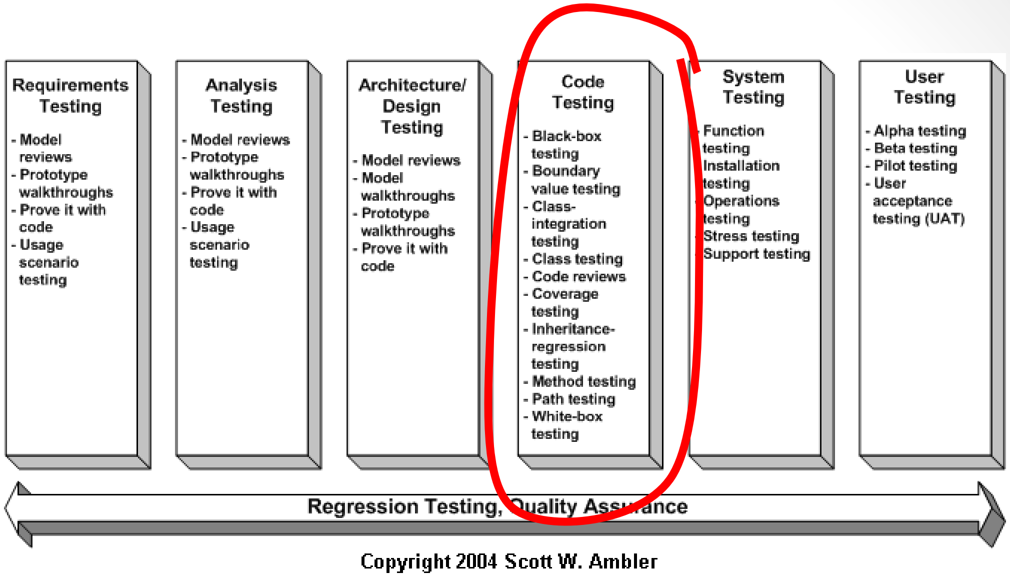
Test-Driven Development (TDD)

