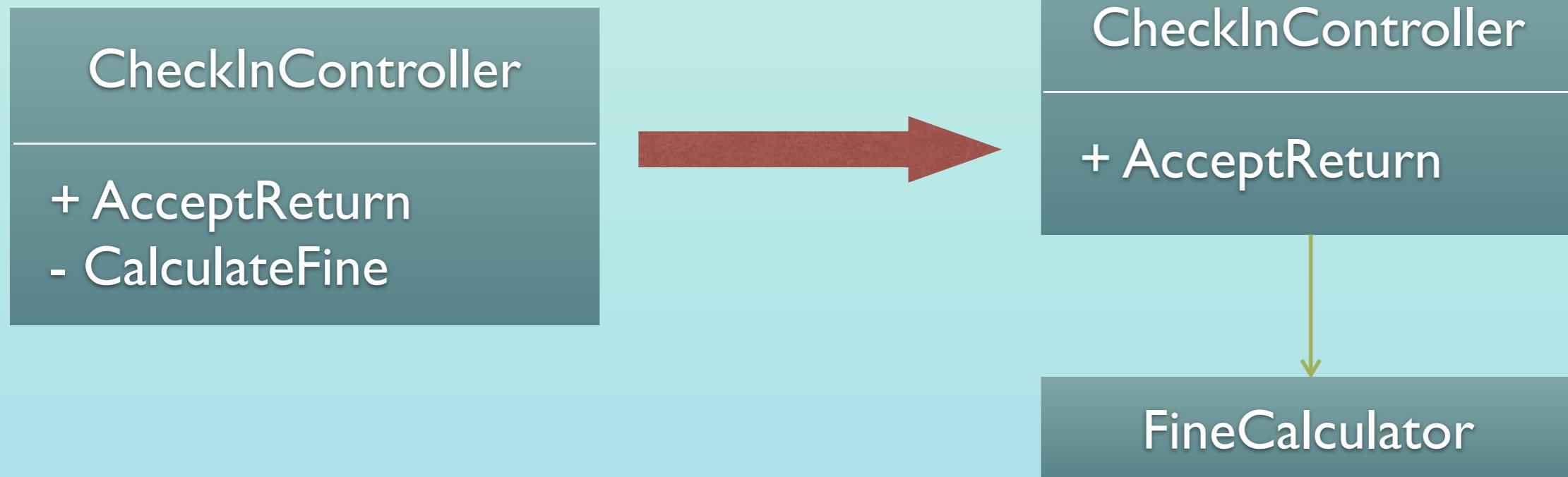


# **[[ TODO ]] Asynchronous Testing**

---

<https://medium.com/@hussachai/beware-the-asyn-test-35c5c848ebe9>

# Testing Non-Public Behavior?



Or simply relax access.

# Rules of Ten



Green & clean in < 10 (minutes)

Delete and repeat if you're late!  
... taking smaller steps this time.

# Rules of Ten



Debate stops at 10 (minutes)

"Show me."

# Rules of Ten

---



All unit tests run in < 10 (seconds).

# Fixing a Slow Test Run



Warn on tests < ***n*** ms

Fail on tests > ***n*** ms

*Running a subset of tests increases risk.*

# Scratch Refactoring



*Discard the results!*

# Uncle Bob's TPP

Transformations have a preferred ordering, which if maintained by ordering of tests, prevents "long outages" in TDD.

