INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Software engineering - Tests

Frédérique Laforest

Mail: Massimo Arnoldi to Kent Beck

- « Unfortunately, for me and for others, tests go at the opposite to the human nature.
- When you free the pig inside you, you see you can program without testing.
- It's only after a while that our rational side takes the lead, and we start writing tests.
- […] programming in tandem reduces the probability the two partners free their pig at the same time. »

Inspection vs test

- Inspection
  - **Static analysis** to identify problems
    - Tools make static analysis of the code, e.g. identify memory leaks
    - Complexity check
    - …
- Software testing
  - **Dynamic analysis** and observation of the system behavior
  - Test suite : set of tests on a unique piece of software

Objectives of a test

- Prove to developers and clients that the software satisfies the requirements
- **Discover** situations where the software behavior is **NOT correct**, not desired or does not conform to the requirements

During software tests

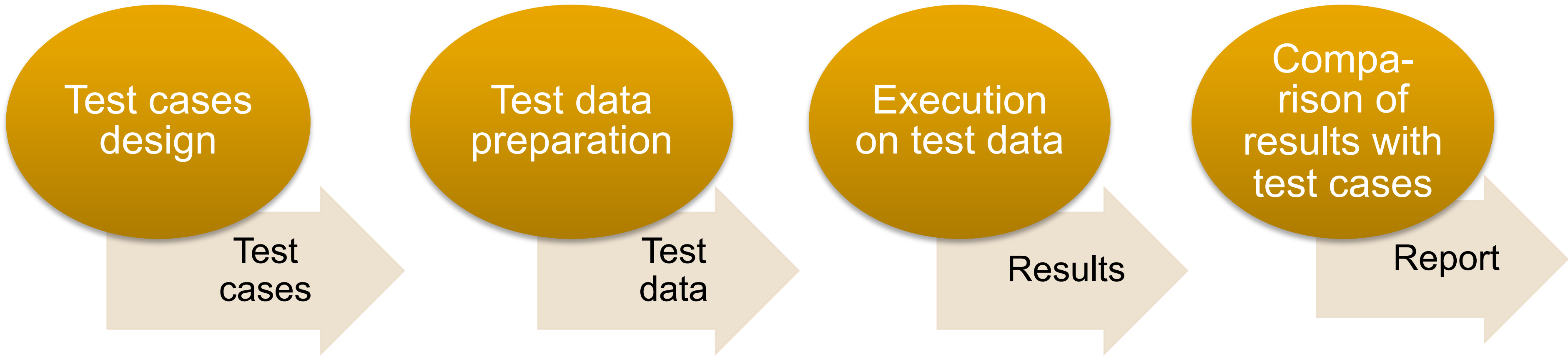
- **Execution** on artificial data
  - Scenarios and testing data must be chosen carefully
- **Control of the results** of each test
  - Identify errors, abnomal results
  - Get information on the software behavior (speed, error rate…)

=> Reveal the presence of errors, NOT their absence

Validity testing vs fault testing

- Validity tests
  - Scenarios showing the normal use of the software
    - We hope the system will execute correctly
    - Successful when the system runs as expected
- Fault tests
  - Scenarios are willingly abnormal
    - Put in evidence faults: incorrect behavior, or different from specifications, find undesired side effects
    - Successful when the system shows a fault
      - Example: system crash, unexpected interactions with other systems, incorrect processings, corrupted data, monkey tests

Testing Process



- This process in interleaved with the software life cycle
  - Test cases and their data are defined during the « requirement engineering » phase
  - Execution and comparison of results during the « test and validation » phase

- User tests
- Version tests
- Development tests

User tests

- Alpha tests
  - Users work with the development team and test the software on site
  - Project team <-> client interaction
- Beta tests
  - A software version is provided to end users who experiment in their own context and identify problems
- Acceptance tests
  - Mainly for software dedicated to an identified client
  - Part of the recette: is the software usable in production?

