RED
GREEN
REFACTOR

1

Applying TDD

Craig Larman
craiglarman.com & less.works

v.11

2

Opening Topics

3

Some Big Ideas

4

own

vs

rent

5

5

start with

why

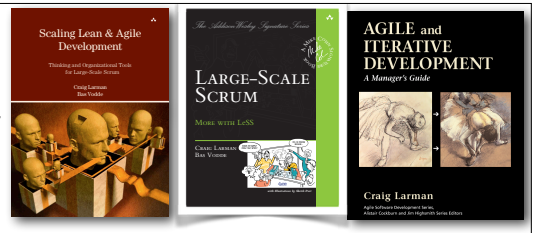
6

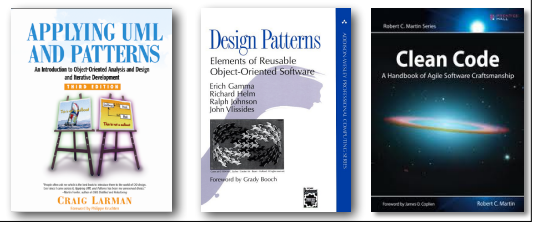
6

org agility

business agility

tech / design agility





7

7

Enabling Agility, **Technically**

- "Oh my god, i hope i don't have to touch that code!"
- low technical agility -> low business agility (and low velocity)

8

8

Practicalities

9

10

?:00

??:00

11

12

13

13

Course Overview

14

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

15

15

My Background

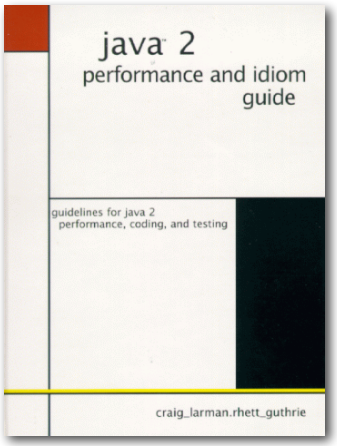
16

- lead coach of lean software development @ Xerox
- lots of "embedded" & legacy TDD, plus "clean" TDD

17

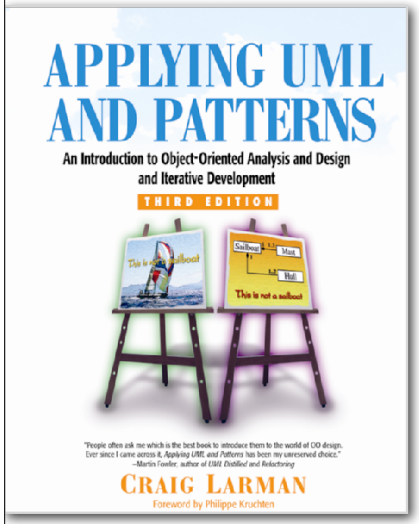
17

One of the First Chapters on TDD ('98)

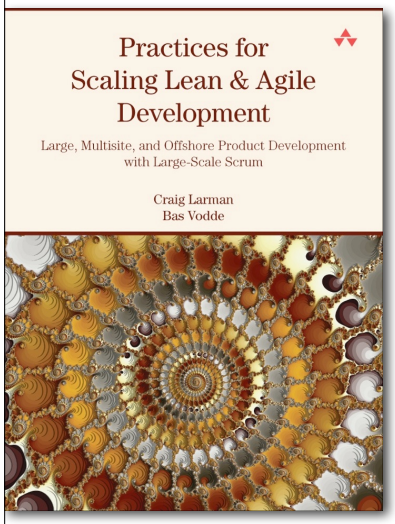


18

18

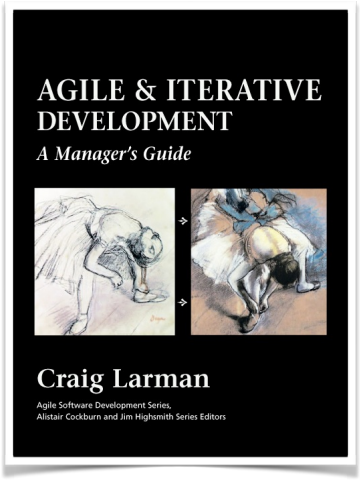


19



20

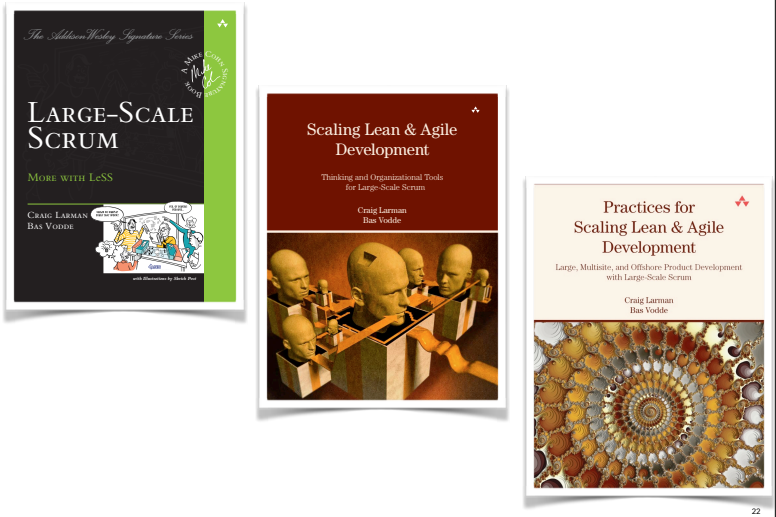
One of the First Agile books…



21

21

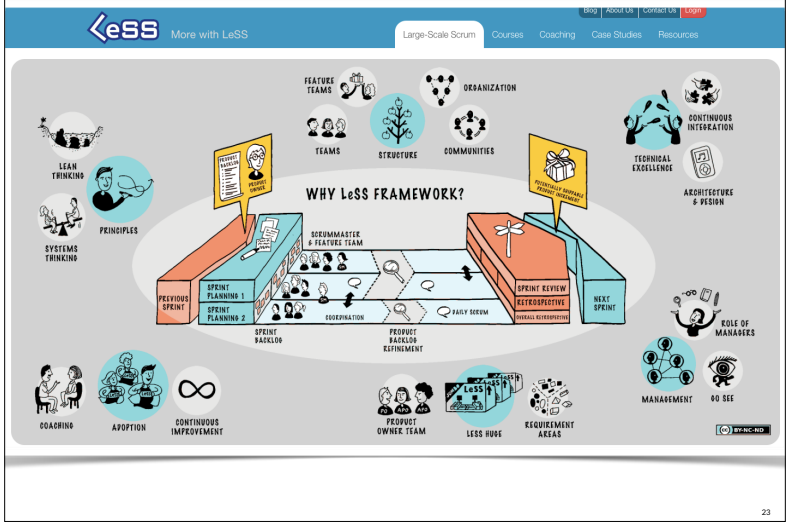
Focus on Scaling…



22

22

less.works



23

23

Introduction to Concepts & Terms

24

24

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

Why TDD?

