

# COPENHAGEN BUSINESS ACADEMY



## Testing Software

## Why Test

**Releasing software with built in defects can lead to the loss of customers and credibility, and in worst cases to the loss of human lives**

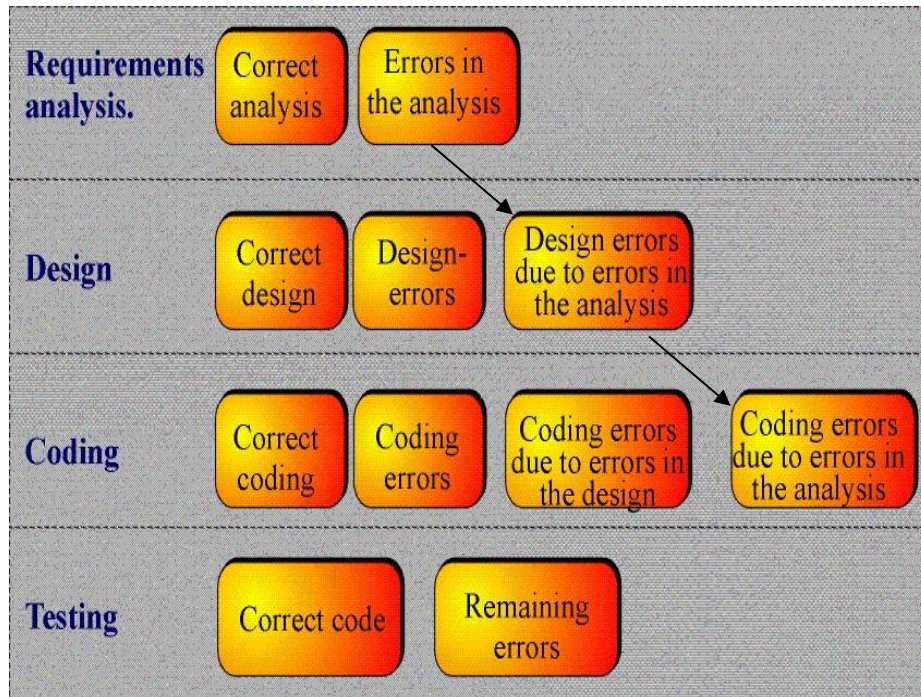
- **On June 4, 1996 the European Space Agency launched the Ariane 5 Rocket. As a consequence of a software fault, the rocket crashed just after lift-off. The loss was about \$500 million, making this the most costly (known) software fault to date.**
- **During the 1991 Gulf War, a Scud missile failed and struck a barracks in Saudia Arabia. 28 Americans where killed and 98 wounded. The cause was a software defect in the missile's control software.**
- **Between 1985 and 1987 at least two patients died as a consequence of severe overdoses of radiation in a medical linear accelerator. Again the cause was a fault in the control software.**
- .....  
.....

# When to Test

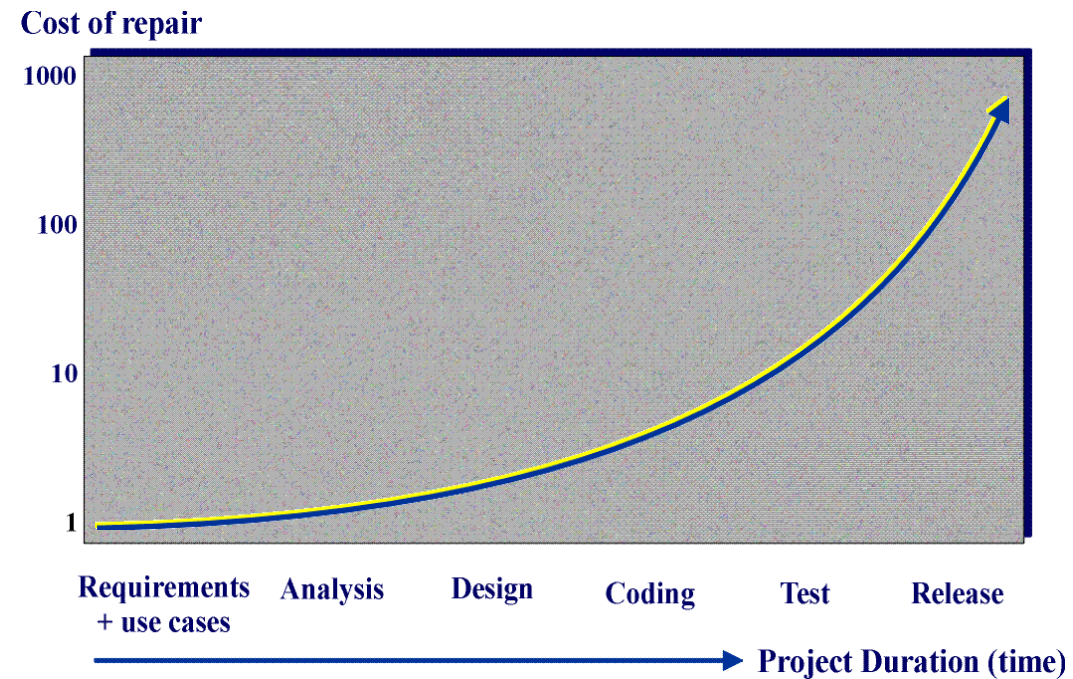
Lessons learned from the old waterfall model