Version tests

- Tests made on a given version
- The test team is **not** the development team
- « Black box » testing
  - Do not look how it is made inside
  - Does the system do what’s expected? Is it robust enough?
    - Conformity to requirements
    - Develop one or more tests per requirement, launch them, compare

Version tests - Example based on requirements

- Requirement
  - If the patient has allergy to a particular molecule, any prescription of a drug containing it must show an alert to the end user.
  - If a doctor decides to ignore the alert, he must give a reason to the system
- Associated tests
  - Define a patient record with no allergy, prescribe a drug
    - Check no alert is launched
  - Define patient record with an allergy. Precribe a drug containing it
    - Check the right alert is launched

…/…

Version tests - Example based on requirements

- Associated tests (continued)
  - Write a patient record with many allergies. Prescribe drugs that concern more than one of the allergens
    - Check all alerts are launched
  - Write a patient record with allergy, prescribe a drug with it, ignore the alert
    - Check the system asks for the reason and stores it.

  - And so on

Version tests - Example based on a scenario

- Scenario
  - Kate is a specialized nurse, she vistes patients at home to check they follow their prescriptions and they have no side effect. Aech morning, Kate logs in to print her day route and to download the summaries of patients' data. The system asks for a password to encrypt data on the laptop.
- Tests
  - Authenticate with login/pass
  - Download medical records of the patients
  - Print visits agenda
  - Encrypt and decrypt medical records

Version tests - Test of non functional properties

- Must reflect the system usage profile

- Often included in versions tests
  - E.g.: performance, reliability
- Performance tests
  - generally, planification of a series of tests where the system load increases until it reaches a non acceptable level
  - “stress testing”: the system is willingly overloaded to test its behavior in a critical situation

Development tests

- Testing activities made by the dev. team
  - Unit tests
    - Small pieces of code are tested **in isolation**
      - Isolation = no interaction with other units
      - Use mock/stub code
        - Mock : check if some interactions are made
        - Stub : simulate interaction replies
        - A well-known tool: https://github.com/mockito/mockito
  - Integration tests
    - Assemble components who interact with each other
    - Check **interactions**

Unit tests

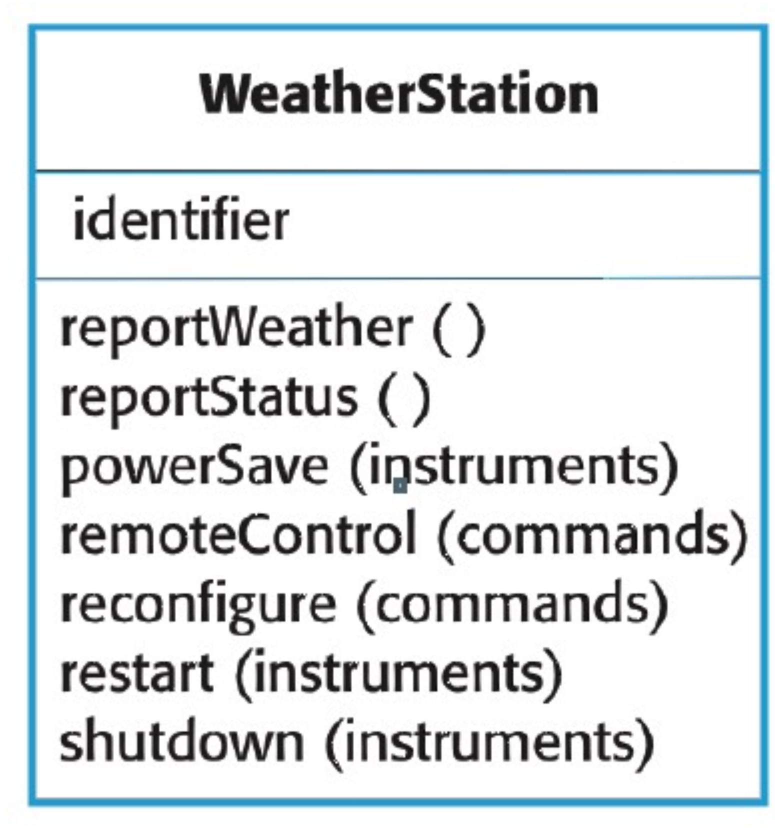
- Individual testing of the smallest pieces of code
  - Functions or methods
  - Classes
- Test a class
  - Check all methods of the class
  - Use objects in all possible states
  - Inheritance adds difficulties as it is not possible to test child classes without their parent class

