Introduction to Test-Driven Development

Mike long - CEO of Kosli
@meekrosoft



About me

Mike Long

CEO @ kosli.com

People enthusiast. Nemophilist.

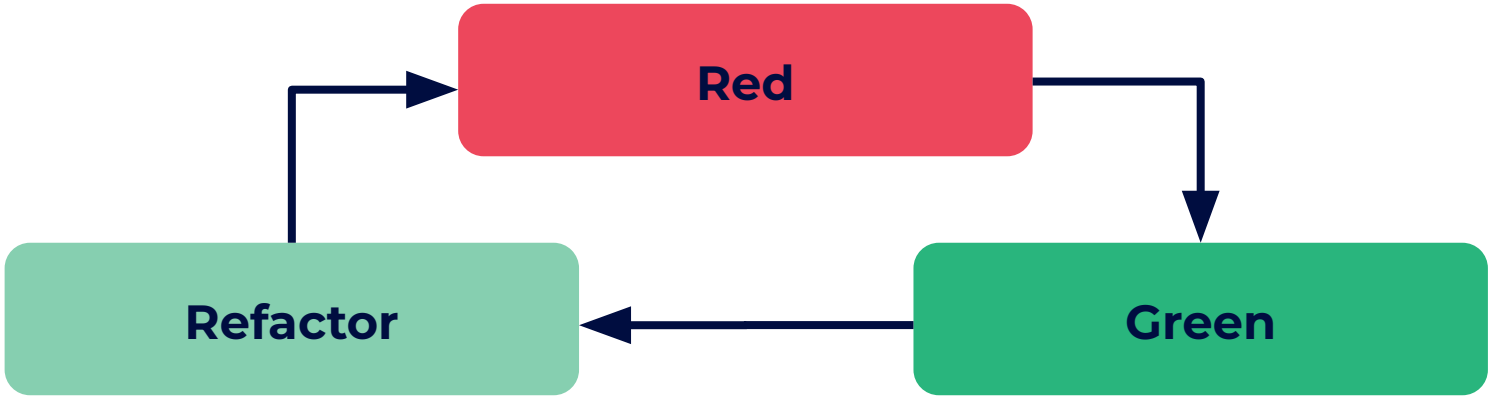
@meekrosoft



Programming skills



Test-Driven Development



Why TDD?

Refactoring support: Make changes with confidence

Design verification: the code does what you think it does



Design benefits: isolated units should have low coupling & high cohesion

Work incrementally: Share your changes often, maintain flow

Wonderful feeling of freedom and productivity

Kent Beck in 2009

The business problem with Max is that there just aren't enough of us who rely on tests minute-by-minute. Even allowing for my marketing and sales mistakes, the data suggests that there are at most a few thousand Java programmers actively practicing TDD. That's not a large enough market to sustain a business, especially as the market turned out to be price sensitive. Even if every Max subscriber successfully convinced 100 of their friends to sign up, there wouldn't have been enough revenue.

Learning to ski



flickr user nonanet

Cross-country skiing



Snowplow



flickr user michaeli

Parallel Turns



flickr user nonanet

Training Course



https://www.flickr.com/photos/fboyd/

Pair Programming



image: flickr user Lisamarie Babik

The Coding Dojo

Dojo = Place you go to learn



A Regular Coding Dojo



Dojo Principles

The first rule of the dojo:

- You can’t discuss a technique without code
- You can’t show code without tests

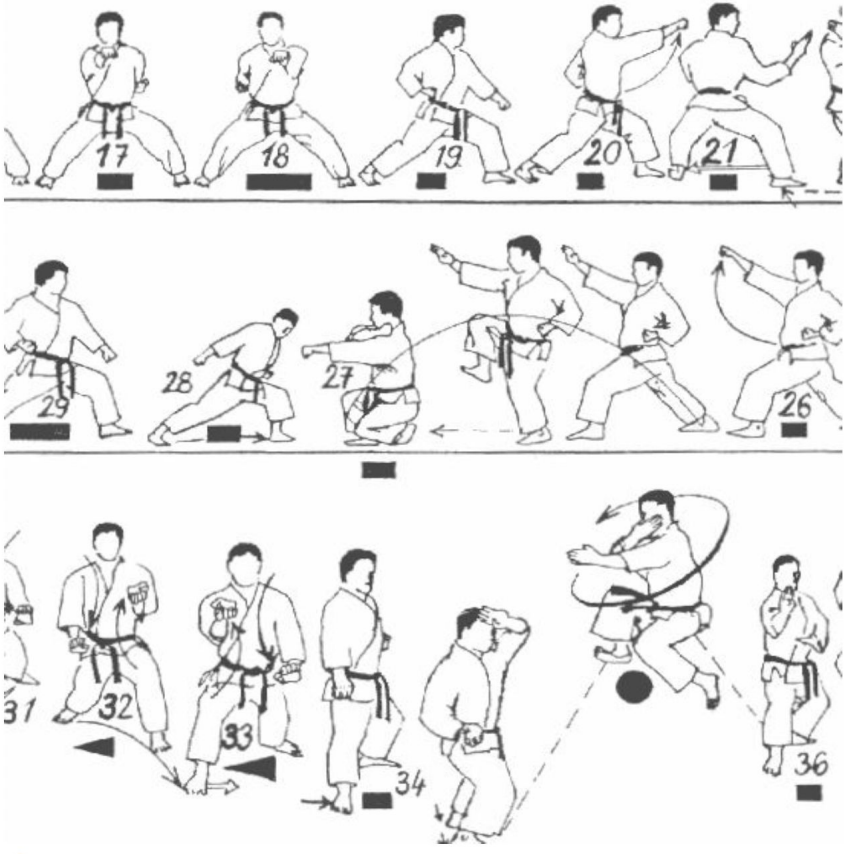


Code without tests simply doesn’t exist!

Code Kata

The first rule of the dojo:

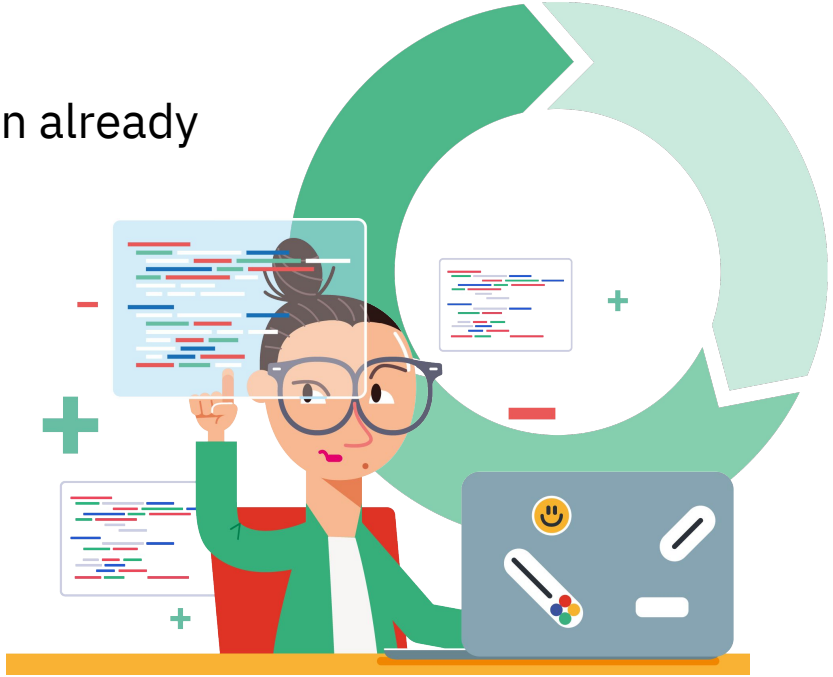
- In martial arts, a “Kata” is a sequence of moves that you learn.
- Dave Thomas proposed the idea of the “Code Kata”
- http://codekata.pragprog.com/
- small exercises, that you repeat



Two Kinds of Practice

Incidental Practice

- Repeatedly doing something you can already
- do, and improving at it



Incidental Practice: Habits

“I’m not a great programmer; I’m just a good programmer with great habits.”