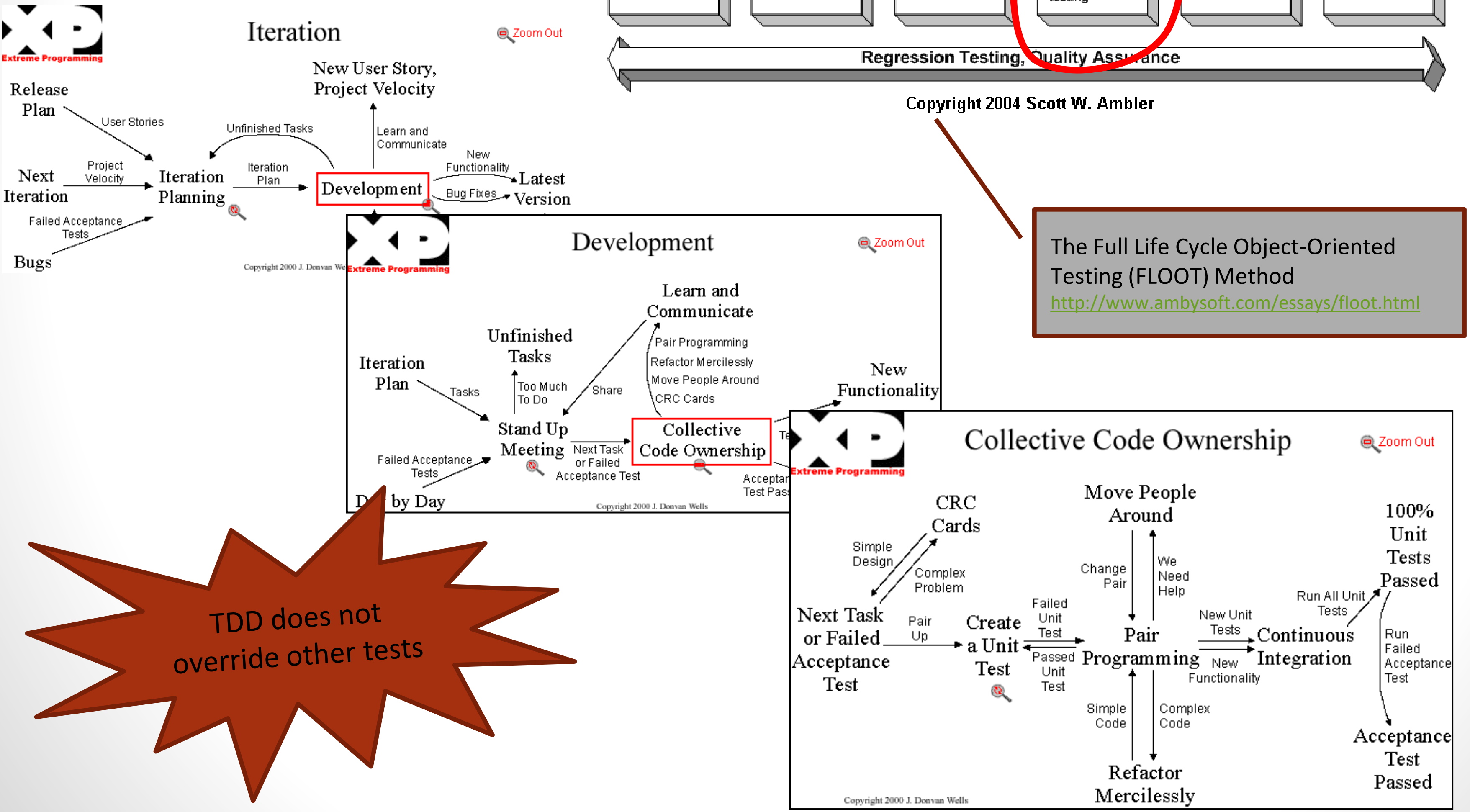
EAPLI

TDD INTRODUCTION

TDD Context



The Full Life Cycle Object-Oriented Testing (FLOOT) Method
http://www.ambysoft.com/essays/floot.html



TDD does not override other tests

Motivations (i)