Why?

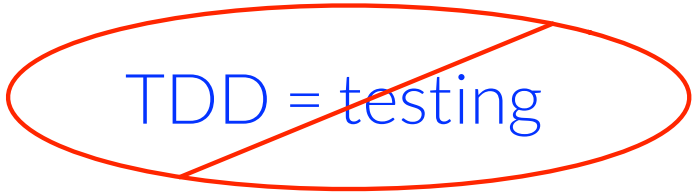
Introduction to TDD & ATDD/BDD

Misconception #1

TDD = testing

29

29



30

30

Tip!

- how to understand an idea?
  - **TDD, stories, agile, Scrum**
- start by analyzing the **name** carefully…
- … as intended by the **creator**

31

31

"Test-Driven Development"

"Test"?

"Driven"?

"Development"?

32

32

"**Test**-Driven Development"

"Test"?

33

33

"**Behavior**-Driven Development"

"Behavior"?

34

34

"**Test**-Driven Development"

when write "test" first, we are …

defining **executable specification**

defining **specification of behavior**

35

35

BIG Idea

Executable Specification!

36

36

Therefore…

Test-Driven Development

Acceptance-TDD or **Behavior**-Driven Development

37

37

BIG Idea

"test" =
"executatable spec" =
"behavior spec" =

38

38

"Test-**Driven** Development"

… "Driven"?

39

39

"Test-**Driven** Development"

"only write code
to fix a **failing** test"

40

40

why?

what are the consequences of writing code **not** driven by an executable spec?

41

41

BIG Idea

"only write code to fix a **failing** test"

42

42

BIG Idea

"driven" -> **failing** spec/test drives development

43

43

"Test-Driven **Development**"

... "Development"?

44

44

"Test-Driven **Development**"

1. create failing test/spec
2. **develop** *some* solution
3. **develop** a *solid* solution

45

Why Separate Steps 2 & 3?

1. create failing test/spec
2. **develop** *some* solution
3. **develop** a *solid* solution

How Does Step 1 **Aid** Step 3?

1. create failing test/spec
2. **develop** *some* solution
3. **develop** a *solid* solution

BIG Idea

"test-driven development" ->
is a **development method for 'good' code**

to summarize…

49

49

BIG Idea

Executable
Specification!

50

50

BIG Idea

"test" =
"executatable spec" =
"behavior spec" =

51

51

BIG Idea

"only write code
to fix a **failing** test"

52

52

BIG Idea

"driven" ->
failing spec/test drives development

53

53

BIG Idea

"test-driven development" ->
is a **development method for 'good' code**

54

54

pairs: standing: flipchart

- one person explain the phrase "test-driven development", word by word
  - do not use prior notes

55

55

Types/ Taxonomy

56

	source… Customer/Expert	Developer
scope… external & end to end	"acceptance" "ATDD", "BDD" outside-in	*what problems can arise?*
internal & less than end to end	*are they really a customer?*	"unit", etc "unit TDD", "TDD" inside out

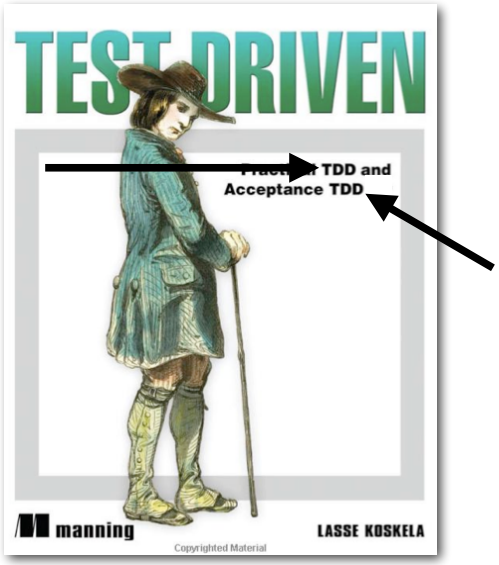
57

"Unit TDD" or just "TDD"

- it is useful to distinguish "unit TDD" from "BDD" or "ATDD" because…
- "unit TDD" includes
  - "unit" test isolation
  - different stakeholders
  - different development approach
    - e.g. inside out vs outside in

58

58



59

pairs: standing: flipchart

- the other partner draw the "quadrant" diagram and explain
  - do not use prior notes

60

60

The TDD Cycle

61

61

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

62

62

The TDD Cycle

63

Test-Last Development

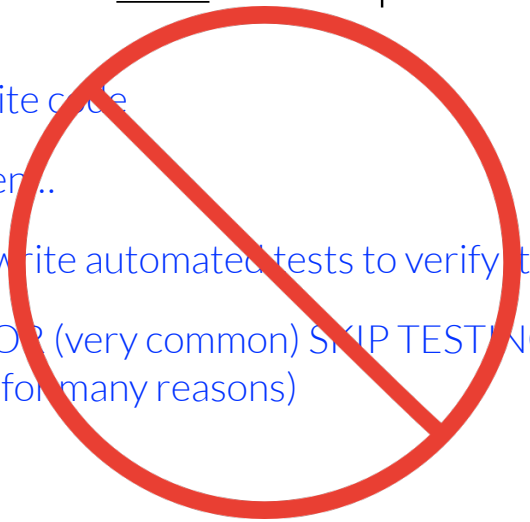
1. write code
2. then…
   1. write automated tests to verify it
   2. OR (very common) SKIP TESTING (for many reasons)