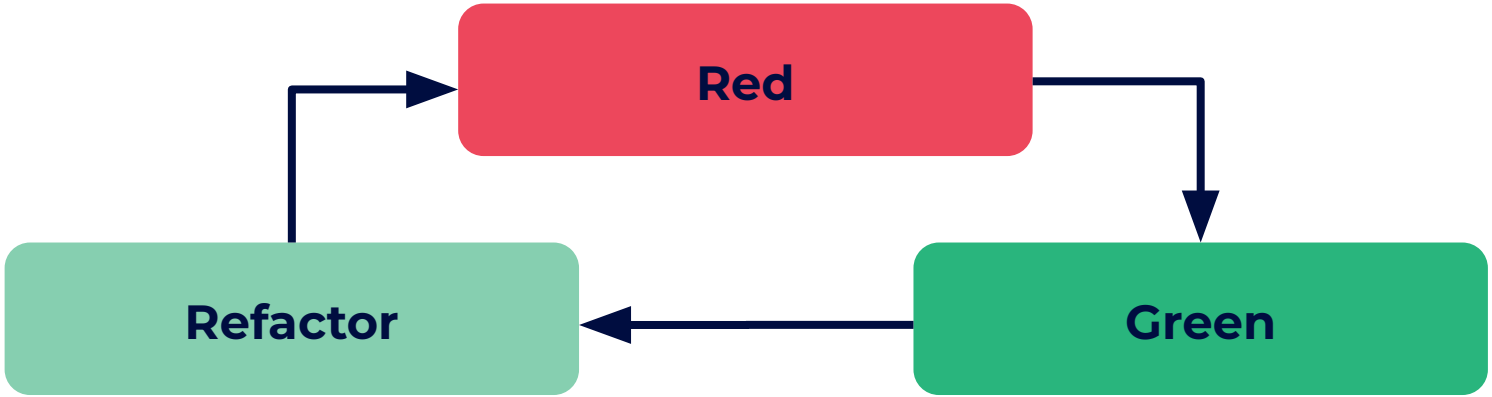
- Kent Beck

quote from p57, “Refactoring” by Martin Fowler



image taken by Martin Fowler at XP2002

Experience TDD



- Experience what TDD feels like when it works
- Recognise problems well suited to it

Two Kinds of Practice

Incidental Practice

- Repeatedly doing something you can already
- do, and improving at it

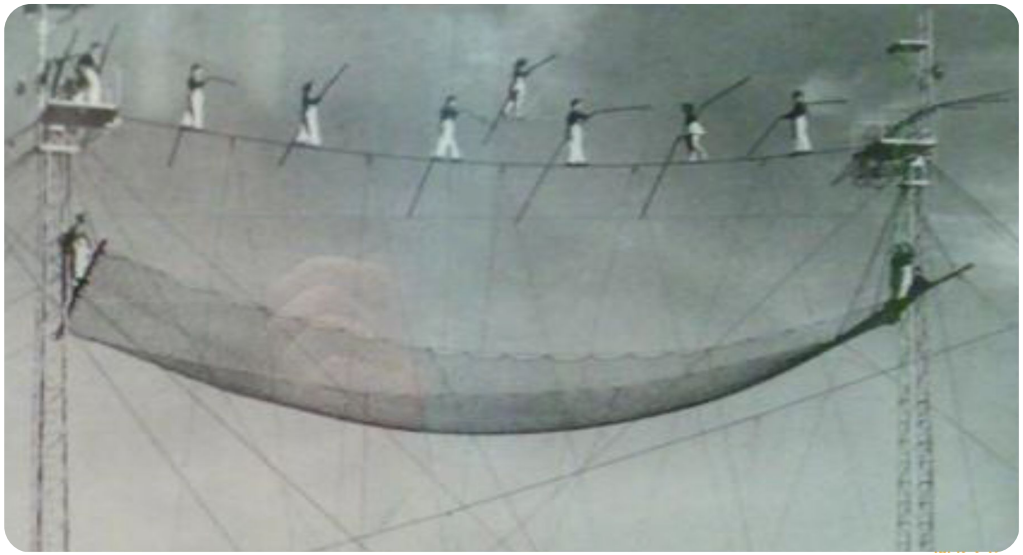


Deliberate Practice

- Trying to do something you can't comfortably do
- Breaking down a skill into components you
- practice separately

