LP IDSE - GL

Software Testing
A bit further

Based on Sebastien Mosser's course

13/03/2017
Cécile Camillieri

Today's plan

- Test coverage
- Building better tests
- Simple case study walkthrough

Before we go

About the answers to the form

- I will not go back on everything
- Our job is not to teach you programming
- No solution is “simple and quick”, and if it is, it’s because you’ve done it a lot before

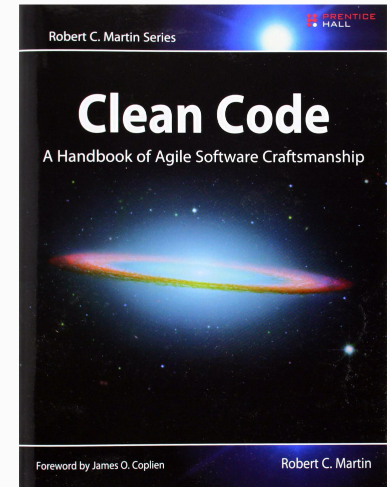
- More generally, we can only give you some concepts and starting points, then you have to provide the effort to improve yourself

How to improve

- Read.
- There are a lot of references out there.

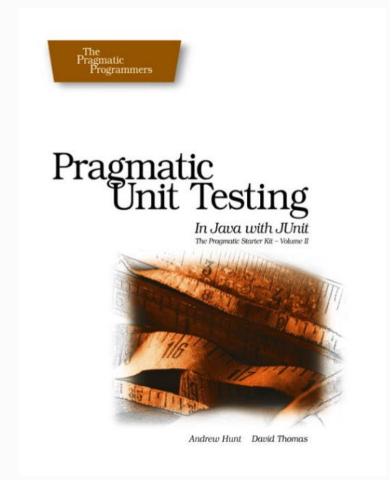
- **Practice.**
- **Knowing (how to do)** is different than **being able to do**.

Some references



Clean Code
A Handbook of Agile Software Craftsmanship
Robert C. Martin

UML@Classroom
(http://www.uml.ac.at/en/lernen)
Martina Seidl, Marion Scholz,
Christian Huemer, Gerti Kappel



Pragmatic Unit Testing
in Java with JUnit
Andrew hunt, David Thomas

How would you describe good code?

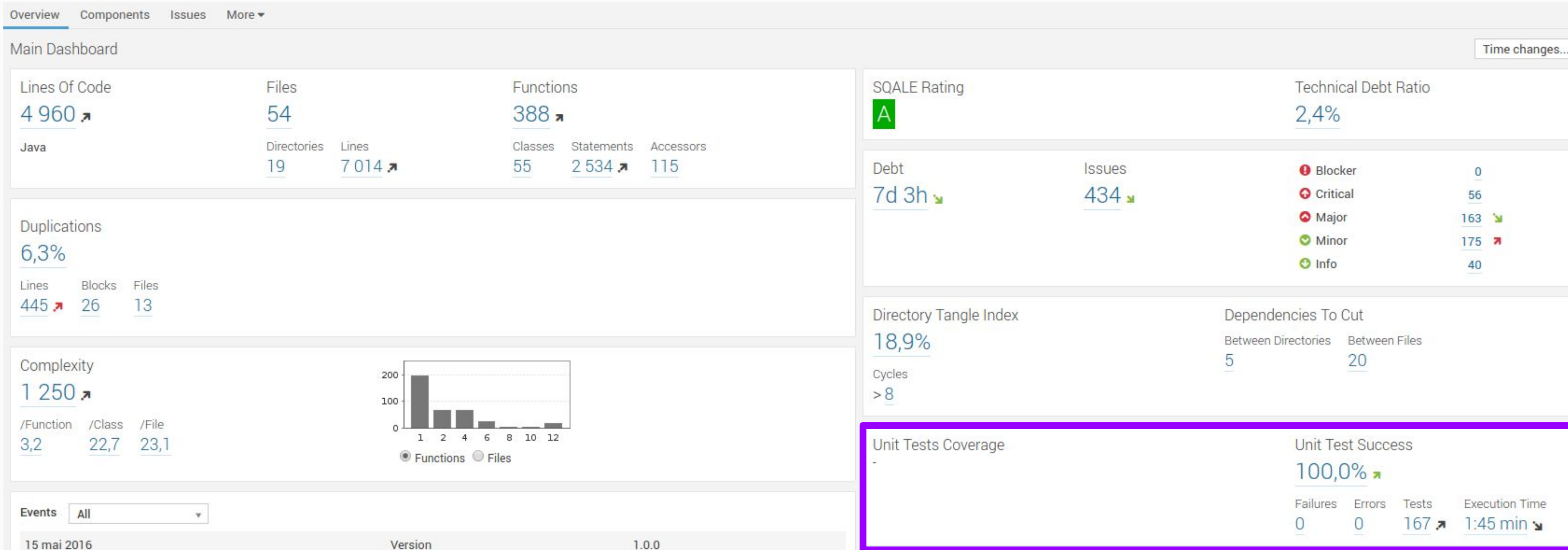
> “The @author field of a Javadoc tells us who we are. We are authors. And one thing about authors is that they have readers. Indeed, authors are responsible for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.”

