

# COPENHAGEN BUSINESS ACADEMY



## Testing

Thomas Hartmann

Litterature:

[http://en.wikipedia.org/wiki/Software\\_testing](http://en.wikipedia.org/wiki/Software_testing)

<https://github.com/junit-team/junit/wiki>

## 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**

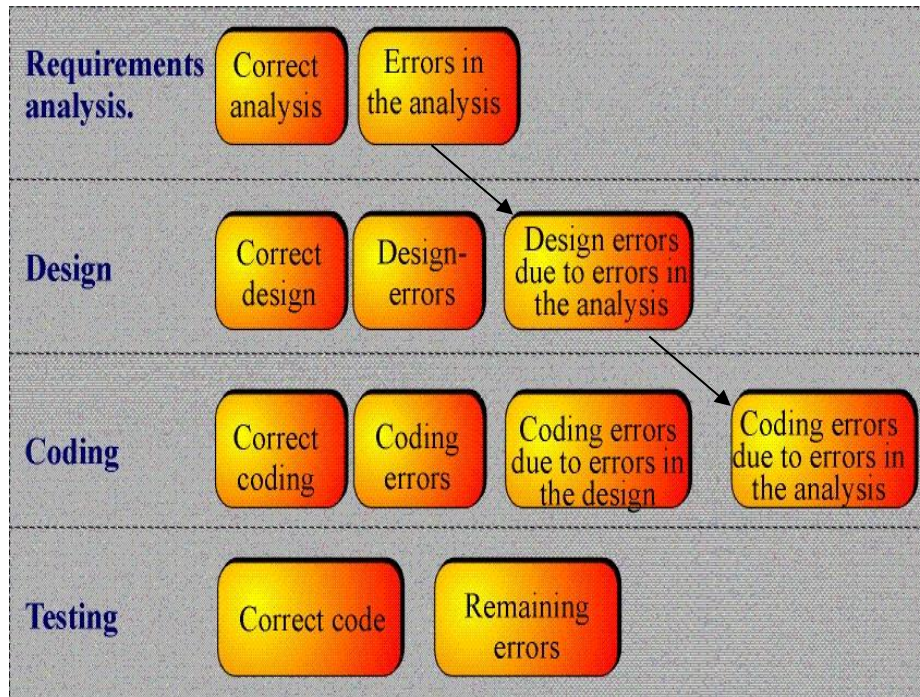
- **On June 4, 1996 the European Space Agency launched the Ariane 5 Rocket. As a consequence of a software fault, the rocket crashed just after lift-off. The loss was about \$500 million, making this the most costly (known) software fault to date.**
- **During the 1991 Gulf War, a Scud missile failed 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

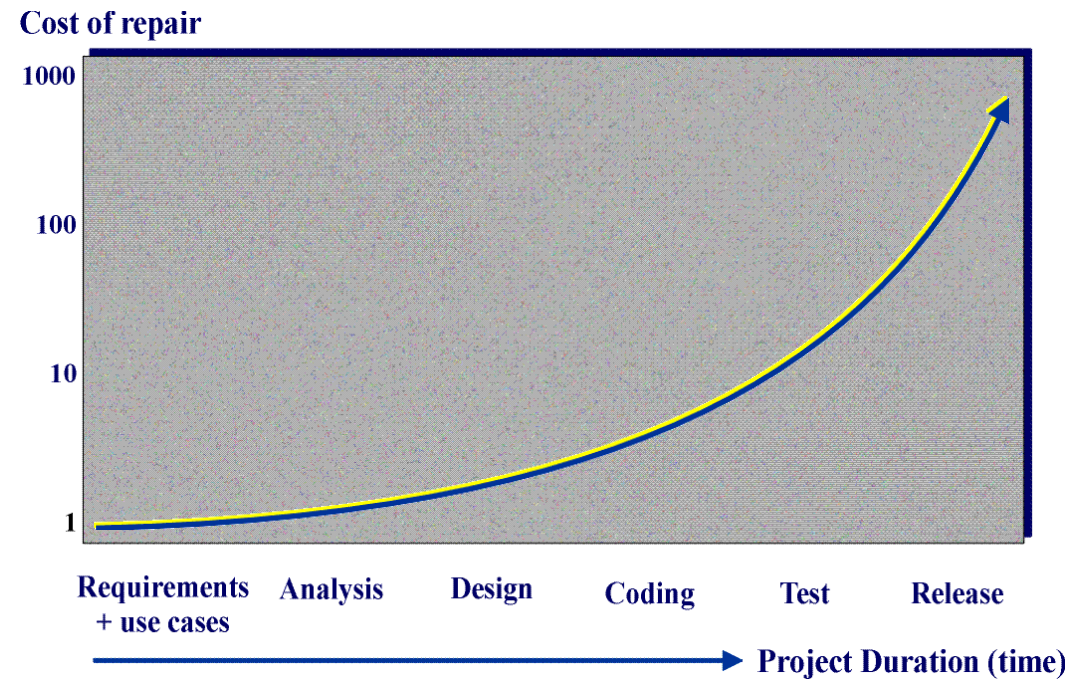
Lessons learned from the old waterfall model