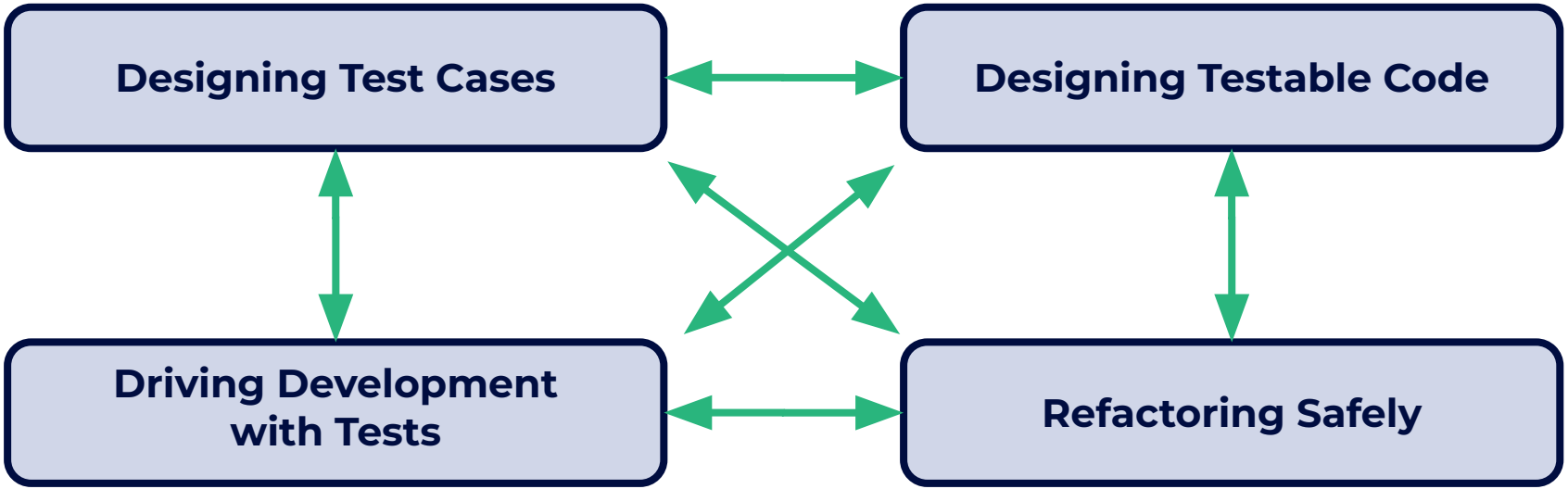
Deliberate Practice

- Need to feel safe
- Need to feel motivated



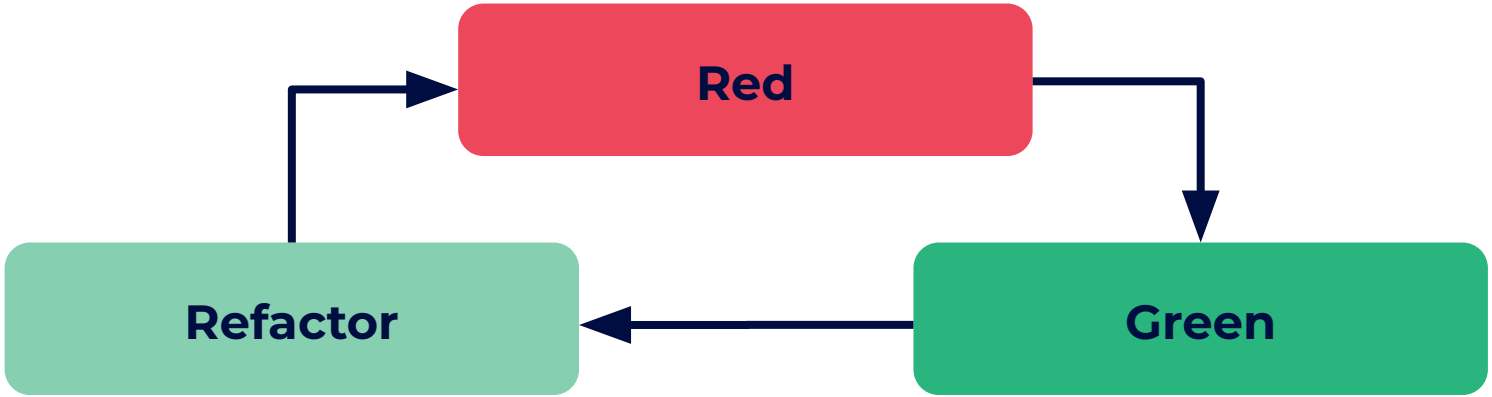
flickr user antony_mayfield

TDD Skills



TDD is a composite skill

Summary: learning TDD



- TDD is not easy to learn, but it is worth it
- You will need to practice, this course is not enough

Exercise: Leap Years

Driving Development with Tests



Mike Long @meekrosoft

kosli | kosli.com

Unit Test Design

Readability is key



Mike Long @meekrosoft

kosli | kosli.com

Self-testing code

"You have self-testing code when you can **run a series of automated tests** against the code base and be **confident** that, should the tests pass, your code is **free of any substantial defects**"

Martin Fowler

Tests Give Confidence in your Code

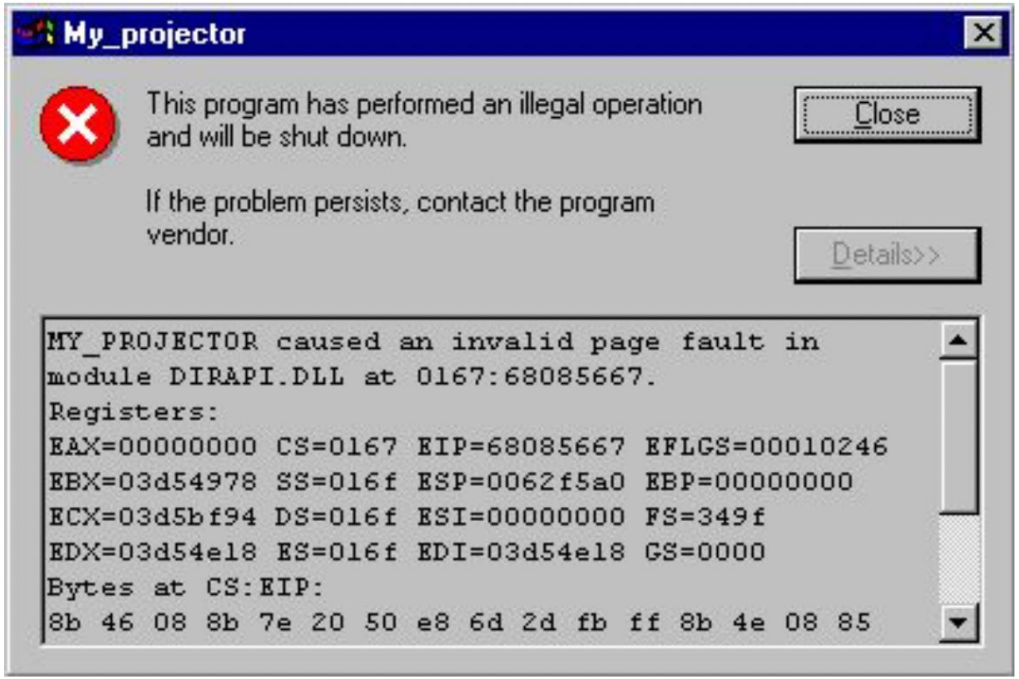
- Does it do what I expected?
- Did I understand the requirements well enough?
- Do I handle error cases: bad input, bad collaborators?