64

64

Test-Last Development

1. write code
2. then ...
   1. write automated tests to verify it
   2. OR (very common) SKIP TESTING (for many reasons)



65

65

Test-DRIVEN (First) Development

1. create executable specification ("test")

   it will fail, when executed
2. create some solution
3. create a good solution

66

66

TDD Cycle

1. Red – create one **specification**/test that can compile, run, and *fail*
2. Green – create a **solution**
3. Refactor – refactor to **solid** code (and test still green)

   –no code smells, clear, consistent, …

67

67

Summary

1. Red – specify
2. Green – solve
3. Refactor – solidify

68

68

RED
GREEN
REFACTOR

69

Example Spec/Test

70

Summary

1. **Red – specify**
2. Green – solve
3. Refactor – solidify

71

Example Executable Specification

```
public void roll_whenTypical() {

  // when "someDie.roll()" …

  // LAST STEP FIRST: specify & check
  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


72

BIG Idea

73

Driven to Define a Signature & Semantic

```
public void roll_whenTypical() {

  // when "someDie.roll()" …


  // LAST STEP FIRST: specify & check
  // we IMAGINE there is a getFaceValue() method


  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


74

TDD is a **Creative** Activity

```
public void roll_whenTypical() {

  // LAST STEP FIRST: specify & check
  // we IMAGINE there is a getFaceValue() method
  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


75

Driven to Define Supporting Elements

```
public void roll_whenTypical() {
  // what do we need to support & setup for the spec?
  // ???


  // LAST STEP FIRST: specify & check
  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


76

Driven to Define Supporting Elements

```
public void roll_whenTypical() {

  // 1. setup
  Die unit = new Die();


  // 2. call method under spec
  unit.roll();


  // 3. specify & check
  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


TDD is a **Creative** Activity

```
public void roll_whenTypical() {

  // 1. setup
  // we imagine there is a Die class and Die() ctor
  Die unit = new Die();


  // 2. call method under spec
  // we imagine there is a roll() method
  unit.roll();


  // 3. specify & check
  assertThat(unit.getFaceValue(),
        allOf(greaterThanOrEqualTo(1),
              lessThanOrEqualTo(6)));
}
```


Will it Compile, Run, & Fail?

1.Red – create one spec/test that can compile, run, & *fail*

2.Green – solve

3.Refactor – solidify

Example Failing

Summary

1. **Red – specify**
2. Green – solve
3. Refactor – solidify

81

Make it Compile, Run, & Fail

```
// … with LEAST EFFORT to compile & fail

class Die {
   public void roll() { }

   public int getFaceValue() { return -1; }
}
```


82

BIG Idea

during RED
apply LEAST EFFORT
to compile & fail

83

Example Solving

84

Summary

1. Red – specify
2. **Green – solve**
3. Refactor – solidify

85

Solve the Specification

```
// … *usually* with LEAST EFFORT to SOLVE

class Die {
   public void roll() { }

   public int getFaceValue() { return 3; }
}
```


86

BIG Idea

during GREEN
usually apply
LEAST EFFORT
to solve

87

Example Solidifying by Refactoring

88

Summary

1. Red – specify
2. Green – solve
3. **Refactor – solidify**

89

89

Solidify WHAT Code?

- both! …
  - specification/test
  - solution

90

90

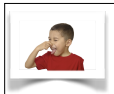
Solidify the Code?

- no code smells
  - duplication, magic numbers, …
- clear
- consistent
- …

91

91

Smelly



```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
            allOf(greaterThanOrEqualTo(1),
                  lessThanOrEqualTo(6)));
}
```

92 Craig Larman

92

Less Smelly

```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
            allOf(greaterThanOrEqualTo(Die.MIN),
                  lessThanOrEqualTo(Die.MAX)));
}
```


Better…

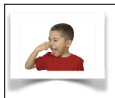
```
class Die {
   public static final int MAX = 6;
   public static final int MIN = 1;

   public void roll() { }

   public int getFaceValue() { return 3; }
}
```


But What is “3” Really Related To?

```
class Die {
   public static final int MAX = 6;
   public static final int MIN = 1;

   public void roll() { }

   public int getFaceValue() { return 3; }
}
```


Fragrant Solution!

```
class Die {
   public static final int MAX = 6;
   public static final int MIN = 1;

   public void roll() { }

   public int getFaceValue() { return MAX; }
}
```



Less Smelly, But What About…

```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
            allOf(greaterThanOrEqualTo(Die.MIN),
                  lessThanOrEqualTo(Die.MAX)));
}
```


Fragrant Specification!

```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
                  isBetween(Die.MIN, Die.MAX));
}
```


(Aside) Custom **Hamcrest Matcher**

```
import org.hamcrest.*;

public class IsBetween extends TypeSafeMatcher<Integer> {
    private int min;
    private int max;

    public IsBetween(int min, int max) {
        this.min = min;
        this.max= max;
    }

    @Override
    public boolean matchesSafely(Integer num) {
        return num >= min && num <= max;
    }

   //...
```


(Aside) Custom Hamcrest Matcher

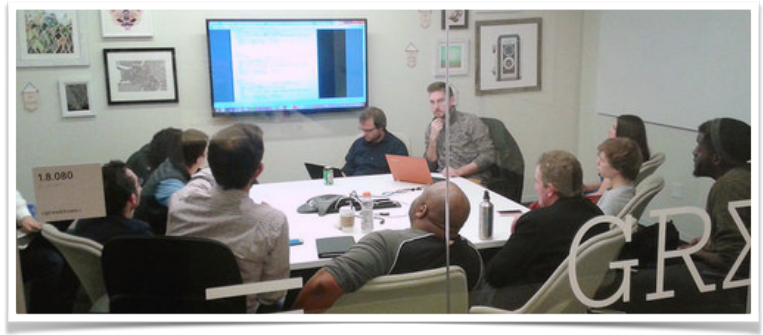
```
// ...