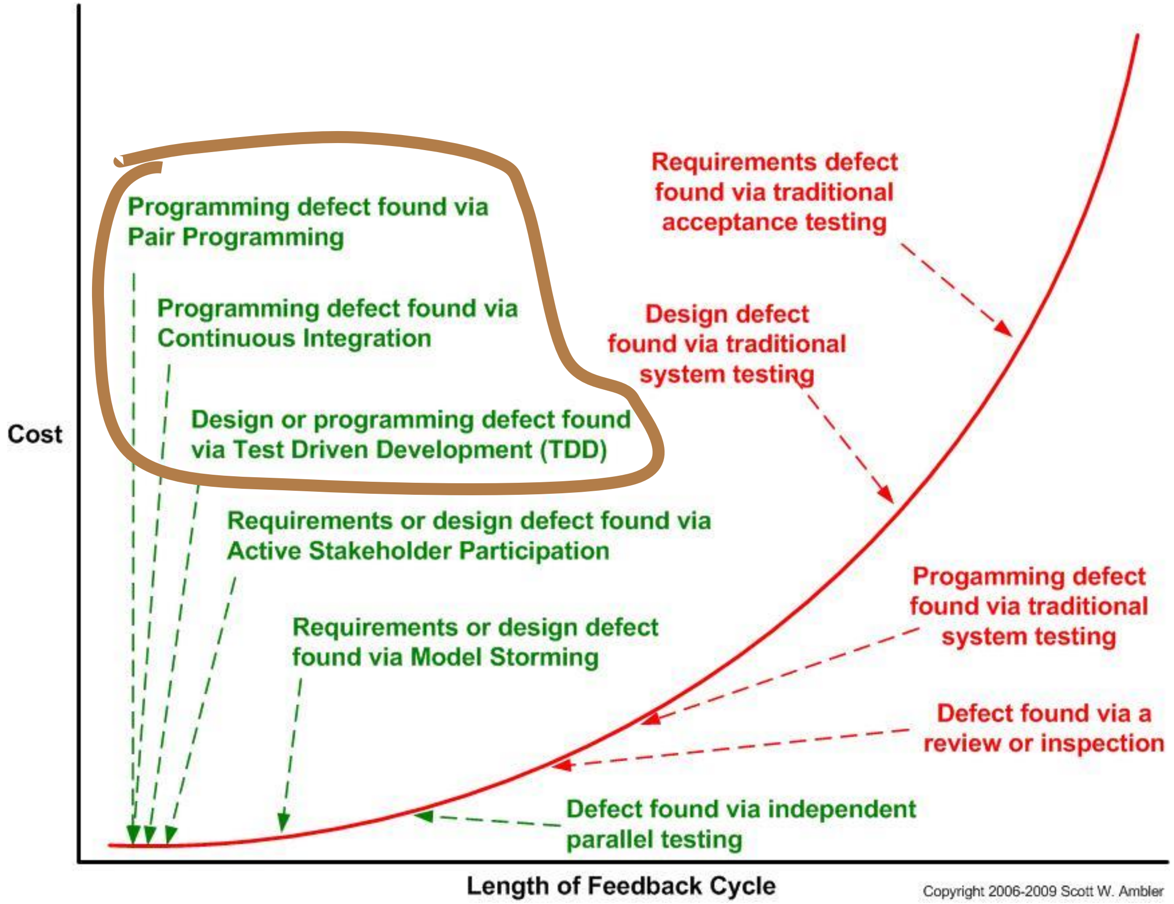
Manual Testing

- Human-Only
- Tiring
- Takes too long
- Difficult to repeat in same conditions
- Hardly Complete

Automatic Testing

- Multiple times execution
  - Bot Execution: by day, hour, etc.
- Whenever necessary
- Always the same way
- Complete Testing
  - For different values
  - Code coverage metrics
    - Jacoco/Cobertura

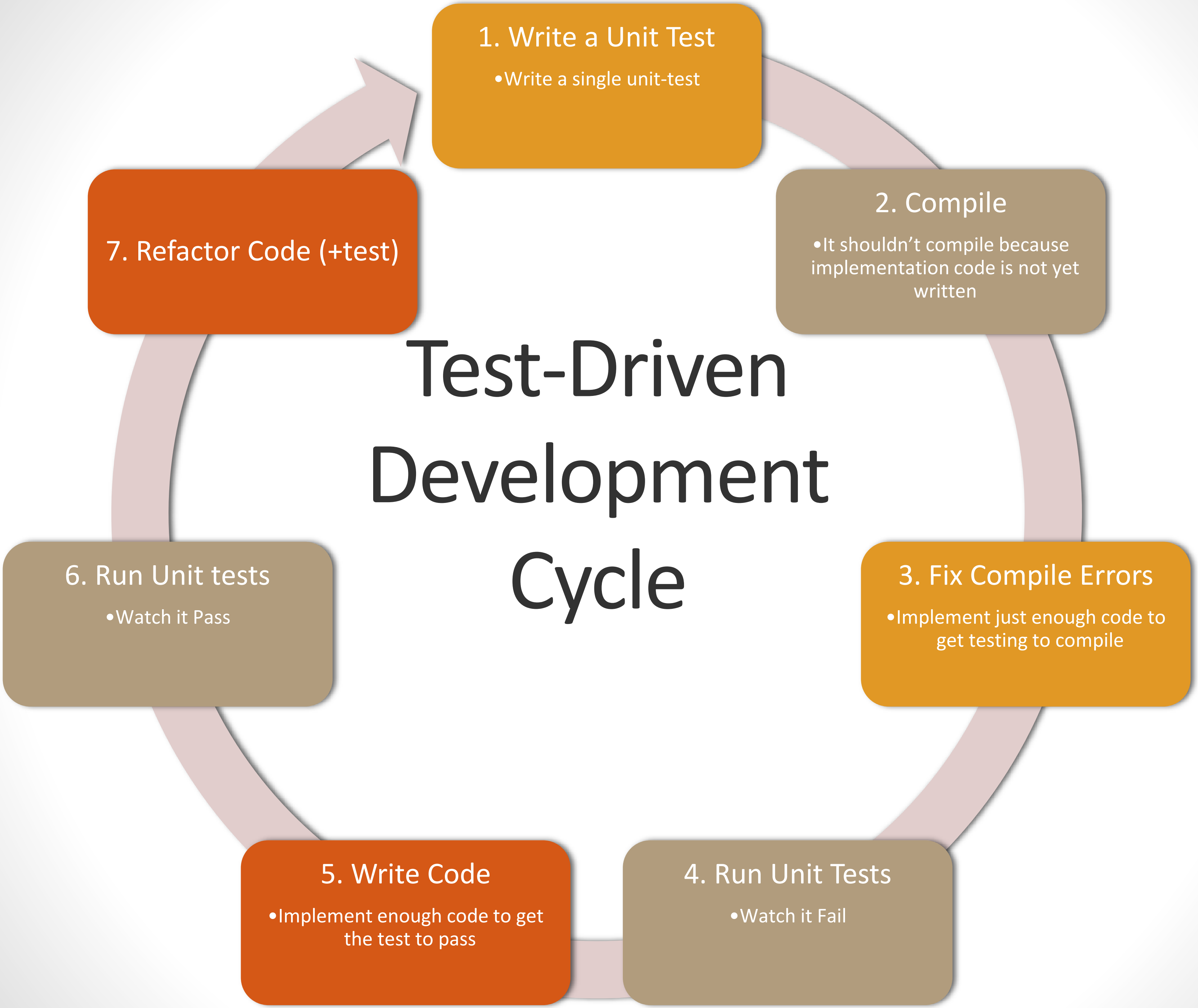
Motivations (ii)