Test Early, Test Often 😊

## Accumulating Errors

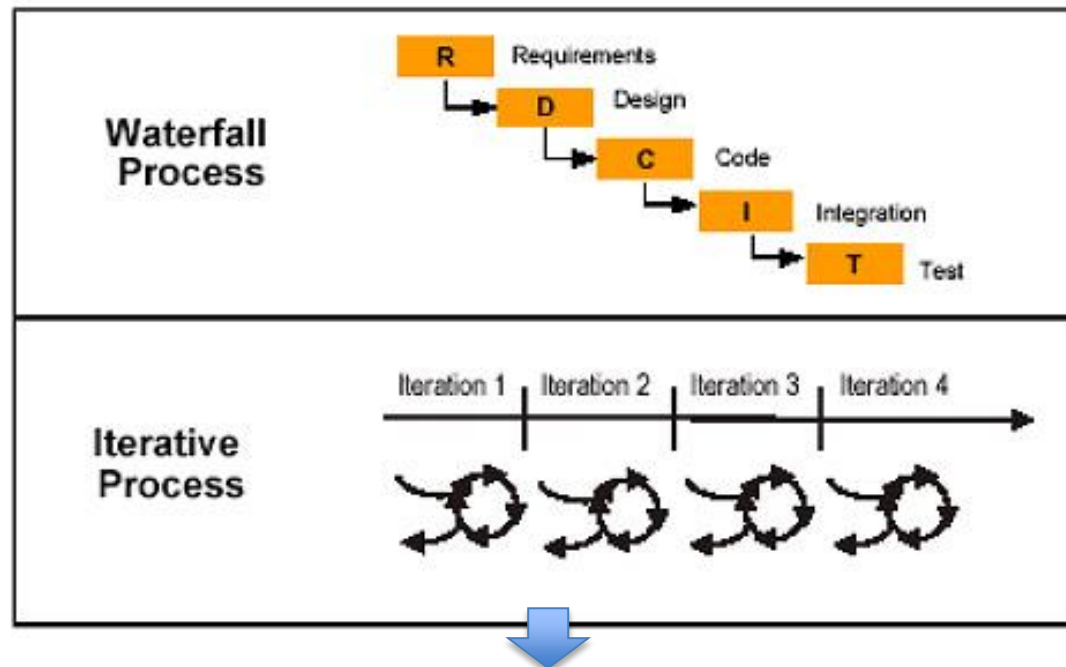


## Cost of Repair



# Iterative software processes cphbusiness

Modern Iterative and Incremental development processes greatly reduces the problem with accumulating errors by breaking the process up into a number of smaller "waterfalls".