Robert C. Martin
Clean Code: A Handbook of Agile Software Craftsmanship

Test

Coverage

SonarQube (previously)



SonarQube and Unit Tests

Unit Tests Coverage
-

????

Unit Test Success
100,0%

Failures	Errors	Tests	Execution Time
0	0	167	1:45 min

Unit Tests Coverage
9,6%

Line Coverage	Condition Coverage
8,9%	22,5%

Unit Test Success
100,0%

Failures	Errors	Tests	Execution Time
0	0	6	158 ms

Measuring code coverage

- Requires an ‘agent’ that observes the JVM
- Jacoco is such an agent

Using Jacoco with Maven

- Thankfully there is a maven plugin !
- We need to use it for all modules
- And connect it to SonarQube

In the parent pom.xml - part 1

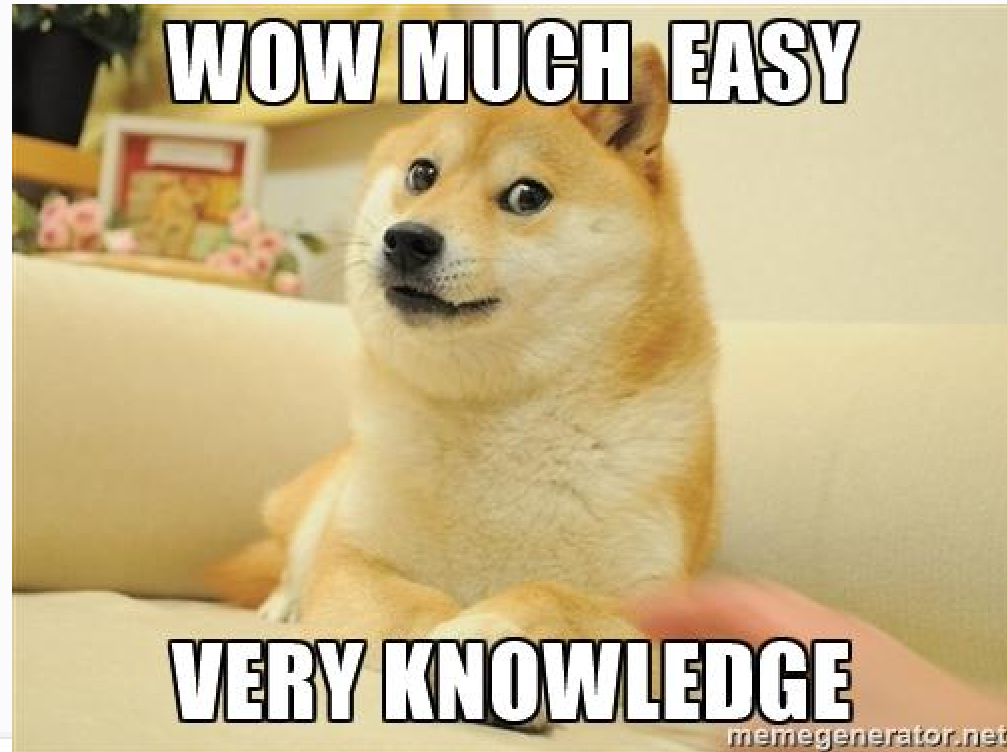
```xml
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.7.6.201602180812</version>
      <executions>
        <execution>
          <id>agent-for-ut</id>
          <goals>
            <goal>prepare-agent</goal>
          </goals>
          <configuration>
            <append>true</append>
            <destFile>${sonar.jacoco.reportPath}</destFile>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Append reports of all modules
in a single file

In the parent pom.xml - part 2