TDD Definition

- TDD is a **software development process** that relies on the **repetition of a very short development cycle**
- Test-cases are a measure/specification of what code should do
- Operation
  - TDD is the technique of writing test-cases ***before*** code
  - Testing ***leads*** to the development of program code

Test-Driven Development Cycle



1. Write a Unit Test
- Write a single unit-test

2. Compile
- It shouldn’t compile because implementation code is not yet written

3. Fix Compile Errors
- Implement just enough code to get testing to compile

4. Run Unit Tests
- Watch it Fail

5. Write Code
- Implement enough code to get the test to pass

6. Run Unit tests
- Watch it Pass

7. Refactor Code (+test)

TDD Rules

- Unit Tests and Code should be written in elementary and incremental steps
- Only write Code when Unit Tests fail
- Each developer should write its own Unit Tests
- Design should follow the High Cohesion Low Coupling (HCLC) principle

UNIT TESTING

Unit Testing

- Unit Testing means testing individual units of behavior
- Should be something that is written instead of performed
- Are pieces of code, comprised by input, conditions and outputs
- Reduces refactoring bugs
- Reduces testing efforts

Unit Testing Tips

- Before writing a unit test, think about the following questions:
  - What are you testing?
  - What should it do?
  - How can the test be reproduced?
  - What is the expected output?
  - What is the actual output?

What to Unit Test?

Usually

- Business Logic Layer rules
- Functions with pre and Post Conditions
  - With Valid inputs
  - With Invalid Inputs
  - By identifying all the exceptions
    - E.g.: fatorial(-1)
  - For extreme value limits
    - E.g. fatorial(0)

Hardly

- User-Interface code
- Database Schemas
- Complex code that requires mocking

TDD References

- Introduction to TDD
  - http://www.agiledata.org/essays/tdd.html
- Agile Testing and Quality Strategies: Discipline Over Rhetoric
  - http://www.ambysoft.com/essays/agileTesting.html
- Agile Model Driven Development
  - http://www.agilemodeling.com/essays/amdd.htm
- Test Driven Development
  - http://users.khbo.be/peuteman/TDD/Test_Driven_Development.pdf
- JUnit in 60 seconds
  - http://android-workshop.blogspot.pt/2010/11/junit-4-in-60-seconds.html
- Why Most Unit Testing is Waste
  - https://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf
- You Still Don't Know How to Do Unit Testing (and Your Secret is Safe with Me)
  - https://stackify.com/unit-testing-basics-best-practices/

Unit Testing & TDD

EAPLI

Unit Testing & TDD

- Unit Testing
  - Allows to understand how to use a method or a class
  - Allows the specification of code
  - Allows the specification of several forms on how to invoke a method

- Unit Testing refers to **what** you are testing, while TDD refers to **when** you are testing.