Unit Tests - Specify and test

- Specify a function / a method
  - Do not say how it works, but
    - WHAT it should do (description)
    - In which context (pre-conditions)
    - With which result (post-conditions)
  - Even before thinking of how to do:
    - Provide the set of use cases
    - Write the list of tests and their code

« tests give me the opportunity to think of what I need without considering how I will implement it »

Unit Tests - Example WeatherStation



Automatise development tests

- As often as possible
  - i.e. no human intervention during the tests (no data input, no choice selection, no manual checking…)
- With a Unit testing environment (e.g. Junit)
  - One can execute at once all implemented tests and get a report (often in a graphical user interface)
- With a Continuous Integration environment (e.g. Jenkins)
  - Automatise components integration and validation at team level
  - Push on repo launches tests and provides reports immediately

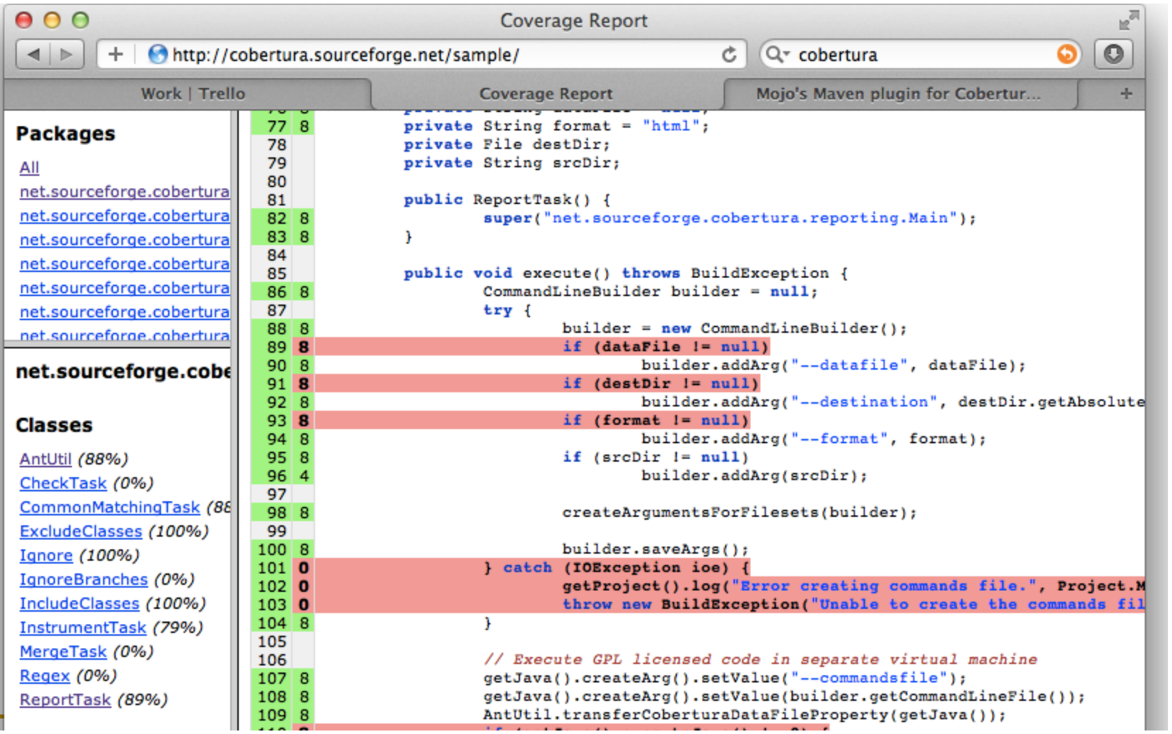
No-regression tests

- Control that changes did not damage code that was previously valid
  - All tests are run at each change
  - Tests must all pass before new code is valid

- In a manual process, no-regression testing is costful.
- In an automatised process, no-regression testing is simple

Tests coverage

- Coverage = percentage of lines tested
  - Often used as a quality indicator
- e.g. https://cobertura.github.io/cobertura/



Coverage Report

http://cobertura.sourceforge.net/sample/

cobertura

Work | Trello — Coverage Report — Mojo's Maven plugin for Cobertur...