Test Early, Test Often 😊

## Accumulating Errors

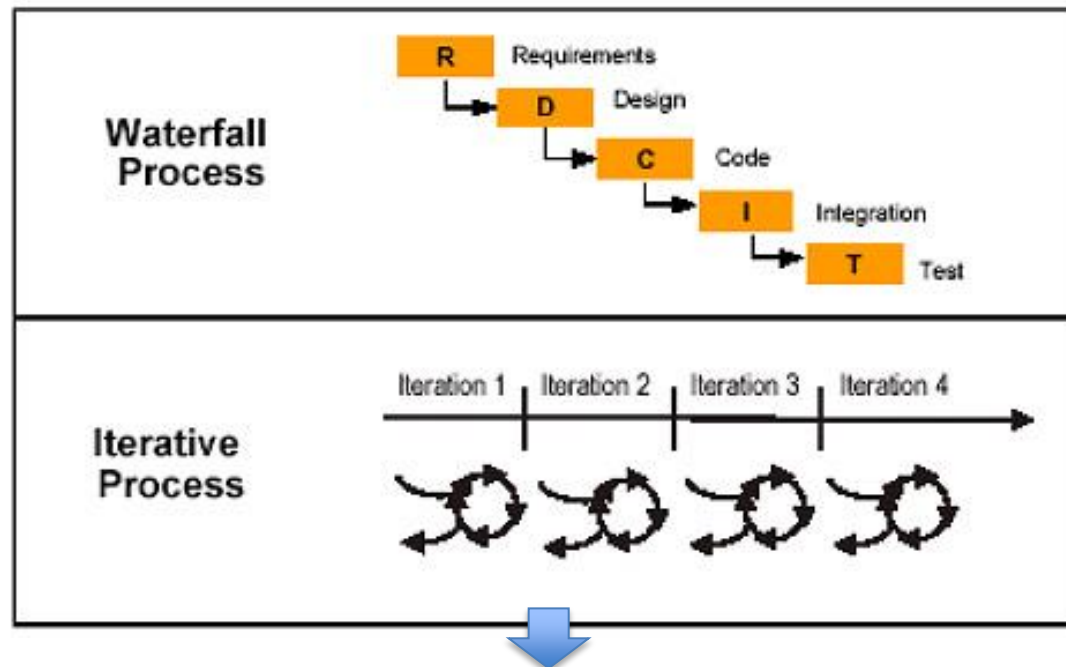


## Cost of Repair



# 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

# 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

?