```
<pluginManagement>
  <plugins>
     <...>
     <plugin>
       <groupId>org.jacoco</groupId>
       <artifactId>jacoco-maven-plugin</artifactId>
       <version>0.7.8</version>
     </plugin>
  </plugins>
</build>

<properties>
     <sonar.jacoco.reportPath>
         ${project.basedir}/../target/jacoco.exec
     </sonar.jacoco.reportPath>
     <sonar.language>java</sonar.language>
     <sonar.java.coveragePlugin>jacoco</sonar.java.coveragePlugin>
</properties>
```

That's it!



Unit Tests Coverage

9,6%

Line Coverage	Condition Coverage
8,9%	22,5%

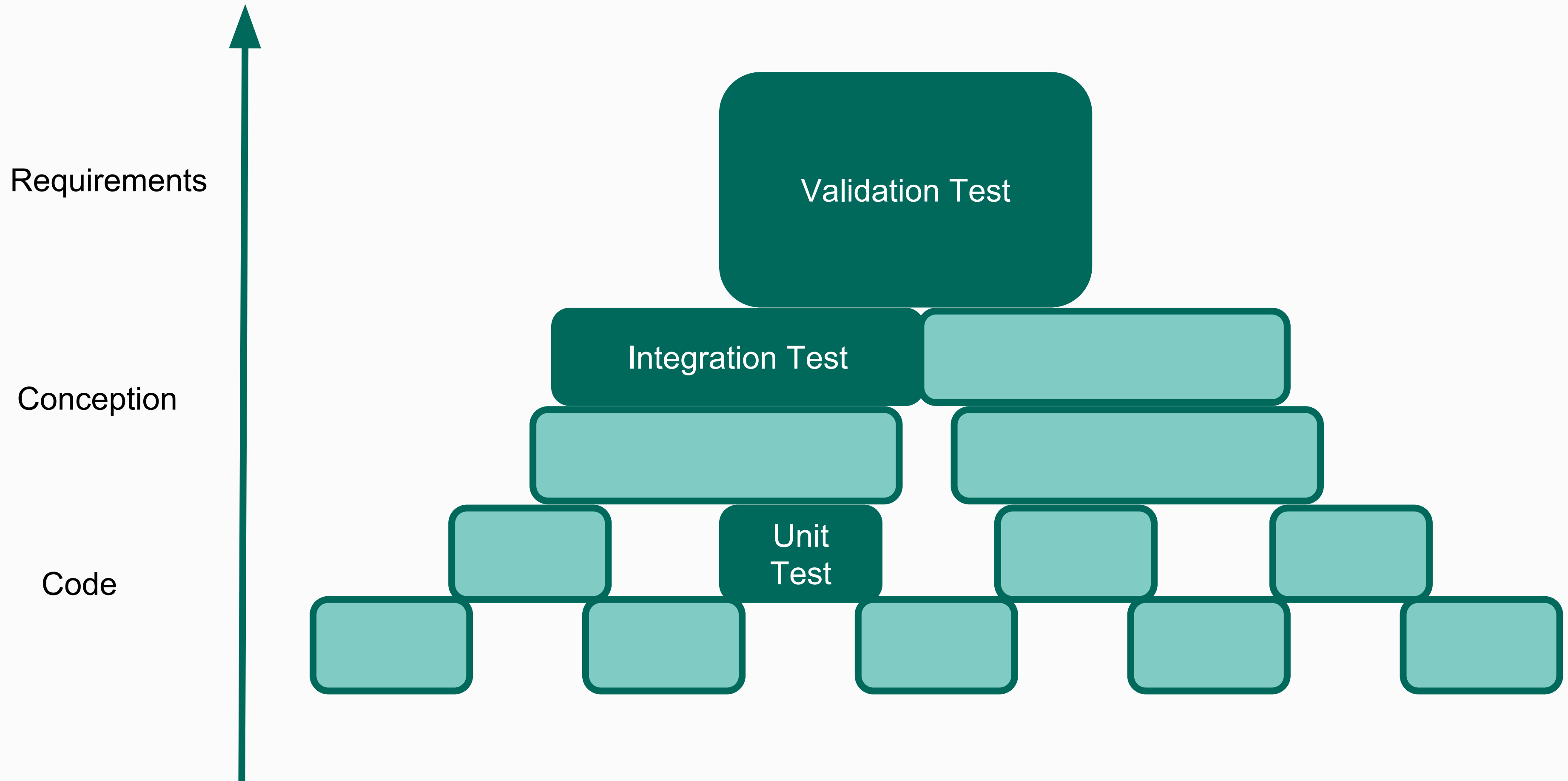
Unit Test Success

100,0%

Failures	Errors	Tests	Execution Time
0	0	6	158 ms

Previously...

Tests strategies



JUnit 5

Software Testing by example

JUnit Tag words

@AfterClass / @BeforeClass

@After / @Before

@Test

assert*

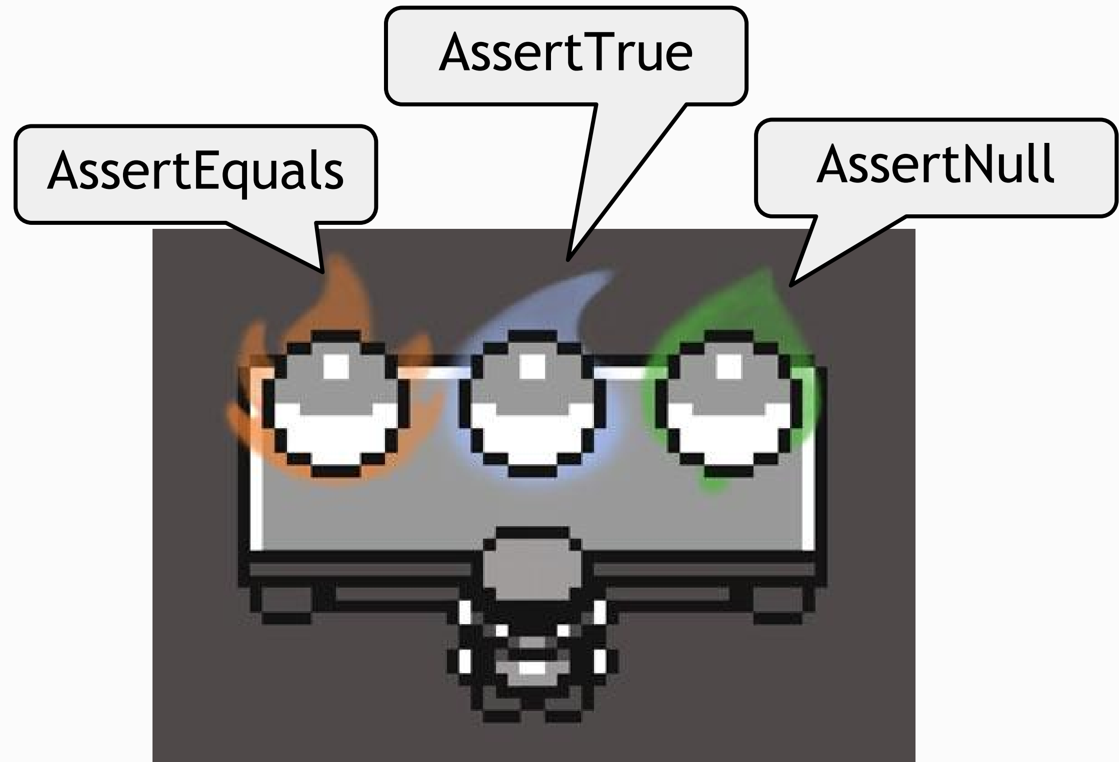
fail()

expected

Chose wisely !

Think about:

- Correctness
- Readability
- Extensibility



Towards more complete tests

The Javadoc should be all we need to write the tests!