Each iteration will increase the amount of test cases

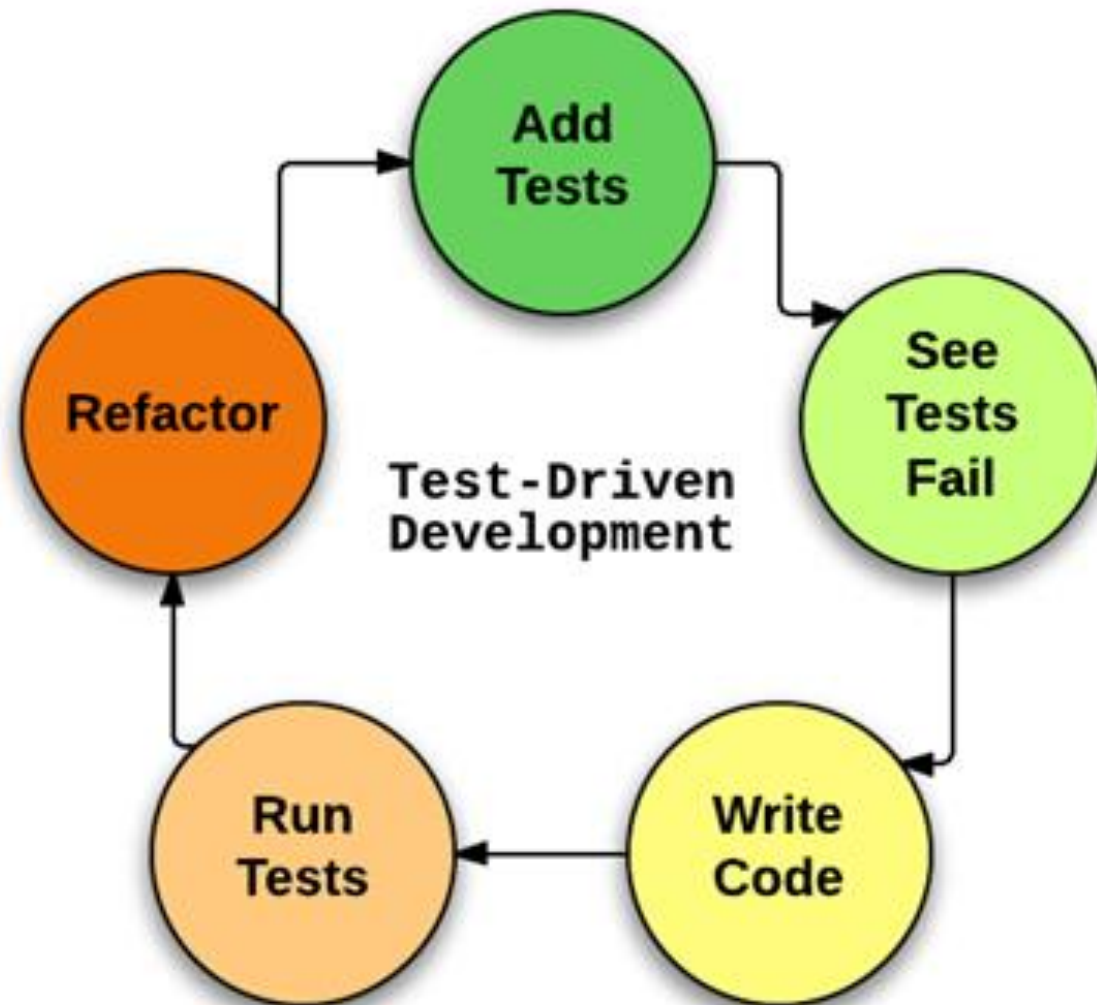


# Test Driven Development TDD

We will use this document for our discussion:

<http://agiledata.org/essays/tdd.html>

Start before any code exists



# Why do we Test ?

- The goal of testing is to show the absence of errors
- The goal of testing is to show the presence of errors

?

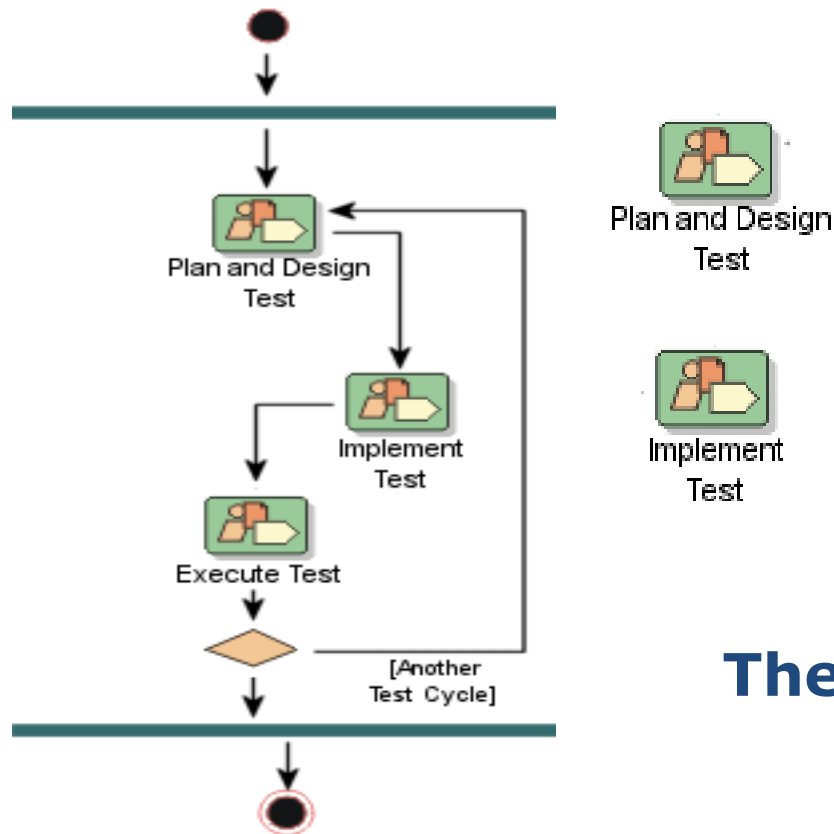
Testing is the process of executing a program with the ***intention of finding errors***

## The intellectual Part

- **Select what is to be identified by the test**
- **Decide how the test should be carried out**
- **Develop the test cases**
- **Determine what the expected or correct result of the test should be.**

## The Trivial Part (calls for automation)

- **Execute the test cases**
- **Compare the results of the test with the expected result of the test.**