Tests give Feedback on your Design

- Is the interface easy to use?
- Are you missing any collaborators?

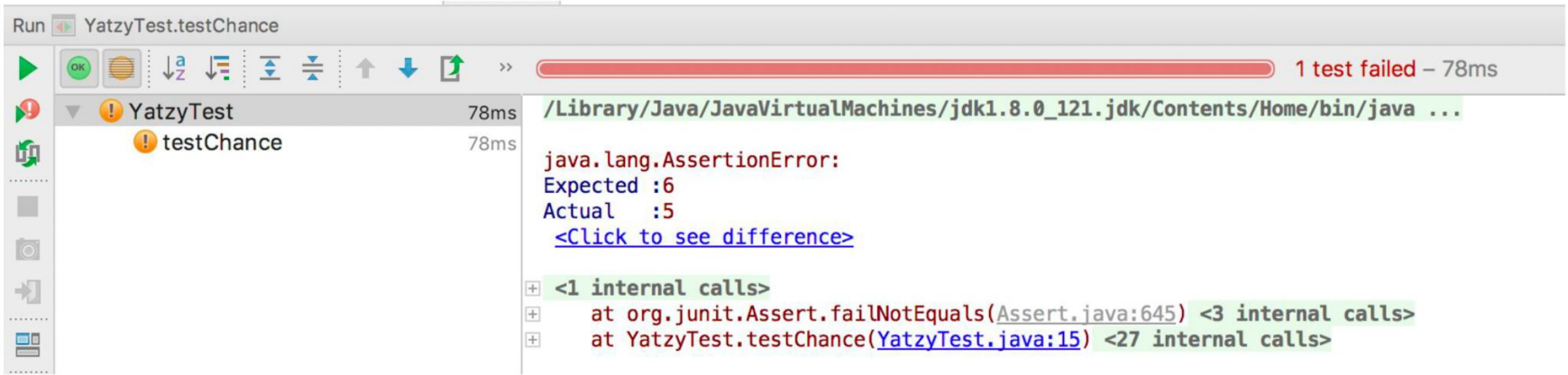
```
10  public class YatzyTest {
11      @Test
12      public void testChance() {
13          List<Integer> dice = Arrays.asList(1,1,1,1,1);
14          int score = new Yatzy1().score(dice, YatzyCategory.chance);
15          assertEquals( expected: 5, score);
16      }
17  }
```

What if they fail?



Ideally you know why!

What failed?