```
/**
* Definit une nouvelle {@link Shape} dans le service.
*
* @param shape - L'objet {@link Shape} a ajouter
* @throws ShapeAlreadyExistException si une forme avec le meme identifiant
etait deja presente.
*/
```

- Parameters: Test edge cases
  - What happens if I give a ‘null’ Shape, or a Shape with only 1, 2, or no vertex? Should it be added?

  => Writing tests helps us realize that our documentation, specifications or implementation is not good enough
  - Maybe an InvalidShapeException could be thrown, or a boolean returned as false if the shape was not added bc it was invalid

Test exceptions - Another way

We can create a separate test method for this case!

```
@Test(expected = ShapeAlreadyExistException.class)
public void testCreateAlreadyExistingShape()
{
    int nb = ShapesProvider.getAllShapes().size();
    // We know star is already in here
    Shape s = new Shape("star");
    ShapesProvider.createShape(s);
}
```

Here, the test will fail if the exception is not thrown, and pass otherwise.

- The @Test(expected …) shows directly what the test is for
- It's cleaner and easier to read

Unit Tests Quality

Right BICEP and CORRECT

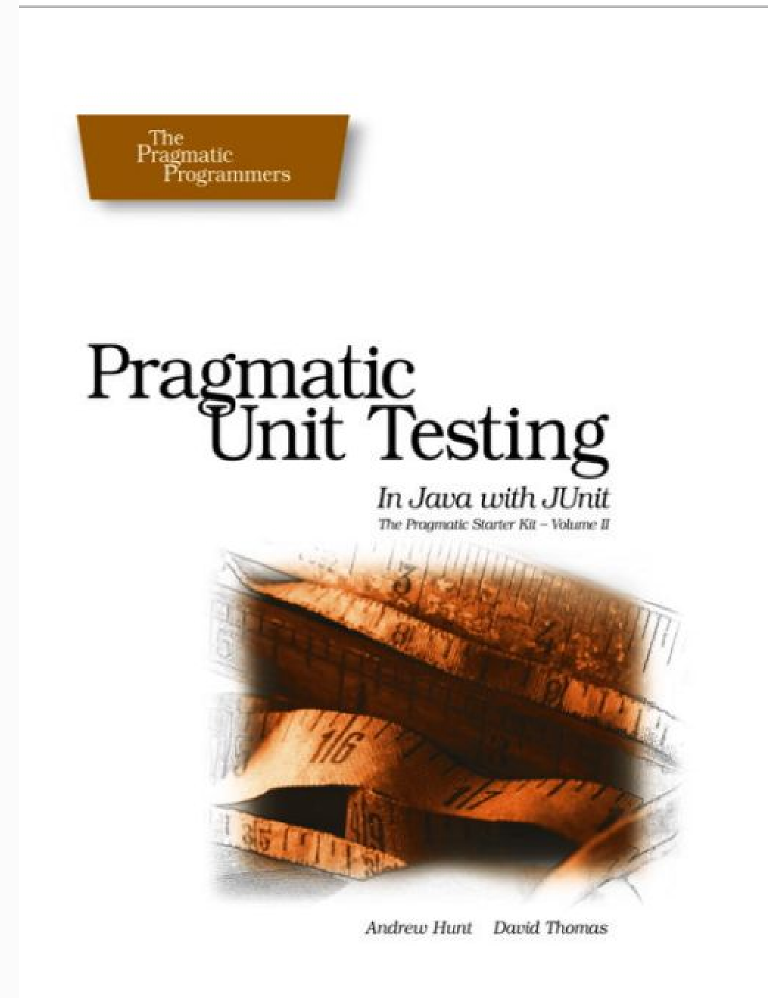
a.k.a Computer Scientists love acronyms

Reference ->

Go read it!

(really!)

Also based on Sebastien Morsser course



Strong tests with (Right) BICEP

- **Right:** are the results right?
- **Boundaries:** are the boundary conditions **CORRECT**?
- **Inverse relationships:** can you check inverse relationships?
- **CrossCheck:** can you cross-check results using other means?
- **Error:** can you force error conditions to happen?
- **Performance:** are performance characteristics within bounds?

“Right” tests

- The basis of all tests:
  - Check that the program behaves as expected
  - Check that the program output it correct

- The square root example:

$$\sqrt{}: \mathscr{R}^+ \rightarrow \mathscr{R}^+$$
$$a \mapsto b, \quad b \times b = a$$

Specification

$$\sqrt{4} = 2$$

“Right” test

Boundary conditions

- Boundaries are where error happens !
- = It’s among the most valuable things to test !

$\sqrt{0}$ = ? $\sqrt{\infty}$ = ?

student.email = "foo@bar"

person.age = -2

room.seats = 42,000

file.path = "#$%&*^@#"

Inverse Relationships

$a \leftarrow \sqrt{343}$

$a^2 = 343$

- We don't care about the value of a
- What's important is to check the property $x^2 = \sqrt{x}$

Cross-checking

- Check against existing libraries or tools if possible

$\sqrt{3285}$ = Math.**sqrt**(3285)

precision = 0.0001

Mine

The one I am replacing

Error

- Thoroughly explore error cases
- **Expect** the **unexpected**

$\sqrt{-1} \Rightarrow$ Illegal Argument

Performance

- Allows to identify performance regression

time(√3285) < 200ms

- With JUnit :

  - Tester une exécution avec une limite de temps
    - Spécifiée en millisecondes