  
Plan and Design  
Test

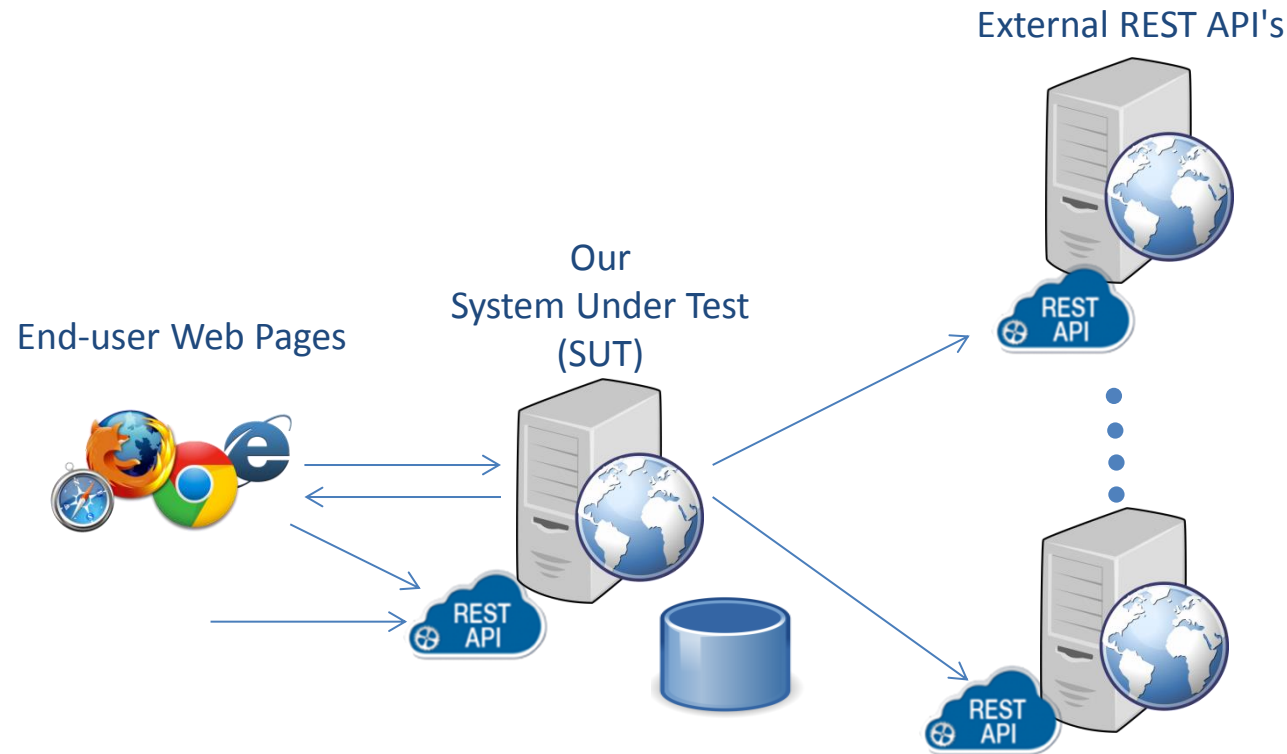
  
Implement  
Test

  
Execute Test

# Testing this Semester

## A typical 3.semester (and real life ;-) ) architecture

(Think Momondo, Hotels.com, Booking.com ...)

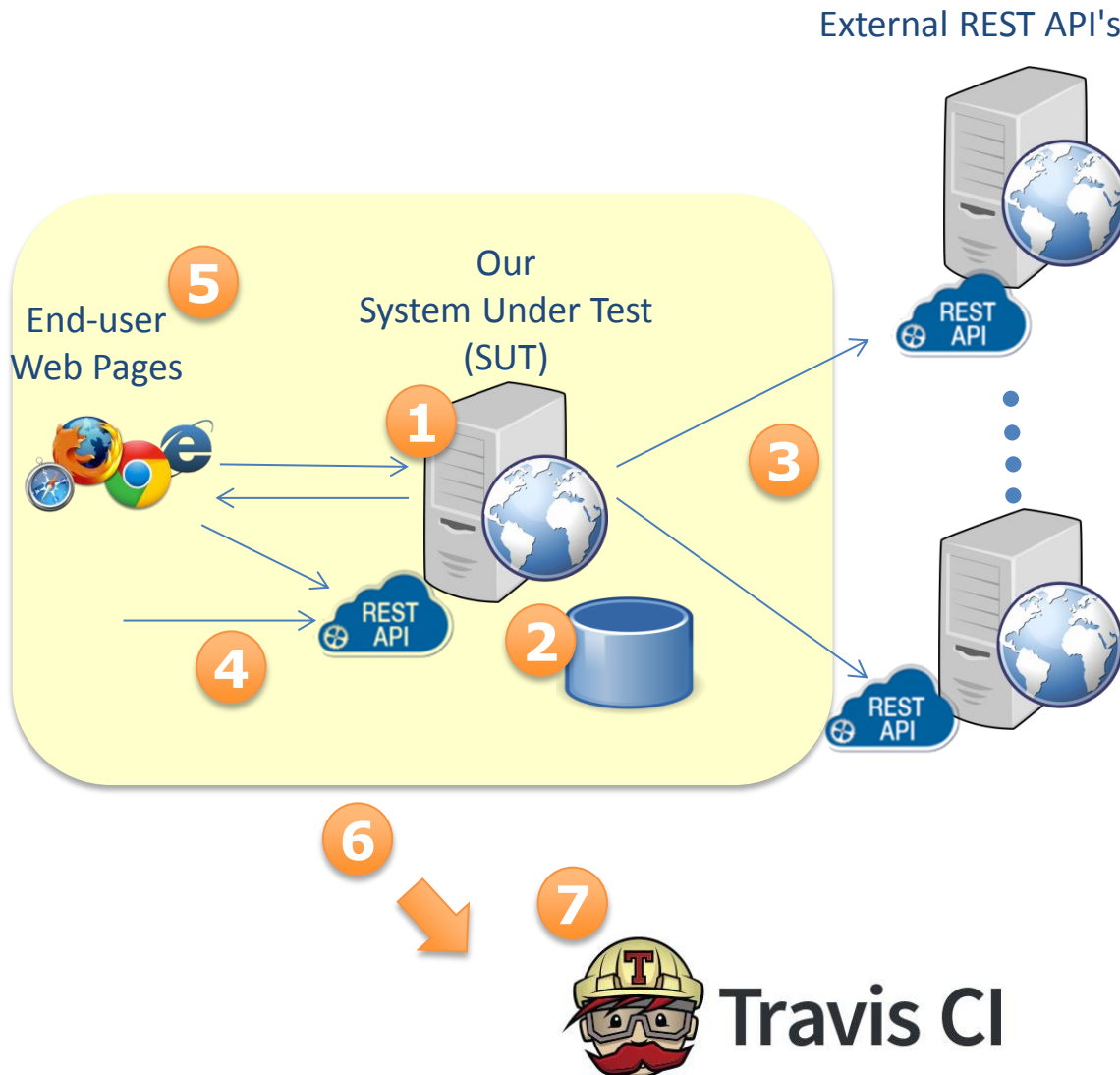




# Testing this Semester -2

## Required Testing for the Semester Project

As a minimum we expect that you will:



- 1 Unit Test relevant methods
- 2 Mock Away External DataBase Dependencies
- 3 Mock Away External HTTP Dependencies
- 4 Test your REST API
- 5 Test Your WEB – Front end using a relevant framework
- 6 Automate the full Test Process using Maven
- 7 Use our Automated Tests, to do Continuous Integration

# Unit Testing

## Unit Testing, one (more) definition:

Unit tests focus on (single classes). They exist to make sure that **your code** works.