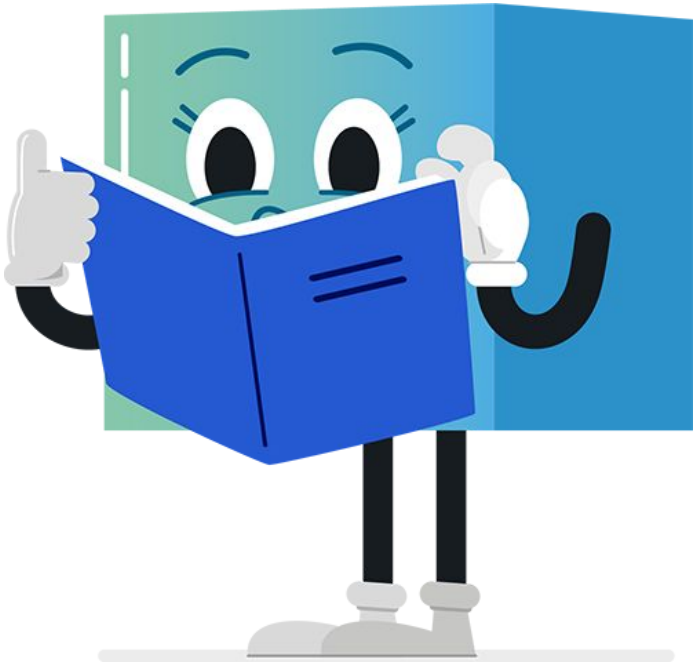

Run YatzyTest.testChance
1 test failed – 78ms
YatzyTest 78ms
testChance 78ms
/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java ...
java.lang.AssertionError:
Expected :6
Actual :5
<Click to see difference>
<1 internal calls>
at org.junit.Assert.failNotEquals(Assert.java:645) <3 internal calls>
at YatzyTest.testChance(YatzyTest.java:15) <27 internal calls>

What failed?

```
10  public class YatzyTest {
11      @Test
12      public void testChance() {
13          List<Integer> dice = Arrays.asList(1,1,1,1,1);
14          int score = new Yatzy1().score(dice, YatzyCategory.chance);
15          assertEquals( expected: 6, score);
16      }
17  }
```

Readability

Optimize for the person who needs to understand the failure



Test Names

What is going to give the best explanation when it fails?

```
10  public class YatzyTest {
11
12      @Test
13      public void test1() {...}
18
19      @Test
20      public void testChance() {...}
25
26      @Test
27      public void chanceSumsTheDice() {...}
32
33      @Test
34      public void score_Chance_Sums_The_Dice() {...}
39
40      @Test
41      public void given_all_ones_score_chance_returns_5() {...}
46
47      @Test
48      public void any_dice_score_chance_gives_sum() {...}
53
```

My preferences for Test naming

- Longer is usually better, use “snake_case” or “CamelCase” or both
- test_MethodName_Args_Gives_Result()

```
54      @Test
55      public void score_any_dice_chance_gives_sum() {...}
60
```

BDD-style frameworks

```
public void <method>With<args>Returns<X>()

public void <method>With<args2>Returns<Y>()
```



```
describe <methodname>
  with context <args>
    it should return <X>
  with context <args2>
    it should do <Y>
```

3 parts of the test

```
10  public class YatzyTest {
11      @Test
12      public void testChance() {
13          List<Integer> dice = Arrays.asList(1,1,1,1,1);
14          int score = new Yatzy1().score(dice, YatzyCategory.chance);
15          assertEquals( expected: 5, score);
16      }
17  }
```

Line 13 : Arrange

Line 14: Act

Line 15: Assert

Rule of thumb: One assert per test

- One reason for the test to fail
- No masked failures

```
TEST(Yatzy, "Playing A Game")
{
    Yatzy *yatzy = new Yatzy();
    int[] roll1 = new int[3,4,4,5,6];
    ASSERT_EQ(8, yatzy->score(roll, PAIR));
    int[] roll2 = new int[3,3,4,5,5];
    ASSERT_EQ(16, yatzy->score(roll, TWO_PAIRS));
    int[] roll3 = new int[1,2,3,4,6])
    ASSERT_EQ(0, yatzy->score(roll3, SMALL_STRAIGHT);
    ASSERT_EQ(24, yatzy->final_score())
}
```

Duplication

```
public void chanceSumsTheDice() {
    Yatzy1 calculator = new Yatzy1();
    List<Integer> dice = Arrays.asList(1,1,1,1,1);
    int score = calculator.score(dice, YatzyCategory.chance);
    assertEquals( expected: 5, score);
}

@Test
public void yatzyWithSameDiceGives50() {
    Yatzy1 calculator = new Yatzy1();
    List<Integer> dice = Arrays.asList(1,1,1,1,1);
    int score = calculator.score(dice, YatzyCategory.yatzy);
    assertEquals( expected: 50, score);
}

@Test
public void yatzyWithDifferentDiceGives0() {
    Yatzy1 calculator = new Yatzy1();
    List<Integer> dice = Arrays.asList(1,1,1,1,2);
    int score = calculator.score(dice, YatzyCategory.yatzy);
    assertEquals( expected: 0, score);
}
```

Duplication

```
@Before
public void setUp() {
    calculator = new Yatzy1();
}

private void assertScore(int expected, YatzyCategory category, Integer... dice) {
    assertEquals(expected, calculator.score(Arrays.asList(dice), category));
}

@Test
public void chanceSumsTheDice() {
    assertScore( expected: 5, YatzyCategory.chance,  ...dice: 1,1,1,1,1);
}

@Test
public void yatzyWithSameDiceGives50() {
    assertScore( expected: 50, YatzyCategory.yatzy,  ...dice: 1,1,1,1,1);
}

@Test
public void yatzyWithDifferentDiceGives0() {
    assertScore( expected: 0, YatzyCategory.yatzy,  ...dice: 1,1,1,1,2);
}
```