```
@Test(timeout=100)
...
```

  - *Pas d'équivalent en JUnit 3*

P. Collet 27

CORRECT Boundary conditions

- **Conformance:** does the value conform to an expected format?
- **Ordering:** is the set of values ordered or unordered as appropriate?
- **Range:** is the value within reasonable minimum and maximum values?
- **Reference:** does the code reference anything external that isn't under direct control of the code itself?
- **Existence:** does the value exist?
- **Cardinality:** are there exactly enough values?
- **Time:** is everything happening in order? At the right time? In time?

Case Study

The Dice Game Kata

Reference ->

https://github.com/polytechnice-si/3A-GL-DiceGame

By Sébastien Mosser and Simon Uril

Task #1: throwing a dice

Being able to throw a dice

Acceptance criteria: The dice has 6 faces, and returns a random number in [1,6].

```
public class Dice {

    private final static int FACES = 6;
    private Random rand;

    public Dice(Random rand) { this.rand = rand; }

    public int roll() {
        int result = rand.nextInt(FACES) + 1;
        if (result < 1 || result > FACES)
            throw new RuntimeException("Dice returns an incompatible value");
        return result;
    }
}
```

Check 1: We can roll a die

```
public class DiceTest {

  Dice theDice;

  @Test
  public void rollReturnsAValue() {
      theDice = new Dice(new Random());
      for(int i = 0; i < 100; i++) {
          int result = theDice.roll();
          assertTrue(result >= 1);
          assertTrue(result <= 6);
      }
  }
}
```

Check 2: Invalid roll die values

- roll() should throw an exception if the rolled value is not between 1 and 6

- The current implementation and available tools don’t allow us to test this directly

=> What should we do?

Check 2: Naive solution

- Implement a specific Random