They control all aspects of the context in which the class to be tested is executed, by replacing real collaborators with **test doubles**.

They know nothing about the users of the system they put to the test, and are unaware of layers, external systems and resources.

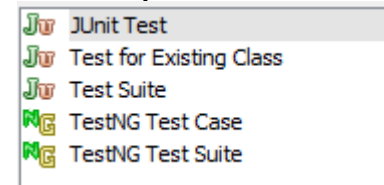
Ref: <http://practicalunittesting.com/>

# Unit Test With JUnit

We Expect you know JUnit and how to use it from previous semesters, but for Maven projects you should either:

- 1 Unit Test relevant methods
- 2 Mock Away External DataBase Dependencies
- 3 Mock Away External HTTP Dependencies
- 4 Test your REST API
- 5 Test Your WEB – Front end using a relevant framework
- 6 Automate the full Test Process using Maven
- 7 Use the Automated Tests to do Continuous Integration

In NetBeans right-click your project tab and select your test type (this will automatically insert the required dependencies in your POM-file)



Manually insert the Dependencies into your POM:

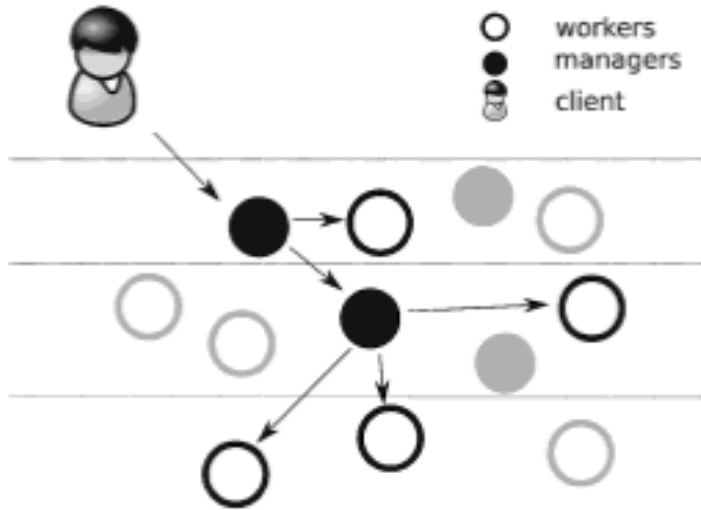
```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-core</artifactId>
  <version>1.3</version>
  <scope>test</scope>
</dependency>
```

You already know about unit-testing, Junit and how to use it.

So today's focus will be on how to unit test code with complex dependencies, like external REST calls, a Database etc.

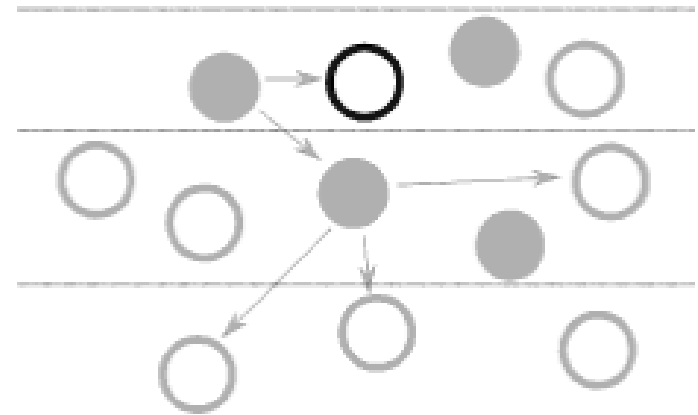
# Unit Testing - Challenges

## A typical OO System Abstraction



Typically in an OO-design a single request is solved by delegating subtasks to a number of objects. In the figure, circles represent objects, arrows are messages being passed between them and lines represent layers (View, Services, DAO etc.)

## Scope of a Unit Test



Unit test representation of the system, with only one element (the SUT) visible. The greyed out elements symbolize those parts of the system not touched fully by the test or replaced by test doubles. A unit test is always located inside a single layer.