Packages

All
net.sourceforge.cobertura
net.sourceforge.cobertura
net.sourceforge.cobertura
net.sourceforge.cobertura
net.sourceforge.cobertura
net.sourceforge.cobertura
net.sourceforge.cobertura

net.sourceforge.cobe

Classes

AntUtil *(88%)*
CheckTask *(0%)*
CommonMatchingTask *(88*
ExcludeClasses *(100%)*
Ignore *(100%)*
IgnoreBranches *(0%)*
IncludeClasses *(100%)*
InstrumentTask *(79%)*
MergeTask *(0%)*
Regex *(0%)*
ReportTask *(89%)*

```
77   8         private String format = "html";
78             private File destDir;
79             private String srcDir;
80
81             public ReportTask() {
82   8                 super("net.sourceforge.cobertura.reporting.Main");
83   8         }
84
85             public void execute() throws BuildException {
86   8                 CommandLineBuilder builder = null;
87                     try {
88   8                         builder = new CommandLineBuilder();
89   8                         if (dataFile != null)
90   8                                 builder.addArg("--datafile", dataFile);
91   8                         if (destDir != null)
92   8                                 builder.addArg("--destination", destDir.getAbsolute
93   8                         if (format != null)
94   8                                 builder.addArg("--format", format);
95   8                         if (srcDir != null)
96   4                                 builder.addArg(srcDir);
97
98   8                         createArgumentsForFilesets(builder);
99
100  8                         builder.saveArgs();
101  0                 } catch (IOException ioe) {
102  0                         getProject().log("Error creating commands file.", Project.M
103  0                         throw new BuildException("Unable to create the commands fil
104  8                 }
105
106                    // Execute GPL licensed code in separate virtual machine
107  8                 getJava().createArg().setValue("--commandsfile");
108  8                 getJava().createArg().setValue(builder.getCommandLineFile());
109  8                 AntUtil.transferCoberturaDataFileProperty(getJava());
```

Test coverage

- 100% coverage is worthless
  - Many methods have no interest for tests, e.g. getters, setters
  - Branch coverage : it is often too heavy to cover all cases
- A good practice
  - Start with tests that cover specification scenarios
  - When bugs are reported: write a test that reproduces the bug, then correct the code (in this order!)