```
class NoRandom extends Random {
  int value;
  public NoRandom(int v) { this.value = v; }
  @Override
  public int nextInt(int m) { return value; }
}
```

- Test the some cases

```
@Test(expected = RuntimeException.class)
public void identifyBadValuesGreaterThanNumberOfFaces() {
  theDice = new Dice(new NoRandom(7));
  theDice.roll();
}
@Test(expected = RuntimeException.class)
public void identifyBadValuesLesserThanOne() {
  theDice = new Dice(new NoRandom(-1));
  theDice.roll();
}
```

Check 2: Naive solution

- Implement a specific Random

```
class NoRandom extends Random {
  int value;
  public NoRandom(int v) { this.value = v; }
  @Override
  public int nextInt(int m) { return value; }
}
```

- Overriding classes for tests does not make any sense

  => Another suggestion?

Check 2: Using mocks

- We only need to consifer a Random where we can change the behavior depending on our contect

- **Mock Objects** are exactly made for that !

```
@Test(expected = RuntimeException.class)
public void identifyBadValuesGreaterThanNumberOfFaces() {
  Random tooMuch = mock(Random.class);
  when(tooMuch.nextInt(anyInt())).thenReturn(7);

  theDice = new Dice(tooMuch);
  theDice.roll();
}
```

Task #2: associate a die to a player

Associating a dice roll result to a given player

Acceptance criteria: A player has a name, and exposes the value obtained from her very own dice

```java
public class Player {

    private String name;
    private Dice dice;
    private int lastValue = -1;

    public Player(String name, Dice dice) {
        this.name = name;
        this.dice = dice;
    }

    public void play() {
        this.lastValue = dice.roll();
    }

    public int getLastValue() {
        return lastValue;
    }

}
```

Check: Player can roll a die

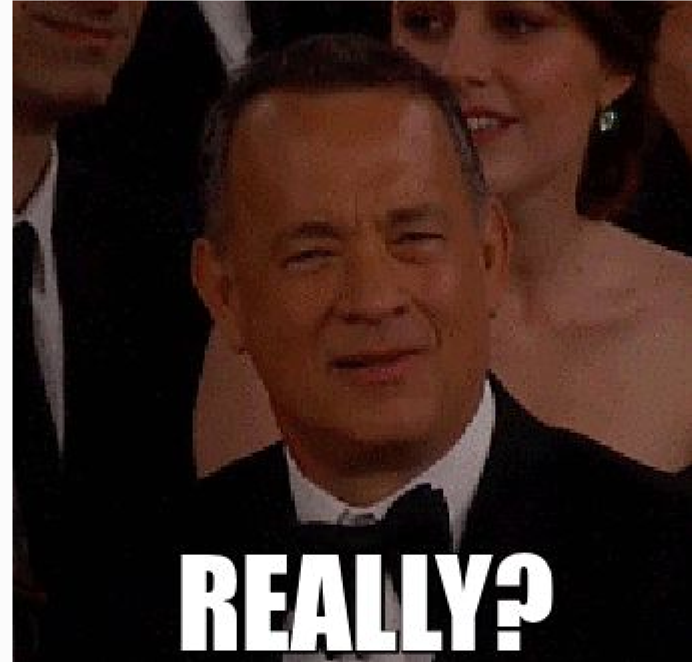
```
public class PlayerTest {

  Player p;

  @Test
  public void lastValueNotInitialized() {
     p = new Player("John Doe", new Dice(new Random()));
     assertEquals(p.getLastValue(), -1);
  }

  @Test
  public void lastValueInitialized() {
     p = new Player("John Doe", new Dice(new Random()));
     p.play();
     assertNotEquals(p.getLastValue(), -1);
  }
}
```

Check: Player can roll a die

- -1 when the die was never rolled?

- Magic number
- An external developer cannot understand what this value means
- It's part of our technical debt

=> Any suggestion?



Check: Java 8 Optionals

- Manipulate objects that may or may not be defined