Parameterized Tests

```
@Parameters
public static Collection<Object[]> getAllScores() {
    return Arrays.asList(new Object[][] {
            { 0, 0, "Love-All" },
            { 1, 1, "Fifteen-All" },
            { 2, 2, "Thirty-All"},
            { 3, 3, "Forty-All"},
            { 4, 4, "Deuce"},

            { 1, 0, "Fifteen-Love"},
            { 0, 1, "Love-Fifteen"},
            { 2, 0, "Thirty-Love"},
            { 0, 2, "Love-Thirty"},
            { 3, 0, "Forty-Love"},
            { 0, 3, "Love-Forty"},
            { 4, 0, "Win for player1"},
            { 0, 4, "Win for player2"},

            { 2, 1, "Thirty-Fifteen"},
            { 1, 2, "Fifteen-Thirty"},
```

Tests give Refactoring support





- Test via Public API, black-box style
- Invest in design for your test cases too

Testing Private Methods

- Test it via a public method that calls it
- Change it package private/protected
- Move it to another class where it is public
- Buy a powertool that modifies the compiled code

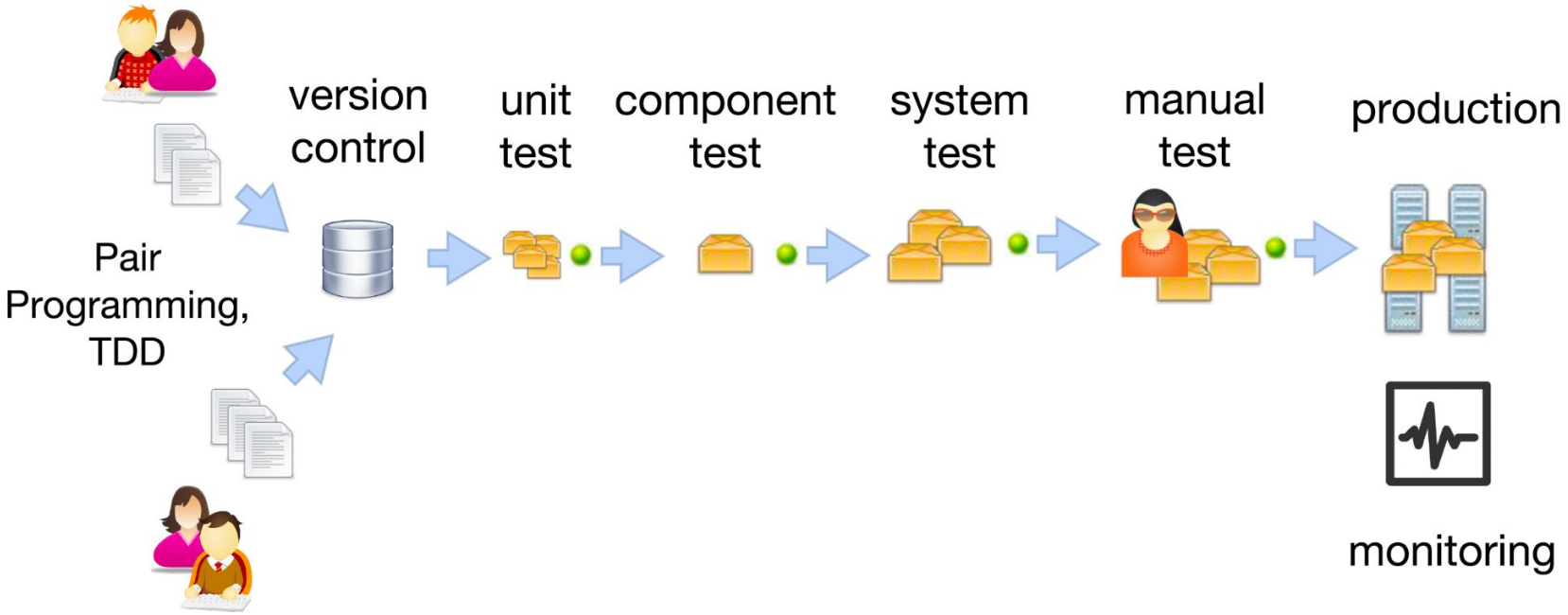


Speed

- Unit tests should be fast
- Run at every build
- It’s not a unit test if it:
    - Touches the filesystem
    - Touches the network
    - Touches the database
    - Requires custom hardware