Tests

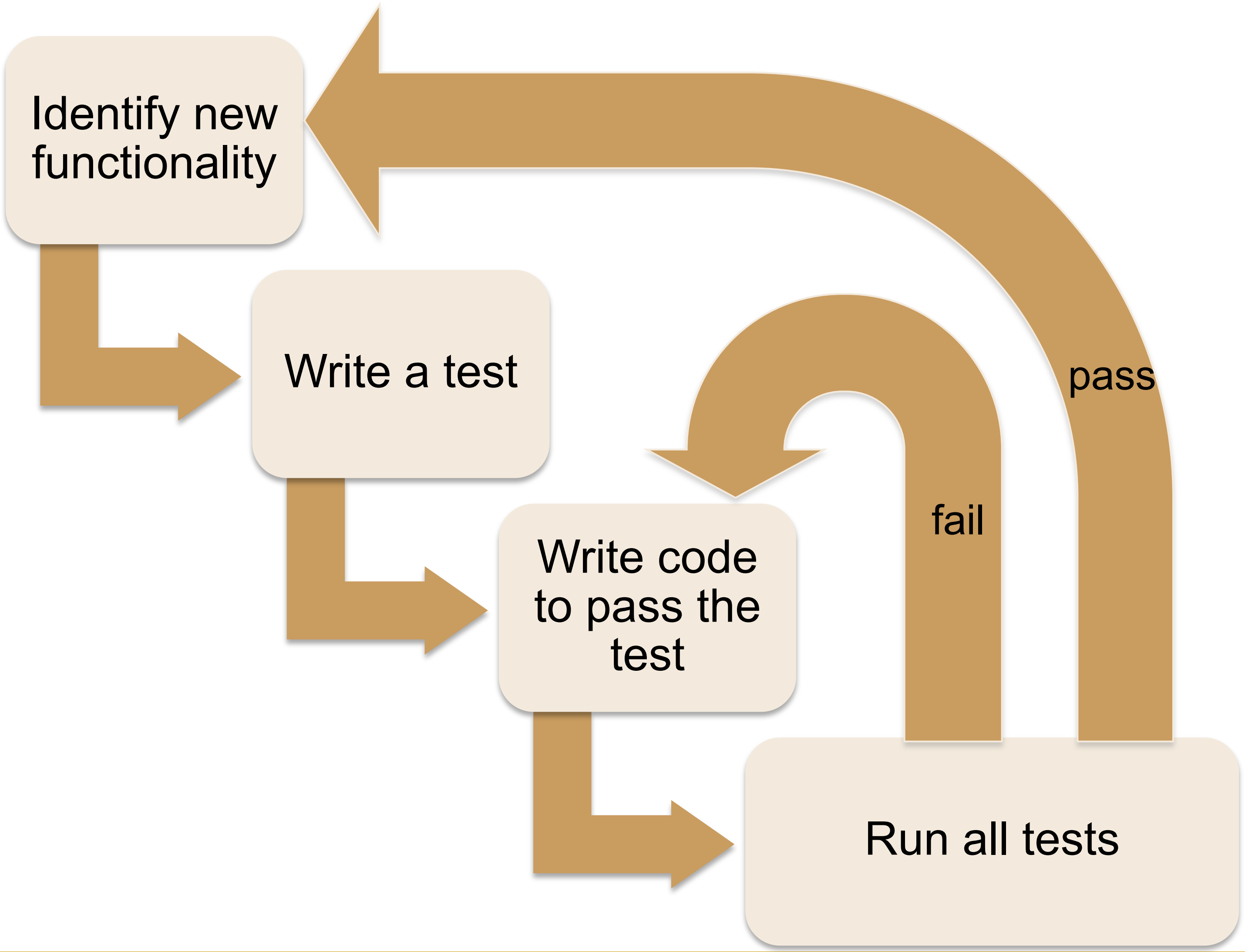
- **Test-driven development**

Test-driven development

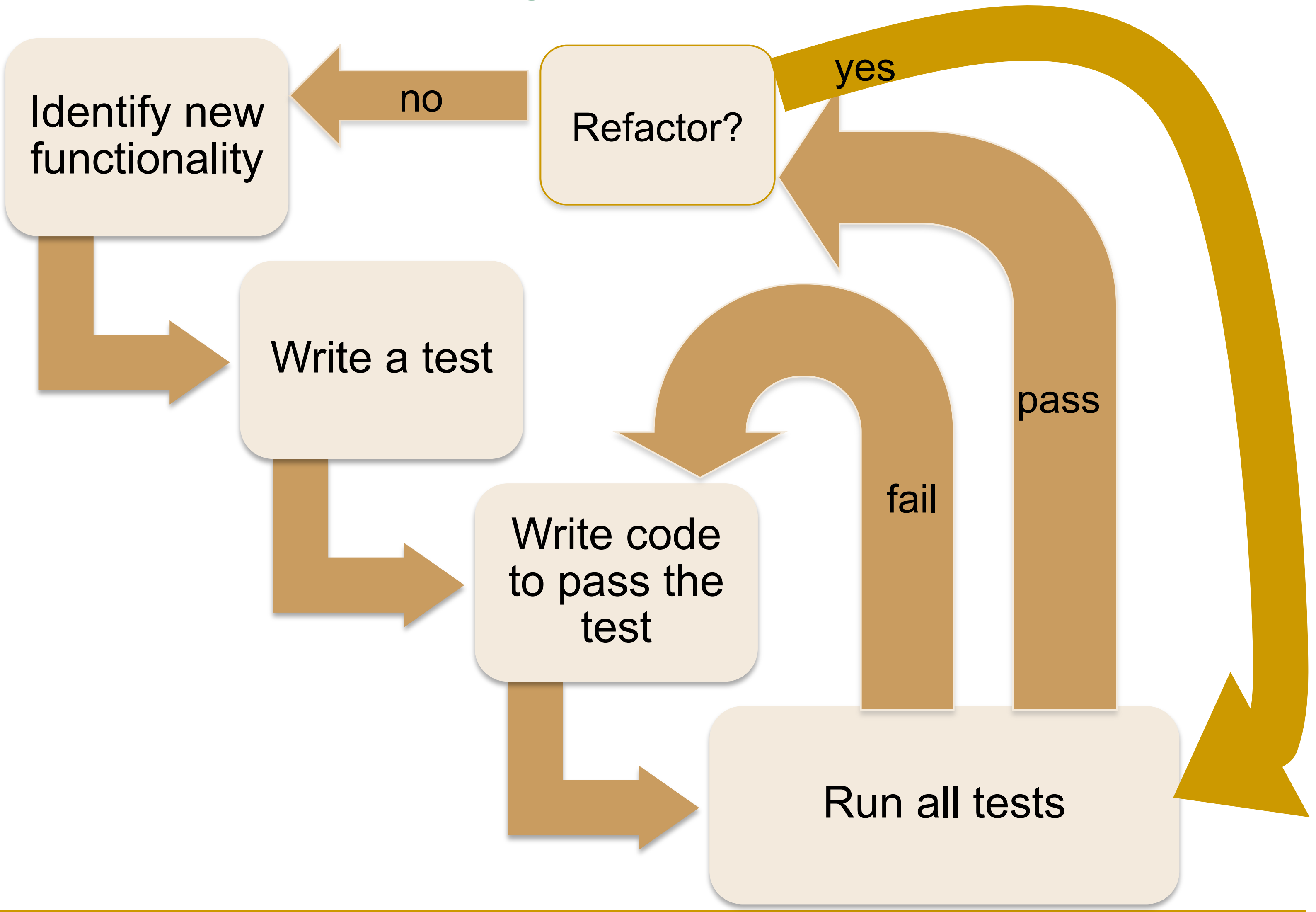
- Development method where the developer alternates development and tests
  - Objectives of the development : “pass tests”