The  
Priority  
List

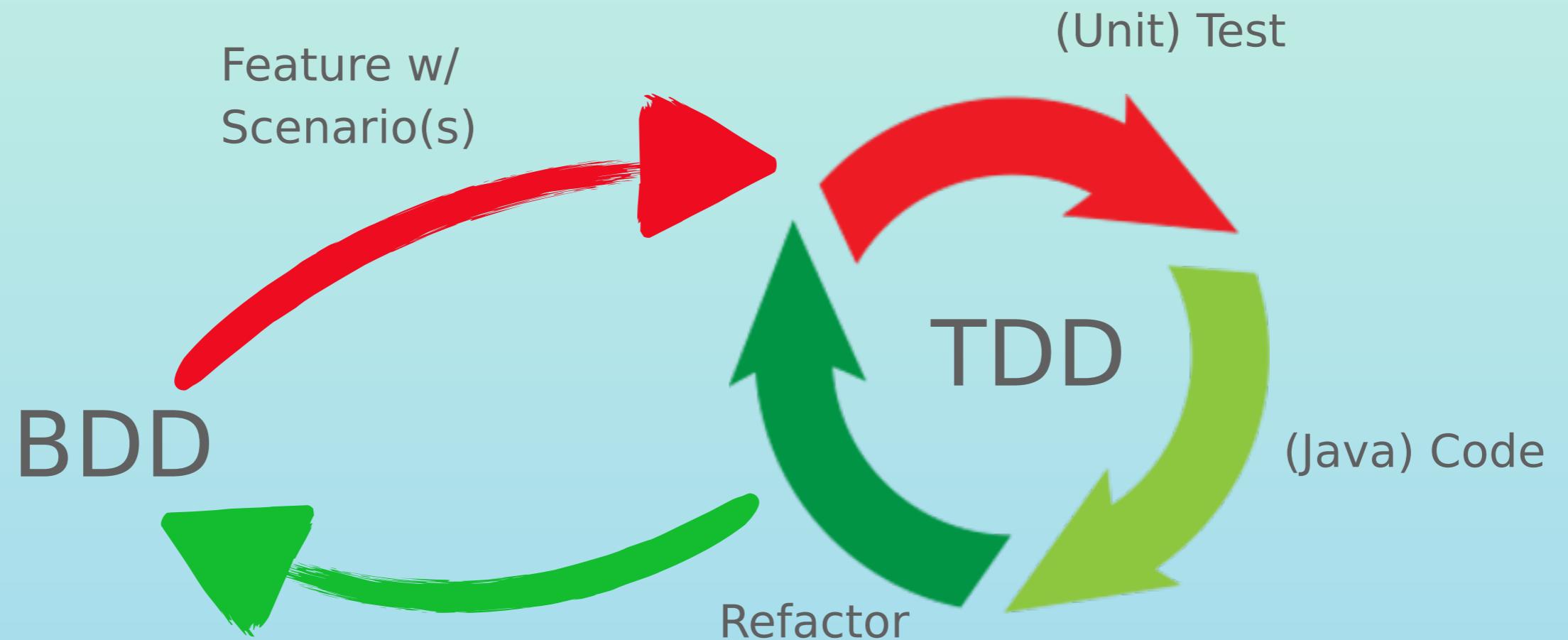
- ({}->**nil**) no code at all->code that employs nil
- (**nil->constant**)
- (**constant->constant+**) a simple constant to a more complex constant
- (**constant->scalar**) replacing a const. with a variable or an argument
- (**statement->statements**) adding more unconditional statements.
- (**unconditional->if**) splitting the execution path
- (**scalar->array**)
- (**array->container**)
- (**statement->tail-recursion**)
- (**if->while**)
- (**statement->recursion**)
- (**expression->function**) replacing expression w/ a function or algorithm
- (**variable->assignment**) replacing the value of a variable.
- (**case**) adding a case (or else) to an existing switch or if

"As the tests get more specific, the code gets more generic."

<http://thecleanncoder.blogspot.com/2011/02/fib-t-p-premise.html>

<https://8thlight.com/blog/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html>

# Behavior-Driven Development (BDD)



# References / Reading List

---

"Chat" icon courtesy Gregor Cesnar, Noun Project (Creative Commons)

[Beck2002] Beck, Kent. **Test-Driven Development: By Example**. Addison-Wesley, 2002.

[Feathers2005] Feathers, Michael. **Working Effectively With Legacy Code**. Prentice Hall, 2005.

[Fowler1999] Fowler, Martin. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 1999.

[Freeman, 2009]. Freeman, S. and Pryce, N. **Growing Object-Oriented Software, Guided By Tests**. Addison-Wesley, 2009.

[Langr2005] Langr, Jeff. **Agile Java: Crafting Code With Test-Driven Development**. Prentice Hall, 2005.

[Langr2011] Langr, Jeff and Ottinger, Tim. **Agile in a Flash**. Pragmatic Programmers, 2011.

[Langr2013] Langr, Jeff. **Modern C++ Programming With Test-Driven Development**, Pragmatic Programmers, 2013.

[Langr2015] Langr, Jeff, et. al. **Pragmatic Unit Testing in Java 8 with Junit**. Pragmatic Programmers, 2015.

[Martin2002] Martin, Robert C. **Agile Software Development: Principles, Patterns, and Practices**. Prentice Hall, 2002.

[Meszaros2007] Meszaros, Gerard. **xUnit Test Patterns: Refactoring Test Code**. Addison-Wesley, 2007.

# Thank you!



**Langr**  
*Software Solutions*

SOFTWARE TRAINING/CONSULTING