Testing in the pipeline



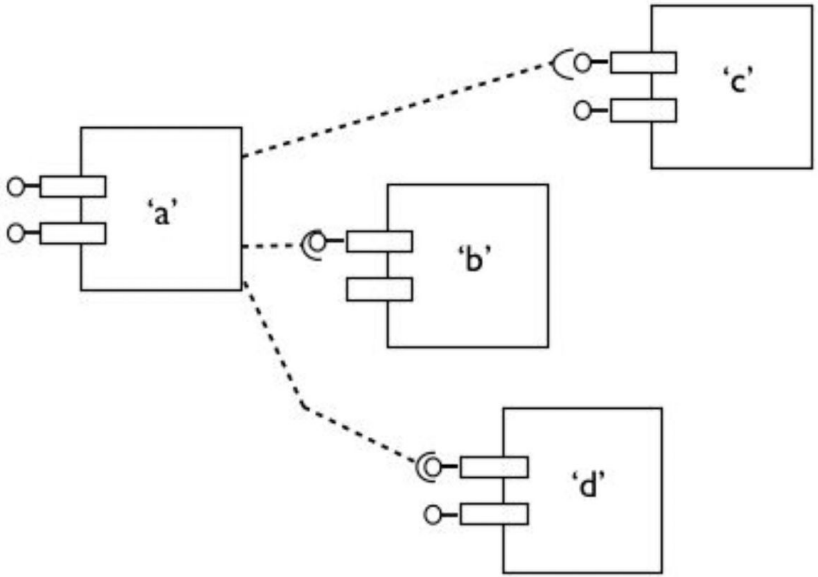
Robustness

- Isolation:
  - Failures don’t cascade
  - Order of test execution irrelevant
  - Execute same test repeatedly
- Failures aren't intermittent
  - No threading/race conditions
  - No network speed issues



Know the limits

- Unit tests cannot find integration errors
- Testing in general can’t find all the errors



Summary: Unit tests

- Give you confidence your code does what you think it does
- When they fail, they tell you what's wrong
- Enable Refactoring
- In order to achieve this, prioritize:
    - Readability
    - Robustness
    - Speed
- Everything else you know about good design applies to tests too!







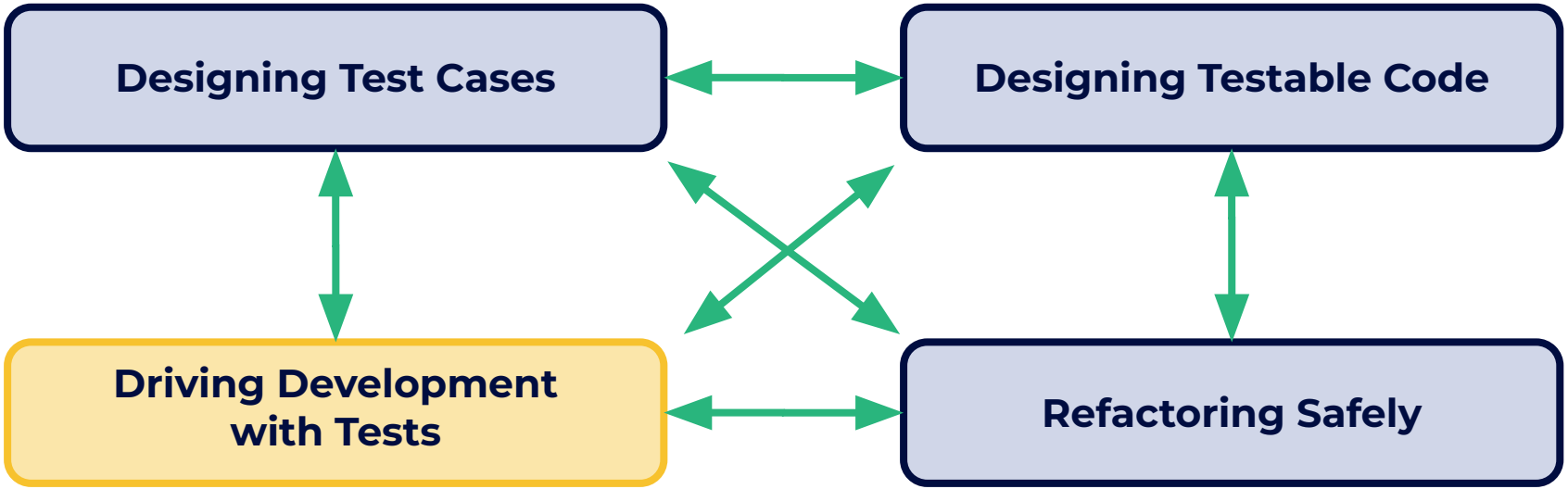
Exercise: Roman Numerals

Designing Test Cases



Mike Long @meekrosoft

TDD Skills


Designing Test Cases
Designing Testable Code
Driving Development with Tests
Refactoring Safely

That's all folks!

Thanks for your time 🙏 If you'd like to know more about Kosli scan the QR code!



Mike Long @meekrosoft