- Incremental development
  - Write a test
  - Write the code to pass the test
  - You don’t go to next loop as long as the current increment has not validated the current test and all previous ones
- Is part of agile methods

TDD process – v0



TDD - Don’t forget to refactor!



TDD advantages

- Tests coverage
  - Each code is associated to at least one test
- Simplification of debug
  - When a test fails, the lastly added code is the reason
- Documentation
  - Tests are a kind of documentation, with usage examples!
- No regression tests
  - Are created as the program is developed

Synthesis

- Testing = running and prove errors presence
- Development tests
  - Made by the development team
  - Unit tests, component tests, system tests
- User tests
  - By potential users
  - Based on usage scenarios
- TDD
  - Use tests as a way to write only the useful code, to validate it and to document it

Junit : Java framework for tests

- Junit forge http://junit.sourceforge.net/
- Junit et Eclipse http://www.cs.uu.nl/wiki/pub/WP/TrainingInSoftwareTesting/JUnit_with_Eclipse.pdf

- Principle
  - A test class for each class
    - Eg. MyClassTest to test MyClass
  - Annotations identify test elements in this test class
    - See later for more explanations

Definition of tests in JUnit

- Test methods
  - 1 method per test
    - choosing a meaningful name is a good idea
      - Eg addTest to test method add
  - constraints
    - public void
    - No parameters, can raise exceptions
    - Annotation @Test
    - Uses assertions

Example Subscription class

```
public class Subscription {
   private int price ; // subscription total price in euro-cent
   private int length ; // length of subscription in months
   // constructor :
   public Subscription(int p, int n) {
     price = p ;   length = n ;
   }
  /**
   * Calculate the monthly subscription price in euros, rounded up to the nearest cent.
   */
   public double pricePerMonth() {
     double r = (double) price / (double) length ;                    //pb when length=0
      return r ;                                               //returns cents instead of euros
   }

  /**
   * Call this to cancel/nulify this subscription.
   */
   public void cancel() { length = 0 ; }
}
```

2 simple tests

Check PricePerMonth calculates a correct cost

Spec: Calculate the monthly subscription price in euros, rounded up to the nearest cent

- If subscription = 200 cents for a 2 months period, monthly cost should be 1€

- Monthly cost should be rounded up to the closest cent, so if cost is 200 cents for 3 months, monthly cost should be 0.67€