@Override
 public void describeMismatchSafely(
      Integer num, Description mismatchDescription) {
   mismatchDescription.appendValue(num)
                      .appendText(" was not between ")
                      .appendValue(min)
                      .appendText(" and ")
                      .appendValue(max);
 }

 @Override
 public void describeTo(Description description) {
     description.appendText("a numeric value between ")
                .appendValue(min)
                .appendText(" and ")
                .appendValue(max);
 }
```

(Aside) Custom Hamcrest Matcher

```
    // ...

    public static Matcher<Integer>
           isBetween(int min, int max) {
        return new IsBetween(min, max);
    }

} // end of class
```


Exercise

Mob Programming



Start With…

```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
            allOf(greaterThanOrEqualTo(Die.MIN),
                  lessThanOrEqualTo(Die.MAX)));
}
```

Start With…

```
class Die {
   public static final int MAX = 6;
   public static final int MIN = 1;

   public void roll() { }

   public int getFaceValue() { return MAX; }
}
```


mob:

- evolve the starting code, according to the requirements of the coach

coach:

- debrief
  - **TDD** learnings?
  - **code/design** learnings?
  - **style** learnings
  - **tool** learnings?

A Few TDD Basic Guidelines

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

109

BIG Idea

write code for the developer **7 years** in the future

Guideline

- (probably) don't test auto-generated methods, such as getters/setters

Guideline

- use a standard variable name for your instance under development; e.g., **unit**

Guideline

- don't forget to test-drive the creation of initialization/ construction (e.g. constructors)

Guideline

- use test names to communicate purpose. e.g.
  - **–[methodName]_when[Situation]**
  - –void constructor_whenDefault()
  - –void constructor_whenTypicalParameterValues()
  - –void addAccount_whenTypical()
  - –void addAccount_whenBadName()

Guideline

- fine-grained high-cohesion tests; 1 per logic path. "test one thing"
- test all logic paths, including failure paths

Guideline

- **class Banana extends Fruit { ... }**
- to test-drive class *Fruit*, create a *FruitTest* class, but test it with an instance of Banana (since Fruit is **abstract**)

```
class FruitTest {
  Fruit unit = new Banana();
  ...
```

Clean Code & Emergent Micro-Design

117

117

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

118

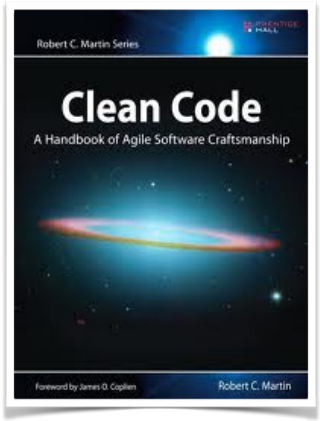
118

From Dirty to Clean Code

119

119

Refactoring to Clean Code



120

120

Dirty Smelly Code?

- duplicate code or data
- large in-cohesive method
- large class
- code blocks of separate sub-steps
- low-level math/string expressions
- unclear names
- magic constants
- high coupling... & many specializations (e.g., "data envy")
- low cohesion... & many specializations (e.g., long method)
- dead code
- long parameter list
- weak encapsulation, i.e., exposed implementation
- case logic rather than polymorphism
- non-conformance to coding standards
- data object (record object)
- comments that explain *what*
- non-idiomatic use language/ libraries ("C++ in Java")
- "i have to think, to understand the code"

121

Some Key Clean Code Guides

- DRY (don't repeat yourself)
- intention revealing — don't make me think
- high cohesion — short & single responsibility

122

Emerging Micro-Design

123

So Far, TDD Supports…

- discover **specifications**
- create **solid clean code**

124

but there's other things TDD supports...

125

125

emerging micro-design

(we've already seen this in our prior exercises)

126

126

Emerging Micro-Design in TDD

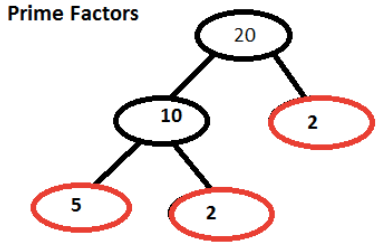
- as the specs/tests grow & become **more specific**...
- ... emerging design (should) become **more general**

127

127

TDD & Emerging Micro-Design

- good for **small** design problems



128

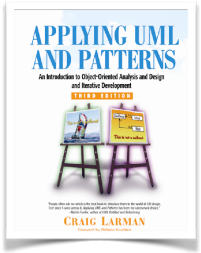
TDD & Emerging Micro-Design

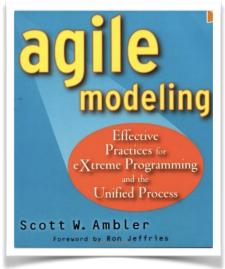
- *maybe* **not** for **big** design/ architectural problems

129

129

Alternatives for Large-Scale Design





130

130

BIG Idea

TDD supports:
(1) discover specifications
(2) create solid clean code
(3) emerging micro-design

131

131

Exercise

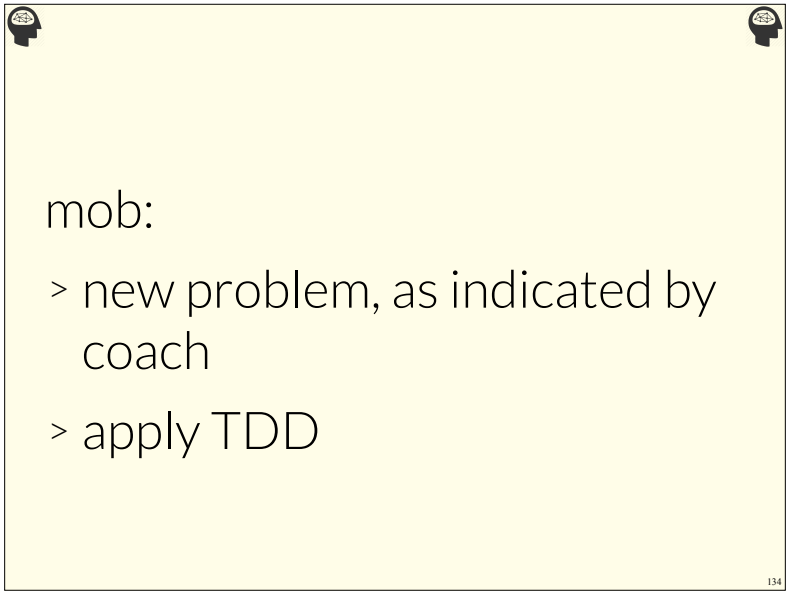
132

Parameterized Tests

- might be relevant to next exercise
- does our xUnit framework support them?

133

133



134

135

136

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

137

So Far, TDD Supports…

- discover **specifications**
- create **solid clean code**
- **emerging micro-design**

138

but there's other things TDD supports…

139

Better Design with Low Coupling

- Is it testable in isolation?
- We will be forced to design with lower coupling, and with "dependency injection."
- Designing with lower coupling is a key quality of good design. It leads to many short and long-term benefits

140

Satisfaction for Sustained Practice

- a **psychological** factor
- creative, puzzle/challenge versus a boring low-creative last step

141

Test-**First** means **Test-Done**

- test-last -> test-little or test-never
- "we don't have time"
- "it's boring, ..."

142

Test-**First** means **Clean-Code-Done**

- test-last -> little or no refactoring to clean code
- "we don't have time"
- "we could break something" (since there's no tests)

143

Executable Specs = **Living Documentation**