```
public class Player {

    private String name;
    private Dice dice;
    private Optional<Integer> lastValue = Optional.empty();

    public Player(String name, Dice dice) {
        this.name = name;
        this.dice = dice;
    }

    public void play() {
        this.lastValue = Optional.of(dice.roll());
    }

    public Optional<Integer> getLastValue() {
        return lastValue;
    }
```

Check: Java 8 Optionals

- Only need to check if the value is defined

```
@Test
public void lastValueNotInitialized() {
  p = new Player("John Doe", new Dice(new Random()));
  assertFalse(p.getLastValue().isPresent());
}

@Test
public void lastValueInitialized() {
  p = new Player("John Doe", new Dice(new Random()));
  p.play();
  assertTrue(p.getLastValue().isPresent());
}
```

Task #3: Take max of two rolls

The player throws two dices and keeps the max

Acceptance criteria: the dice is only thrown twice, and only the max value is kept.

- Redefine the play method

```
public void play() {
    int a = dice.roll(); int b = dice.roll();
    this.lastValue = Optional.of(Math.max(a,b));
}
```

Check 1: Player rolls only twice

- Mock Objects allow measure the execution flow that goes through a given mock when a method is called

```
@Test
public void throwDiceOnlyTwice() {
  Dice d = mock(Dice.class);

  p = new Player("John Doe", d);
  p.play();

  verify(d, times(2)).roll();
}
```

Check 2: Player takes max value

- Like before, we control the values returned by our Mock

```
@Test
public void keepTheMaximum() {
  Dice d = mock(Dice.class);
  p = new Player("John Doe", d);

  when(d.roll()).thenReturn(2).thenReturn(5);
  p.play();
  assertEquals(p.getLastValue().get(), new Integer(5));

  when(d.roll()).thenReturn(6).thenReturn(1);
  p.play();
  assertEquals(p.getLastValue().get(), new Integer(6));
}
```

Task #4: Play a Game of Dice

A *Game of Dice* is a two players game, and the player who obtains the max value on a dice roll win (ex-aequo implies to restart the game, no winner after 5 ex-aequo matches)

Acceptance criteria:the game exposes a winner, according to the game rules

```
public class Game {

    private Player left;
    private Player right;

    public Game(Player left, Player right) {
        this.left = left;
        this.right = right;
    }

    public Optional<Player> play() {
        int counter = 0;
        while(counter < 5) {
            left.play();  int l = left.getLastValue().get();
            right.play(); int r = right.getLastValue().get();

            if(l > r )     { return Optional.of(left);  }
            else if (r > l) { return Optional.of(right); }

            counter++;
        }
        return Optional.empty();
    }
}
```

Check 1: No winner case

- Mock dice to always return 1
- “Spy” on player to check number of calls

```
@Test
public void noWinnerAfter5Attempts() {
  Dice single = mock(Dice.class);
  when(single.roll()).thenReturn(1);

  Player p1 =  spy(new Player("John", single));
  Player p2 =  spy(new Player("Jane", single));

  g = new Game(p1,p2);
  assertFalse(g.play().isPresent());
  verify(p1, times(5)).play();
  verify(p2, times(5)).play();
}
```

Check 2: Winner case

- Like before, we control the values returned by our Mock objects

```
@Test
public void andTheWinnerIs() {

   Player p1 = mock(Player.class);
   when(p1.getLastValue()).thenReturn(Optional.of(new Integer(5)));

   Player p2 = mock(Player.class);
   when(p2.getLastValue()).thenReturn(Optional.of(new Integer(2)));

   g = new Game(p1,p2);
   assertEquals(p1, g.play().get());
}
```

What we've seen - Summary

- Mock objects to control return values.
- Mock objects to monitor execution flow.
- Spy objects to monitor execution flow.

- Optionals to manipulated not always defined objects.