Class SubscriptionTest (JUnit 4)

```
import static org.junit.Assert.*;      //to access assertion methods assertxx
import org.junit.Test;                 //to get annotations @Test
public class SubscriptionTest {
  @Test
  public void testReturnEuro() {
    System.out.println("Test if pricePerMonth returns Euro...") ;
    Subscription s = new Subscription(200,2) ;
    double ppm=s.pricePerMonth();
    assertEquals(1.0, ppm) ;
  }
  @Test
  public void testRoundUp() {
    System.out.println("Test if pricePerMonth rounds up correctly...") ;
    Subscription s = new Subscription(200,3) ;
    double ppm=s.pricePerMonth();
    assertTrue(ppm == 0.67) ;
  }
}
```

Junit4 assertions – non exhaustive list

Instruction	Description
assertEquals (expected, actual)	True if expected=actual
assertEquals(expected, actual, delta)	True if expected=actual±delta
assertNull(object)	True if object=null
assertNotNull(object)	True if object not null
assertSame(expected, actual)	Checks expected and actual reference the same object
assetThat(T object, matcher<T>)	Returns the result of comparison btw first parameter and result of second
assertTrue(expression)	Checks expression is true
fail(message)	Always fails, with message shown in report

Annotations

- Annotations should be placed before the corresponding methods

Annotation	Description
@Test	This is a test method
@Before	This method must be run before each test
@After	This method must be run after each test
@BeforeClass/ @BeforeAll	This method must be run before the first test
@AfterClass/ @AfterAll	This method must be run after the last test
@Ignore	Ignore it when testing

Order of methods execution

- The @BeforeAll methods

- For each @Test method
  - @Before methods (no predefined order)
  - The @Test method
  - @After methods (no predefined order)

- The @AfterAll methods

Class SubscriptionTest V2 (JUnit 4)

```
import static org.junit.Assert.*;       //for assertion methods
import org.junit.Test;                  //for the @Test annotation
Import org.junit.Before;                //for the @Before annotation
public class SubscriptionTest {
   Subscription s1, s2;
 @Before
  public void setUp(){s1 = new Subscription(200,2) ;
     s2 = new Subscription(200,3);}
  @Test
  public void testReturnEuro() {
    System.out.println("Test if pricePerMonth returns Euro...") ;
    assertEquals(1.0, s1.pricePerMonth()) ;
  }
  @Test
  public void testRoundUp() {
    System.out.println("Test if pricePerMonth rounds up correctly...") ;
    assertTrue(s2.pricePerMonth() == 0.67) ;
  }
}
```

Test exceptions - Subscription class v2

```
public class Subscription {
   private int price ; // subscription total price in euro-cent
   private int length ; // length of subscription in months
   // constructor :
   public Subscription(int p, int n) {
     price = p ;   length = n ;
   }
  /**
   * Calculate the monthly subscription price in euros, rounded up to the nearest cent.
   */
   public double pricePerMonth() throws ArithmeticException {
     if (length==0) throw new ArithmeticException(“divide by zero”);
     double r = (double) price / (double) length ;
     return r ;                                   //returns cents instead of euros
   }

  /**
   * Call this to cancel/nulify this subscription.
   */
   public void cancel() { length = 0 ; }
}
```

Testing exceptions

- JUnit 4

```
@Rule public final ExpectedException exception=ExpectedException.none();
@Test
public void testZeroDuration(){
        s=new Subscription(2,0);
        exception.expect(ArithmeticException.class);
        double x=s.pricePerMonth();
}
```

- JUnit 5

```
@Test
public void testZeroDuration(){
  s=new Subscription(2,0);
  Throwable e = assertThrows (ArithmeticException.class, () -> s.pricePerMonth());
  assertEquals("divide by zero", e.getMessage());
}
```

Execute a set of test classes

- Tests can form a « suite »
  - Several test classes, testing several classes
- Use a specific class usually called AllTests