- Developers usually do not read code documentation
  - They prefer to work with code, hence Unit Testing
  - TDD combined with Unit Testing is considered a Design Activity

Unit Testing: factorial example

- Business rules
  - n! for any n < 0 is undefined
  - 0! = 1
  - 1! = 1
  - 2! = 2
  - …
  - n! = (n-1)! + (n-2)!

Test-Case

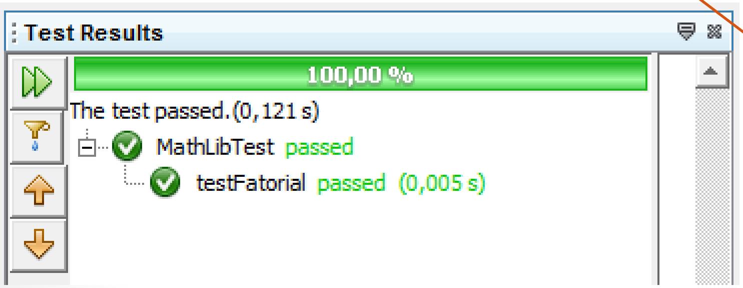
```
public class MathLibTest {
   public MathLibTest()   {

  @Test (expected = IllegalArgumentException.class)
   public void ensureFatorialOfMinusOneFails() {
      MathLib tester = new MathLib();
      tester.fatorial(-1);
   }
   public void ensureFatorialOfZeroIsOne() {
      MathLib tester = new MathLib();
      assertEquals("Result", 1, tester.fatorial(0) );
   }
   public void ensureFatorialOfTwoIsTwo() {
      MathLib tester = new MathLib();
      assertEquals("Result", 2, tester.fatorial(2) );
   }
   public void ensureFatorialOfFourIsTwentyFour () {
      MathLib tester = new MathLib();
      assertEquals("Result", 24, tester.fatorial(4) );
   }
}
```

Unit Testing: factorial example

Code

```
public class MathLib {
   public int fatorial(int v) throws IllegalArgumentException {
      if( v==0 )
         return 1;
      else if( v==1 )
         return 1;
      else if(v > 1)
         return v * fatorial(v-1);
      else
         throw new IllegalArgumentException();
   }
}
```



The test passed.(0,121 s)

MathLibTest passed

testFatorial passed (0,005 s)

Test-Case

```
public class MathLibTest {
  public MathLibTest()  {

  @Test (expected = IllegalArgumentException.class)
  public void ensureFatorialOfMinusOneFails() {
     MathLib tester = new MathLib();
     tester.fatorial(-1);
  }
  public void ensureFatorialOfZeroIsOne() {
     MathLib tester = new MathLib();
     assertEquals("Result", 1, tester.fatorial(0) );
  }
  public void ensureFatorialOfTwoIsTwo() {
     MathLib tester = new MathLib();
     assertEquals("Result", 2, tester.fatorial(2) );
  }
  public void ensureFatorialOfFourIsTwentyFour () {
     MathLib tester = new MathLib();
     assertEquals("Result", 24, tester.fatorial(4) );
  }
}
```

JUnit: Unit Testing Example

```
import org.junit.BeforeClass;
// …
import org.junit.Test;
import static org.junit.Assert.*;

public class MathLibTest {
   private MathLib tester;

   public MathLibTest() {
      tester = new MathLib();
   }
   @BeforeClass
   public static void setUpClass() throws Exception {
      System.out.println("before class");
   }
   @AfterClass
   public static void tearDownClass() throws Exception {
      System.out.println("tearDown");
   }
   @Before
   public void before() {
      System.out.println("before");
   }
   @After
   public void after() {
      System.out.println("after");
   }
  @Test(expected = IllegalArgumentException.class, timeout=1)
   public void testFatorial() {
      System.out.println("test start");
      tester.fatorial(-1);
      System.out.println("test finish");
   }
   @Test (timeout=1)
   public void outroTestFatorial() {
      System.out.println("outro test start");
      System.out.println("outro test finish");
   }
}
```

Executed only once when class is loaded, i.e. allows a text fixture to be defined