Ref: <http://practicalunittesting.com/>

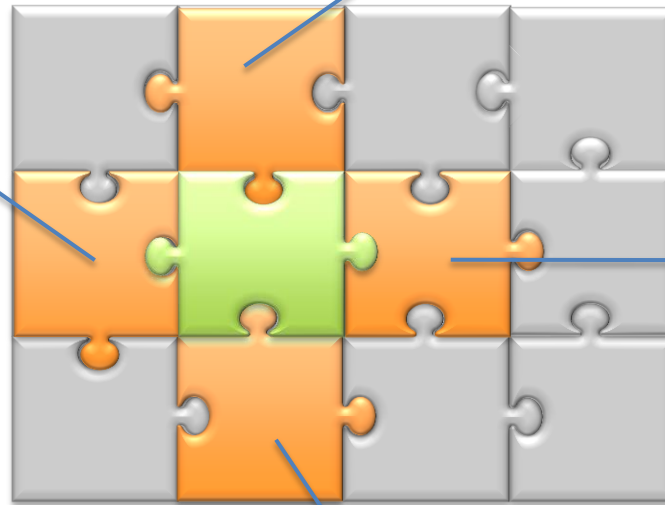
# Unit Testing - Challenges

When testing a class  
What if this class depends  
on classes/systems that?



Supplies non-deterministic results  
(current time/date/temperature  
etc.)?

Is not yet created?



Is complex and itself relies on  
external resources (web-  
service calls, file-I/O, external  
hardware etc.)?

Relies on a Database (takes a long time  
to start, could be on a remote server,  
must be kept clean etc.)?



Class in focus



Dependencies



Unrelated classes



# Testing systems with dependencies cphbusiness

Clean code is code that does one thing well  
*Bjarne Stroustrup*

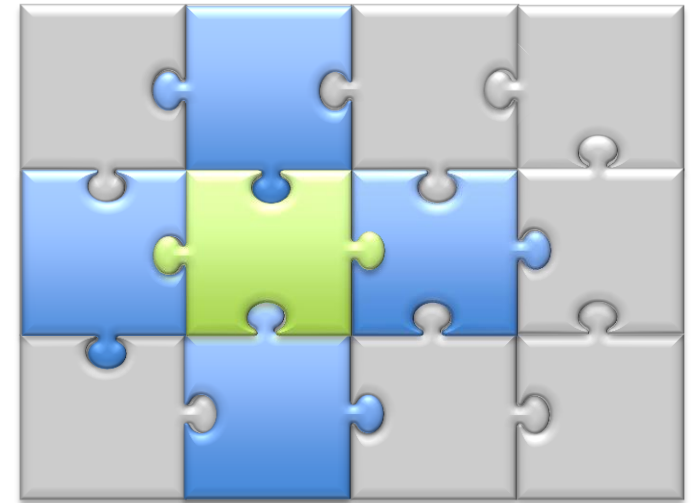
Clean tests are focused tests that fail for only one reason  
*Petri Kainulainen*

How to get Clean Code and Clean Tests →  
Surround Objects Under Test with predictable  
**test doubles**

## Test Doubles

In general, test doubles are used to replace  
DOC's allowing us to:

- Gain full control over the environment in which the SUT is running
- Verify interactions between the SUT and its DOC's



Class in focus (SUT)



Depending On Component (DOC)



Unrelated classes

# Vocabulary Recap

## SUT

System Under Test. The part of the system being tested. Depending on the type of test the granularity can range from a single class to the complete system

## DOC

Depend On Component. Any entity that is required by the SUT to fulfill its duties

## Test Double

Test Doubles are used to replace DOC's allowing us to:

- Gain full control over the environment in which the SUT is running
- Verify interactions between the SUT and its DOC's

# Kinds of Test Doubles

## Mocks, Fakes, Stubs and Dummies ...

It is very easy to get confused about the terminology related to **Test Doubles**, since many articles use different terms to mean the same thing, and sometimes even mean different things for the same term ;-)

These are some of the terms you will meet "out there".

Dummy Objects, Mocks, Stubs, Fakes, Spy Objects etc..

This semester we will use the term double or Mock as something covering "all of it"

Next page, goes in details (if you insist ;-)

# Kinds of Test Doubles

## Mocks, Fakes, Stubs and Dummies ...

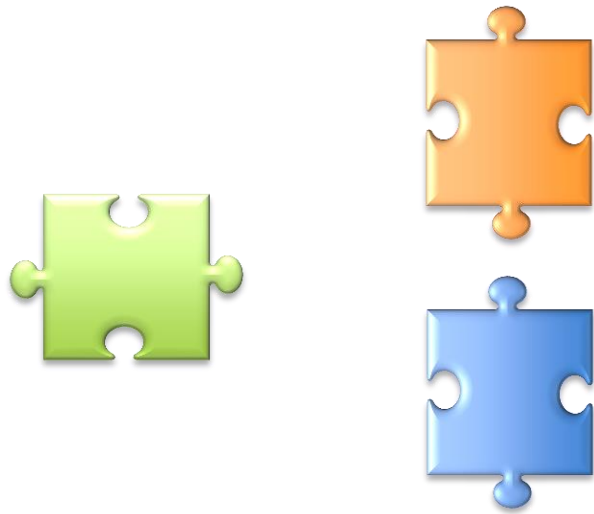
**Fake objects** have working implementations, but usually take some shortcut which makes them not suitable for production

**Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.

**Mock Objects** are objects pre-programmed with expectations which form a specification of the calls they are expected to receive

**Dummy Objects** are objects passed around but never used, i.e., its methods are never called (for example be used to fill the parameter list of a method)