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ MyClassTest.class, MyOtherClassTest.class })
public class AllTests {
}
```

Run tests

Command line:

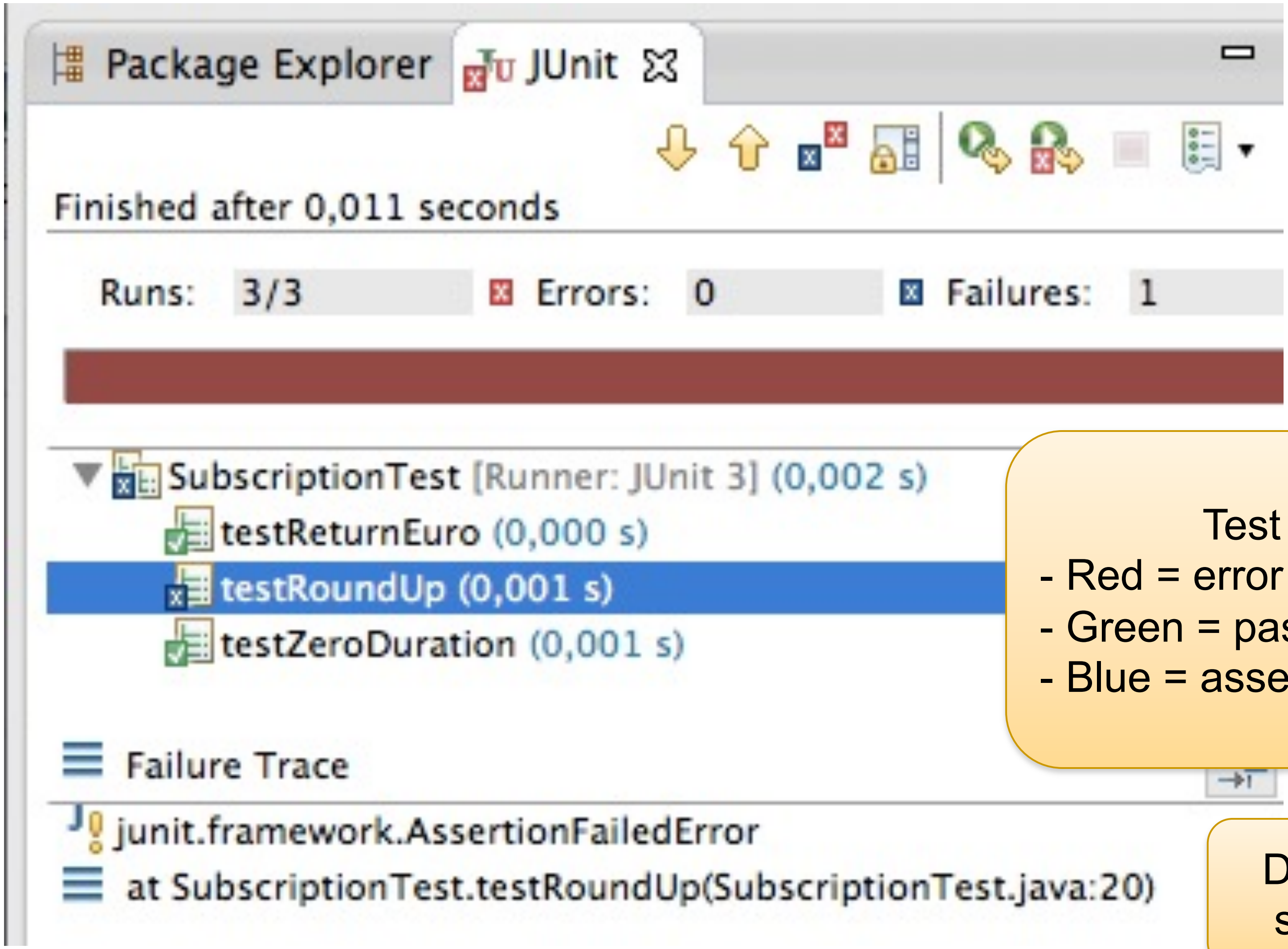
```
prompt> javac -classpath .;<full path to JUnit.jar>
SubscriptionTest.java

prompt> java  -classpath .;<full path to JUnit.jar>
org.junit.runner.JUnitCore SubscriptionTest
```

In Eclipse:

Select the test class

menu run > run as…> Junit Test

Eclipse example



```
Finished after 0,011 seconds

Runs: 3/3    Errors: 0    Failures: 1

SubscriptionTest [Runner: JUnit 3] (0,002 s)
    testReturnEuro (0,000 s)
    testRoundUp (0,001 s)
    testZeroDuration (0,001 s)

junit.framework.AssertionFailedError
at SubscriptionTest.testRoundUp(SubscriptionTest.java:20)
```

Test report:
- Red = error
- Green = passed
- Blue = assertion failed

Details on the selected line