Executed only once when test are finished

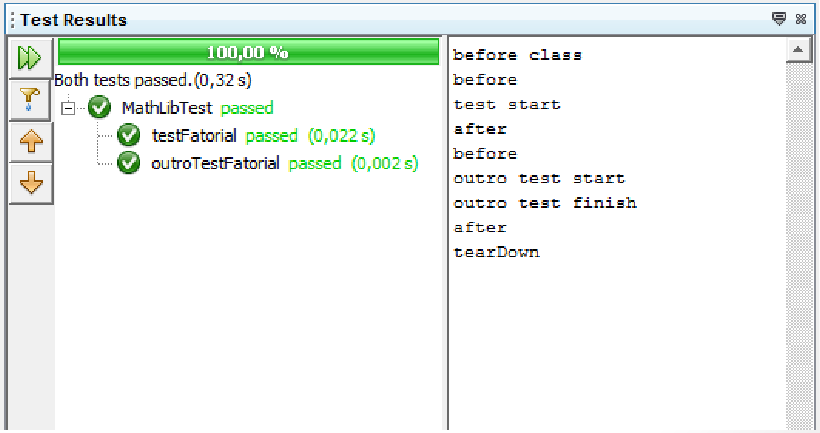
Executed before each method testing

Executed after each method testing

Method Testing

Another Method Testing

☞ http://android-workshop.blogspot.pt/2010/11/junit-4-in-60-seconds.html



Test Results

100,00 %

Both tests passed.(0,32 s)

MathLibTest passed

testFatorial passed (0,022 s)

outroTestFatorial passed (0,002 s)

```
before class
before
test start
after
before
outro test start
outro test finish
after
tearDown
```

Testing Syntax (i)

- Arrange-Act-Assert
  - https://www.typemock.com/unit-test-patterns-for-net#aaa
  - More used with Unit Testing
- Given-When-Then
  - https://martinfowler.com/bliki/GivenWhenThen.html
  - More used with functional or End-to-End (E2E) testing

Testing Syntax (ii)

- Given / Arrange
  - The **Given** part describes the state of the world before you begin the behaviour you're specifying in this scenario. You can think of it as the pre-conditions to the test.
    - e.g. Given two numbers
  - **Arrange:** setup everything needed for running the tested code. This includes any initialization of dependencies, mocks and data needed for the test to run.
    - e.g. Arrange two numbers

Testing Syntax (iii)

- When / Act
  - The **When** section is the behaviour you're specifying.
    - e.g. when I sum them
  - **Act:** Invoke the code under test.
    - e.g. sum them

Testing Syntax (iv)

- Then / Assert
  - The **Then** section describes the changes you expect due to the specified behaviour.
    - e.g. then verify if the result equals **?**

  - **Assert:** Specify the pass criteria for the test, which fails it if not met.
    - e.g. assert the result equals **?**

Testing Syntax: Example

```
@Test
public void ensureThereAreNoNewMessages()
{
   // Arrange / Given
   Mailbox mailbox = new Mailbox();
   int expectedResult = 0;

   // Act / When
   int result = mailbox.newMessagesCount;

   // Assert / Then
   Assert.AreEqual(expectedResult, result);
}
```

QUALITY MEASURES

Code Coverage

- Code Coverage **measures** the **degree** to which the source code of a program is executed when a particular test suite runs.
- Tools
  - Clover, Cobertura, Jacoco
- Result
  - The percentage of covered code

Code Coverage Example

Unit Test

```
@Test
public void
ensureSecondNegativeOperandWorks() {

 // Arrange
  int firstOperand = 10;
  int secondOperand = -5;
  int expected = 5;

  // Act
  CalculatorExample calculator = new
CalculatorExample();
  int result = calculator.sum(firstOperand,
secondOperand);

  // Assert
  assertEquals(expected, result);
}
```

Code

```
public int sum(int firstOperand, int
secondOperand)
{
   return firstOperand + secondOperand;
}
```

100%

Code Coverage Example

Unit Test

```
@Test
public void
ensureSecondNegativeOperandWorks() {

 // Arrange
  int firstOperand = 10;
  int secondOperand = -5;
  int expected = 5;

  // Act
  CalculatorExample calculator = new
CalculatorExample();
  int result = calculator.sum(firstOperand,
secondOperand);

  // Assert
  //assertEquals(result, result);
}
```