```
public void roll_whenTypical() {
      Die unit = new Die();

      unit.roll();

      assertThat(unit.getFaceValue(),
                  isBetween(Die.MIN, Die.MAX));
}
```

144 Craig Larman

BIG Idea

TDD supports:

(1) discover specifications
(2) create solid clean code
(3) emerging micro-design
(4) better design
(5) satisfaction & sustained
(6) "tests" done
(7) clean code done
(8) living documentation

145

145

individual:

- review deeper motivations & dynamics of TDD

146

146

pairs:

- with a talking partner, stand up and answer this scenario:
  - Someone comes to you and says, "**I don't think we should be writing so many little tiny tests**."
  - or they ask, "**What did you learn in the *testing* course**?"
  - How do you respond?
- reverse roles, and repeat

147

147

Test Doubles & Dependency Injection

148

148

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

149

149

Test Doubles

150


coach: discuss

- Suppose doing TDD for class X and you create a dependency on a class Y… and class Y has one or more of these qualities:
  - **"non-deterministic"**
  - **created by a third party**
  - **not created yet**
  - **slow — interacts with DB, network, etc.**
- Question: How design the coupling or dependency from X to Y to support TDD for class X?

151

151

"**Test Doubles**"

- when?
  - "non-deterministic"
  - created by a third party
  - not created yet
  - slow — interacts with DB, network, ...

152

152

Test Doubles

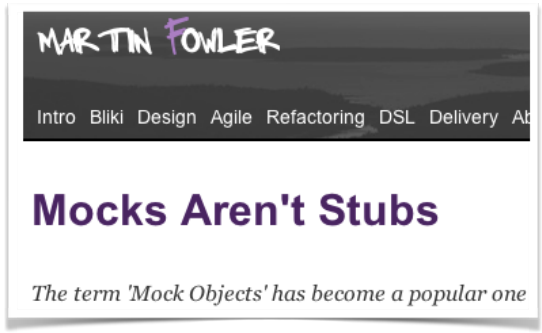
- when **NOT**?
  - **data object**
  - **in-memory** "official" library class that doesn't "launch a rocket": *Math*, *String*, *List*
  - **in-memory** "*simple*" in-house class that doesn't "launch a rocket": *OurStringUtil*

153

153

Common Terminology Mistake

- Mocks aren't Test Doubles

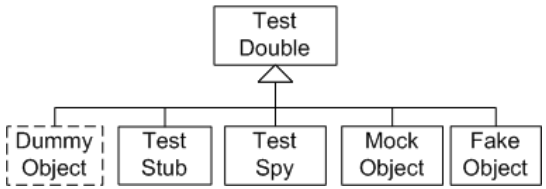


154

154

Common Terminology Mistake

- widespread ignorance of correct meaning of "mock"
- many kinds of "test doubles"…

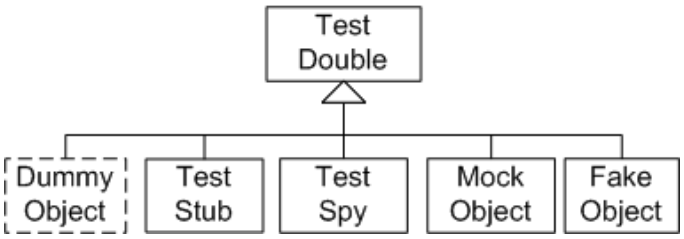


155

155

Test Double: **Fakes**

- the most common useful test double is a **FAKE**, not a mock



156

156

Manual Creation of Test Doubles

- object-oriented polymorphism