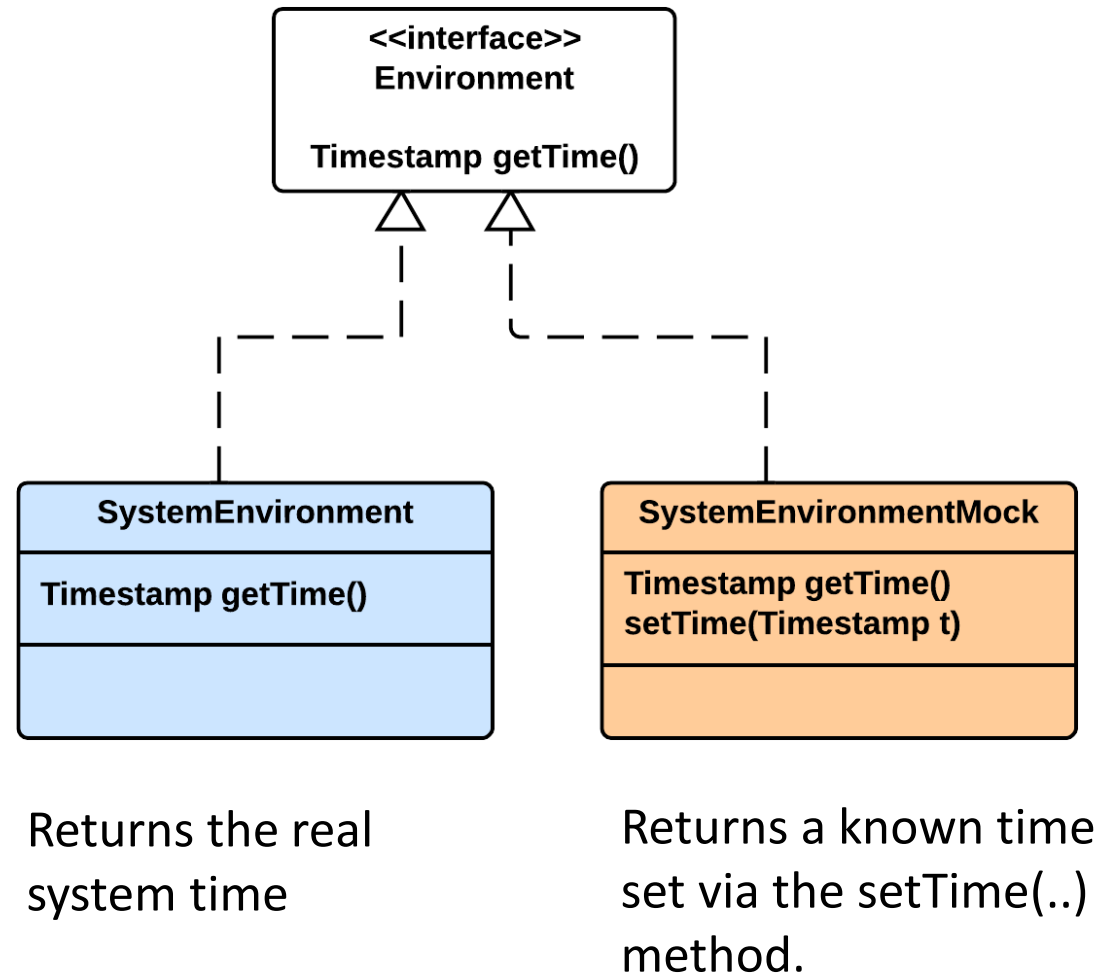
# Unit Tests and Mock Objects



Class in focus

Mocks for the Unittest

Real dependency class



# Mock example

```
public interface Environment {
    Timestamp getTime();
}

public class SystemEnvironment implements Environment {
    @Override public Timestamp
    getTime() {
        return new Timestamp(new Date().getTime());
    }
}

public class SystemEnvironmentMock implements Environment {
    private Timestamp time;
    @Override public Timestamp getTime() {
        return time;
    }
    public void setTime(Timestamp time) {
        this.time = time;
    }
}
```



# How to do Mocking

One way to make absolutely sure, you would never do Unit Testing was if you had to write all the mocking code manually  
But (obviously) there are great tools out there to help us ☺

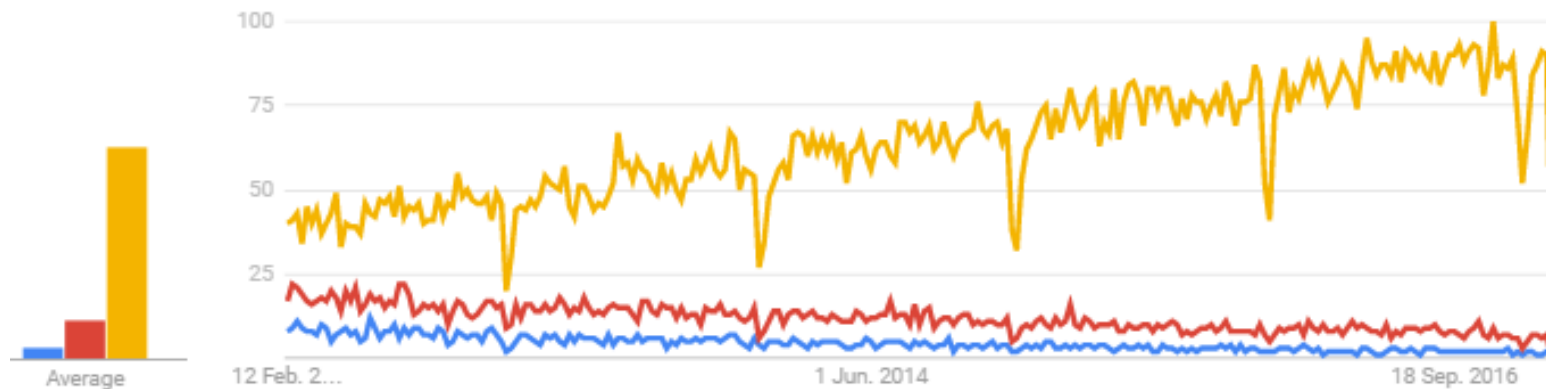
If you were a Java Based Company, trying to decide for a relevant framework, these are probably the ones that would pop up:

● JMock

● EasyMock

● Mockito

Let's see what others are using (searching for)



Live: <https://www.google.com/trends/explore?q=JMock,EasyMock,Mockito>

# Mocking with mockito

We will go for mockito 😊



Massive StackOverflow Community voted Mockito the best mocking framework for java.

Top 10 Java library across all libraries, not only the testing tools.

Ref: <http://mockito.org> (front page)

# Mocking – Using Mockito

## Quick getting started

Add this entry to your POM file

```
<dependency>  
  <groupId>org.mockito</groupId>  
  <artifactId>mockito-all</artifactId>  
  <version>1.9.5</version>  
  <scope>test</scope>  
</dependency>
```

# Mocking – Using Mockito

## Quick getting started

```
@RunWith(MockitoJUnitRunner.class)
public class MockitoTest
{
```

This ensures that mocked values will be populated

```
    @Mock
    ArrayList<String> listMock;
```

Mock the ArrayList instance

```
    @Test
    public void testAppend() {
        //Alternative way of creating the Mock
        // ArrayList<String> listMock = mock(ArrayList.class);
        when(listMock.get(0)).thenReturn("Hello");
        when(listMock.get(1)).thenReturn("World");
```

Configure which values are returned by the mock

```
        String res = listMock.get(0);
        assertEquals("Hello", res);
        res = listMock.get(1);
        assertEquals("World", res);
```

```
        verify(listMock, times(2)).get(anyInt());
```

Verify that get(..) was called 2 times

```
    }
}
```

# Mocking – Using Mockito



We will do a Quick Demo together.

Clone this project: <https://github.com/Lars-m/sillygame.git>

- After that you should go through this tutorial:  
<http://www.vogella.com/tutorials/Mockito/article.html#target>
- And complete the exercises

1 Unit Test relevant methods

2 Mock Away External DataBase Dependencies

3 Mock Away External HTTP Dependencies

4 Test your REST API

5 Test Your WEB – Front end using a relevant framework

6 Automate the full Test Process using Maven

7 Use the Automated Tests to do Continuous Integration

We have seen that when testing a complex class it could rely on external classes that:

- Supplies non-deterministic results (current time/date etc.)
- Is not yet created
- Is complex and itself relies on external resources (web-service calls, file-I/O, external hardware etc.)
- Relies on a Database (takes a long time to start, could be on a remote server, must be kept clean etc.)

Writing Mock Object can be simplified a lot, using a Mocking Framework. We suggest two strategies for this semester.

- **Mockito** as your Mocking Framework
- Use and **in-memory database** to Mock away external Database dependencies



# Mocking the Database



If you write your database code interface-based, with testing in mind, it should always be possible to mock the real database operations.

You can create you own facades and mock those during testing

And, since the EntityManager in JPA is an Interface, you can even mock the EntityManager 😊

# Mocking the Database

## Using an in-memory database

Mocking database operations the traditional way can be very cumbersome and time consuming.

If we limit our goals to: *we want **Database Unit Test Cases** that can be executed immediately after a project is cloned, without the need to set up a local/external-test database, we can "cheat" and use an in-memory database for testing*

System Under Test  
(SUT)



Real database



In-memory Database  
created by the test

We can use the **Apache Derby Database** for this purpose, since it can be run embedded (in-memory), which means: No need for database installation, setup, start, external database files etc.

All it requires is this entry in your POM-file and a new Persistence Unit in your Persistence.xml

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derby</artifactId>
  <version>10.12.1.1</version>
</dependency>
```

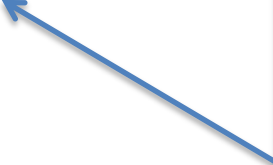
# Mocking the Database

## Using an the Derby in-memory database

You can have more than one persistence-unit in your persistence.xml, as long as you supply each with a unique subpackage name (see below)

```
<persistence ver.....>
  <persistence-unit name="pu_dev" transaction-type="RESOURCE_LOCAL">
    <!--the original Persistence Unit File -->
    <properties>
      .....
      <property name="eclipselink.canonicalmodel.subpackage" value="development"/>
    </properties>
  </persistence-unit>
```

### This Is the new PU you should add to your persistence.xml



```
<persistence-unit name="pu_test" transaction-type="RESOURCE_LOCAL">
  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
  <class>entity.Person</class> <!-- REPLACE THIS WITH YOUR OWN ENTITY CLASSES -->
  <exclude-unlisted-classes>>false</exclude-unlisted-classes>
  <properties>
    <property name="javax.persistence.jdbc.driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="javax.persistence.jdbc.url" value="jdbc:derby:target/testDB;create=true"/>
    <property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>
    <property name="eclipselink.ddl-generation.output-mode" value="database"/>
    <property name="eclipselink.canonicalmodel.subpackage" value="test"/>
  </properties>
</persistence-unit>
```

```
</persistence>
```