Code

```
public int sum(int firstOperand, int
secondOperand)
{
  return firstOperand + secondOperand;
}
```

Code Coverage “Problem”

Unit Test

```
@Test
public void
ensureSecondNegativeOperandWorks() {

 // Arrange
  int firstOperand = 10;
  int secondOperand = -5;
  int expected = 5;

  // Act
  CalculatorExample calculator = new
CalculatorExample();
  int result = calculator.sum(firstOperand,
secondOperand);

  // Assert
  //assertEquals(result, result);
}
```

Code

```
public int sum(int firstOperand, int
secondOperand)
{
  return firstOperand + secondOperand;
}
```

100%

TDD References

- Introduction to TDD
  - http://www.agiledata.org/essays/tdd.html
- Agile Testing and Quality Strategies: Discipline Over Rhetoric
  - http://www.ambysoft.com/essays/agileTesting.html
- Agile Model Driven Development
  - http://www.agilemodeling.com/essays/amdd.htm
- Test Driven Development
  - http://users.khbo.be/peuteman/TDD/Test_Driven_Development.pdf
- JUnit in 60 seconds
  - http://android-workshop.blogspot.pt/2010/11/junit-4-in-60-seconds.html
- Why Most Unit Testing is Waste
  - https://rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf
- You Still Don’t Know How to Do Unit Testing (and Your Secret is Safe with Me)
  - https://stackify.com/unit-testing-basics-best-practices/

Code Coverage Tools Reference

- Comparison
  - https://confluence.atlassian.com/clover/comparison-of-code-coverage-tools-681706101.html
- Clover
  - https://www.atlassian.com/software/clover
- Cobertura
  - http://cobertura.github.io/cobertura
- Jacoco
  - http://www.eclemma.org/jacoco

TDD MYTHS

Time Consuming

- TDD is **TIME CONSUMING**, business teams would never approve
  - **Business teams don't really care** at all about the **development process** you use, as long as it's **effective**

Code First is quicker

- **Implementing code before** designing a **test-case is quicker**, and tests can be added afterwards
  - This leads developers into programming with no direction in mind
  - I call it “Mindless-Driven Development”

Write all Unit Test beforehand

- **A developer has to write ALL test cases** before starting code development
  - Design and code development should be made in an incremental and iterative approach, using **short & quick development cycles**

Continuous refactoring

- Because the TDD development cycle has a refactoring step, **continuous refactoring is a requirement**
  - A developer should **only refactor when required**

Unit Tests for all code

- **All code requires Unit Tests**
  - **Unit tests work best for functions**
    - Same input => Same output
  - **Avoid Unit Testing when code has many dependencies**
    - **Having to mock** a lot of dependencies is a good indicator that **it is not a unit test**
    - **Unit Test should be short and quick**

TDD: ADVANTAGES & DISADVANTAGES

TDD Advantages

- Keep Customers Satisfied
- Reduce maintenance costs
  - Code is easier to follow and understand
- Improve developer productivity
  - Long term

TDD Advantages (ii)

- Reduce internal defect density from 40% to 80%
  - Side-effects: brand and quality reputation
  - Reduce code complexity
- Increases development speed on the long term
  - Less time spent looking for problems
- Design Aid
  - Test cases provide a clearer perspective on the ideal Software Design
  - Encourage modular design

TDD Advantages (iii)

- Requirement Analysis leads to Unit-Testing
- Unit-Testing leads to Code writing
- No Code should be written if tests do not fail
- Project Documentation
  - Test cases provide descriptions of every implemented code feature
  - Test cases provide examples on how to use objects

TDD Advantages (iv)

- Code Refactoring is a continual operation supported by a batch of unit-tests

- Tests are performed frequently, for each code refactoring

- Quality Assurance (QA)
  - Avoids manual QA
  - Increases Code confidence and Software Quality

- When used with Continuous Integration/Delivery
  - Prevents broken builds from being deployed to production

TDD Disadvantages

- Adds 10% to 30% to initial development costs
- Requires developer discipline