  - –common interface + alternate "test double" implementation
  - –subclass real implementation
- lambda expressions
- templates (generics)
- C and/or C++
  - –alternate implementation, same signature
  - –functor

157

157

Tool Creation of Test Doubles

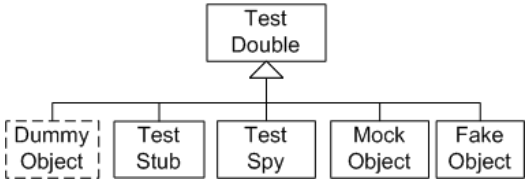
- JMockit
- NMock
- Google Mock
- CPPUTest
- …

158

158

Common Terminology Mistake

- why? because "*mocking* frameworks" are used to create **fakes**, **stubs**, (true) **mocks**, etc



159

159

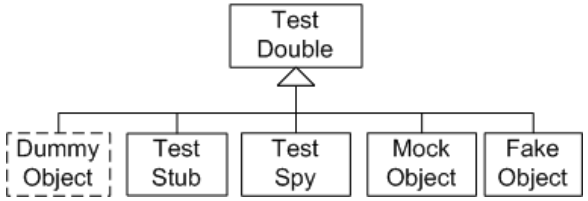
Dependency Injection

160

160

"**Dependency Injection**"

- inject an alternate "test double" implementation for the dependent object



161

161

Manual: How Inject a Dependent?

- constructor parameters
- setter parameters
- direct assignment to attributes
- factories
- AOP
- C and/or C++ specific:
  - –preprocessor
  - –linker
  - –function pointer assignment

162

162

Tool: How Inject a Dependent?

- tool support:
  - "mocking" frameworks (JMockit)
  - "dependency injection framework" (Spring)
- these tools use "magic" that injects via one or more of:
  - constructor & setter parameters
  - direct assignment to attributes
  - factories
  - VM instrumentation
  - AOP

163

163

State-Based Tests
with Fakes
versus
Interaction-Based Tests
with Mocks

164

State-Based Tests & Fakes

- for feature "delete all files in a directory"

Interaction-Based Tests & Mocks

- for feature "send message to external market exchange"

Interaction-Based Tests & Mocks

- WARNING! **fragile**
- why?

Guideline

- 95%: **state-based** testing with **Fakes**
- 5%: **interaction-based** testing with **Mocks**
  - WARNING! **fragile**

Exercise

169

mob or individual:

- ensure the “test double” “mocking” framework is installed and accessible

170

170

mob:

- new problem to illustrate Test Doubles and Dependency Injection, as indicated by coach
- use tool to create a **fake** and do **state-based testing**
- apply TDD

171

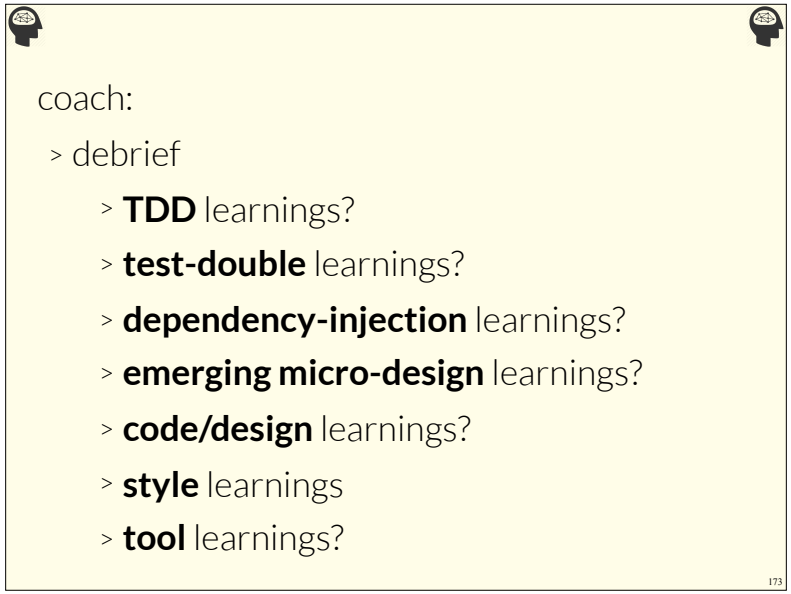
171

mob:

- new problem to illustrate Test Doubles and Dependency Injection, as indicated by coach
- use tool to create a **mock** and do **interaction-based testing**
- apply TDD

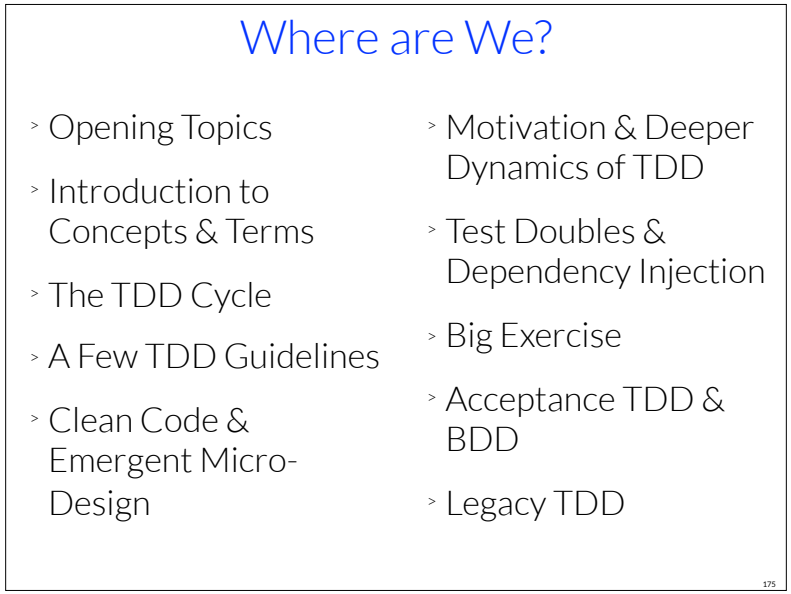
172

172



173

174



175

176

mob:

> new product!

> we will start with mob programming, but transition to pair-programming later

177

177

pairs:

> prepare for pair-programming by copying the code

178

178

TDD with Pair Programming

1. Red – person 1
2. Green – person 2
3. Refactor – person 1
4. Red – person 2
5. Green – person 1
6. Refactor – person 2
7. ...

179

179

pair:

> apply pair-programming with TDD to the product

> we will do all-together code reviews regularly

180

180

coach:

- debrief
  - **TDD** learnings?
  - **test-double** learnings?
  - **dependency-injection** learnings?
  - **emerging micro-design** learnings?
  - **code/design** learnings?
  - **style** learnings
  - **tool** learnings?

181

181

Acceptance TDD & BDD

182

182

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

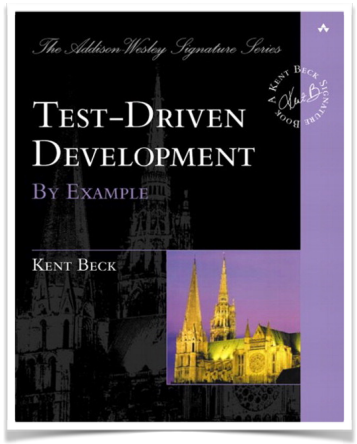
183

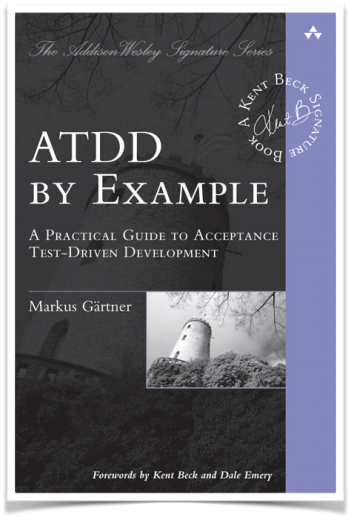
183

	source… Customer/Expert	Developer
scope… external & end to end	"acceptance" "ATDD", "BDD" outside-in	*what problems can arise?*
internal & less than end to end	*are they really a customer?*	"unit", etc "unit TDD", "TDD" inside out

184

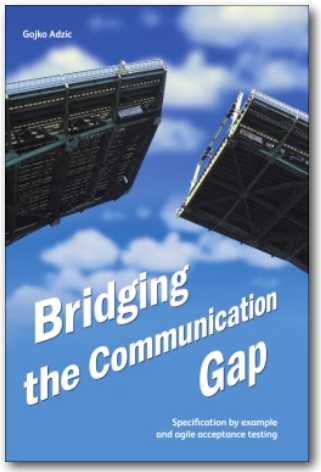
184

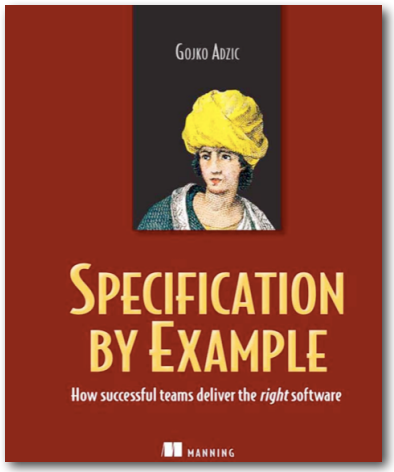




185

Some of the Best...

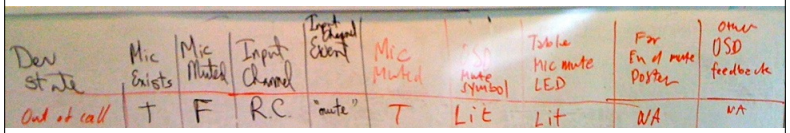




186

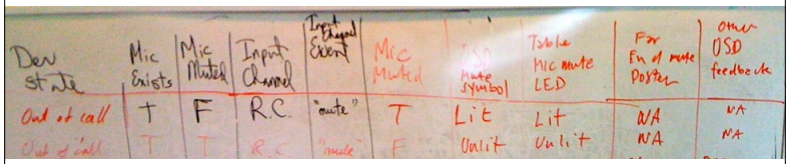
186

Specification by Example



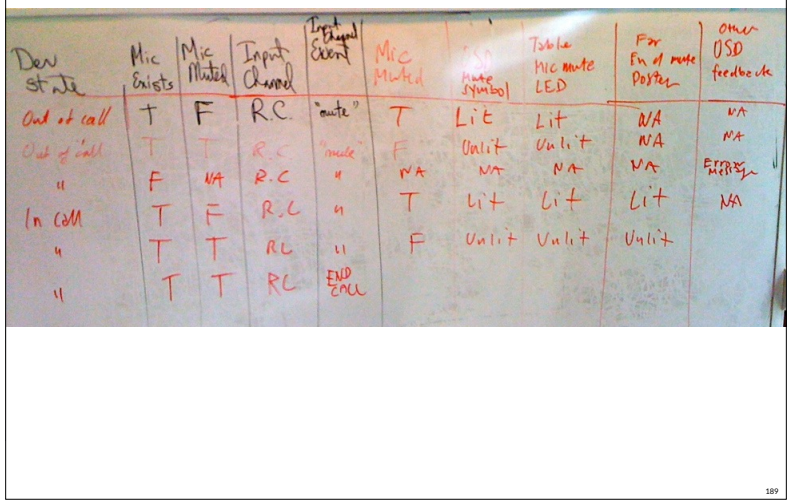
187

187



188

188



189

Specification by Example

specification with examples (real case)

					Order Details						RefData			
currency pair	fx type	Verb	Amount	Dealt	Markup 1	Markup 2	Client Value	Value	Markup Type	Pip Fmt	Success	Total Markup	Wholesale Value	Profit
EURUSD	Spot	Buy	1,000,000	EUR	2		1.30	0.00	Pts	4	OK	2	1.3002	200.00
USDJPY	Spot	Sell	2,000,000	USD	10		110.00	0.00	Pts	2	OK	10	109.9	200,000.00
GBPUSD	Fwd	Buy	1,000,000	GBP	1	3	1.50	0.00	Pts	4	OK	4	1.5004	400.00
GBPUSD	Fwd	Sell	1,000,000	GBP	1	3	1.50	0.00	Pts	4	OK	4	1.4996	400.00
GBPUSD	Fwd	Sell	1,000,000	GBP	1	0	1.50	0.00	Pts	4	OK	1	1.4999	100.00

executable specifications “without change”, with automated validation

Action	currency pair	fx type	Verb	Amount	Dealt	Markup 1	Markup 2	Client Value	Value	Markup Type	Error Code	Total Markup	Wholesale Value	Profit
Verify Profit Details For Order	EURUSD	Spot	Buy	1000000	EUR	2.0	0.0	1.30	0.00	Pts	0	2.0	1.3002	200.00
Verify Profit Details For Order	USDJPY	Spot	Sell	2000000	USD	10.0	0.0	110.0	0.00	Pts	0	10.0	109.9	200000.00
Verify Profit Details For Order	GBPUSD	Fwd	Buy	1000000	GBP	1.0	3.0	1.5	0.00	Pts	0	4.0	1.5004	400.00
Verify Profit Details For Order	GBPUSD	Fwd	Sell	1000000	GBP	1.0	3.0	1.5	0.00	Pts	0	4.0	1.4996	400.00
Verify Profit Details For Order	GBPUSD	Fwd	Sell	1000000	GBP	1.0	0.0	1.5	0.00	Pts	0	1.0	1.4999	100.00

190

190

BIG Idea

the language/ representation of the *example* specifications & *executable* specifications should be “without change”

191

191

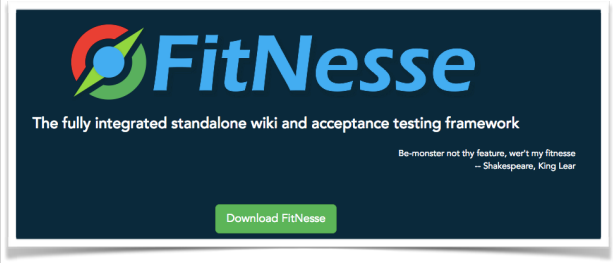
Acceptance TDD Cycle

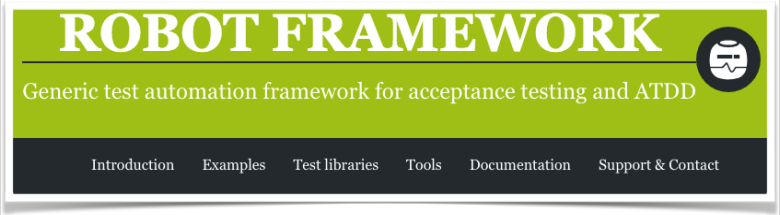
1. Red – for the next customer feature, transform *all* the specifications by example into *all* the automated acceptance tests (1-3 HOURS)
2. Green – implement feature, so that all the acceptance tests pass (1-3 DAYS)

192

192

Tools: Fitnesse & Robot FW





193

193

Legacy TDD

194

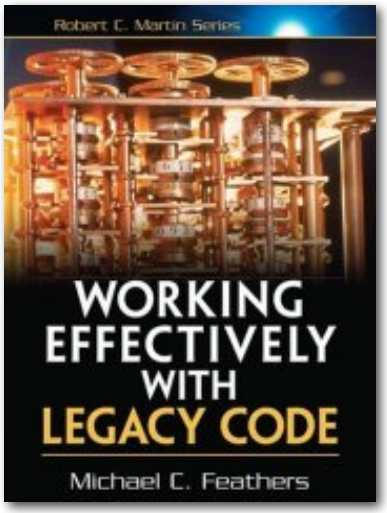
194

Where are We?

- Opening Topics
- Introduction to Concepts & Terms
- The TDD Cycle
- A Few TDD Guidelines
- Clean Code & Emergent Micro-Design
- Motivation & Deeper Dynamics of TDD
- Test Doubles & Dependency Injection
- Big Exercise
- Acceptance TDD & BDD
- Legacy TDD

195

195



196

Legacy-Code TDD cycle

1. Add test(s) for existing method, that **passes**.
   - These are **characterization tests**. Implies dependency breaking
2. Add test (or modify existing characterization test) for existing method, that **fails** — reflecting the new desired outcome (that is not yet implemented)
3. Add solution to make the failing test pass
4. Refactor

197

197

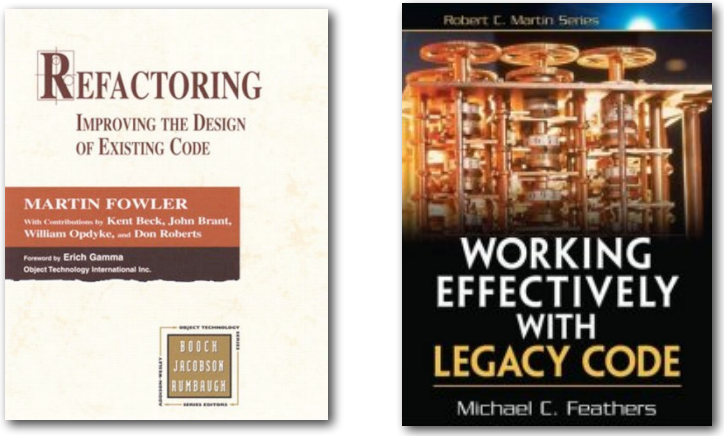
Closing

198

198

199

199



200

200

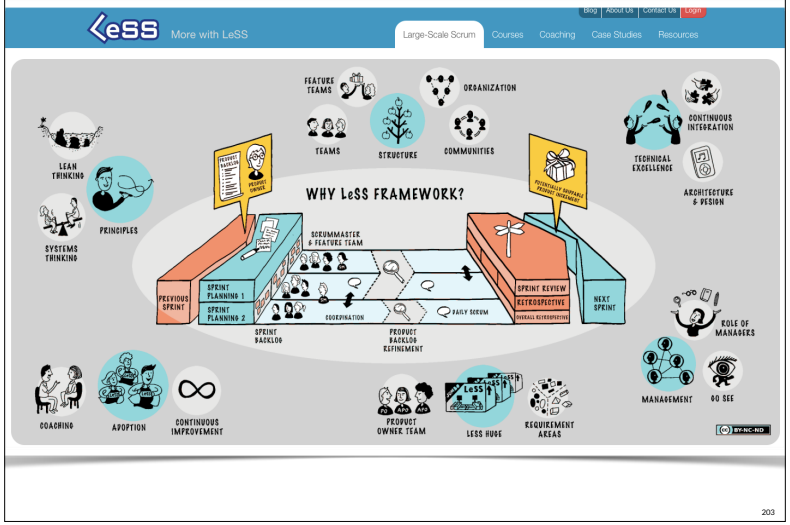
201

share
blog
tweet
spread the word

202

202

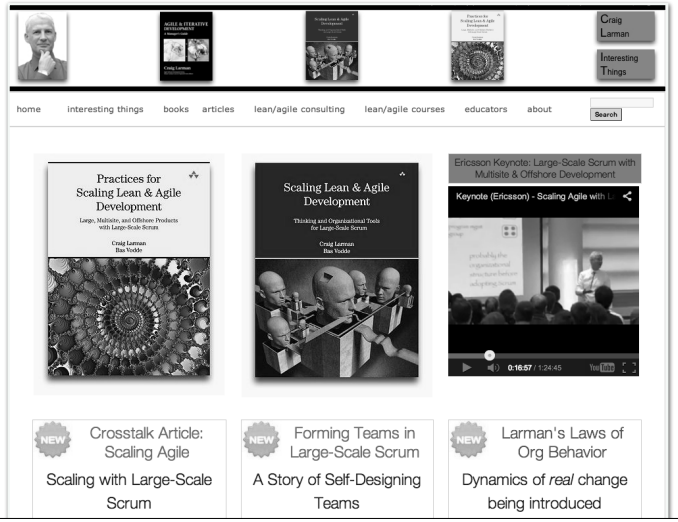
less.works



203

203

craiglarman.com



204

204