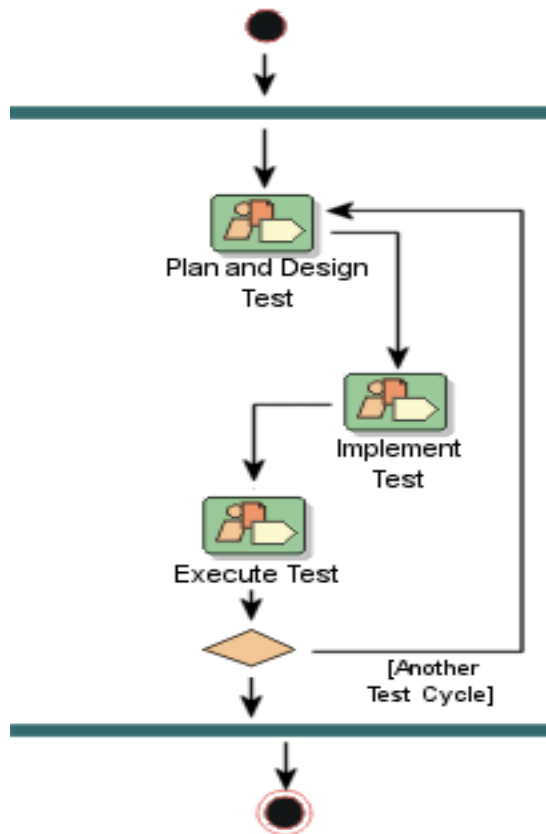
Testing is the process of executing a program with the ***intention of finding errors***

## 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



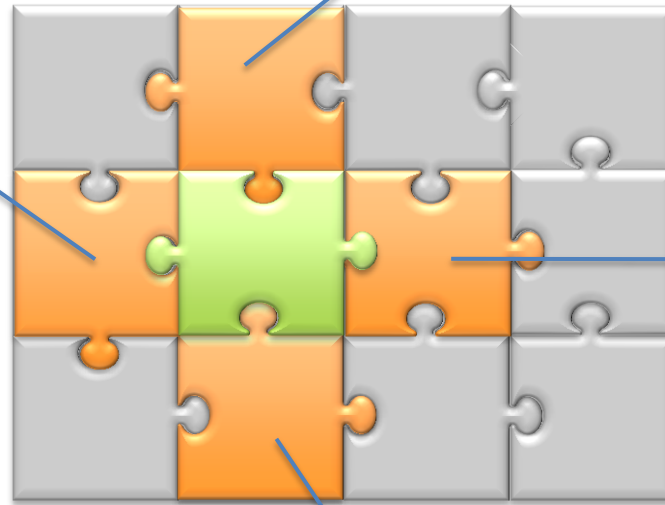
# Testing in the Large

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



Class in focus



Dependencies



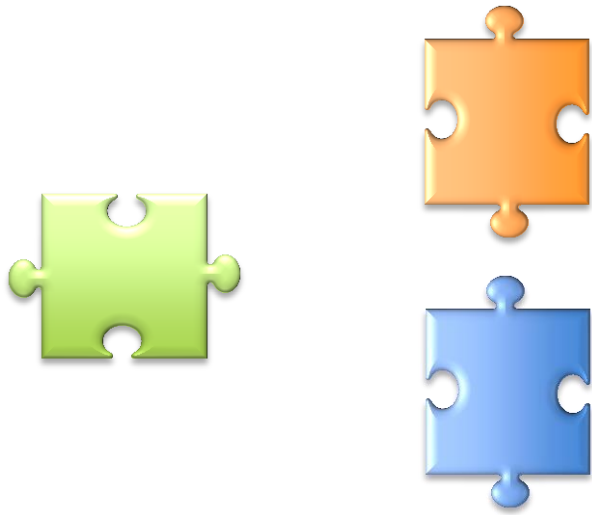
Unrelated classes

- Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.
- Mock objects have the same interface as the real objects they mimic, allowing a client object to remain unaware of whether it is using a real object or a mock object.
- Many available mock object frameworks allow the programmers to create mocks Automatically from existing code/interfaces



- Some times a distinction is made between fake and mock objects.
- Fakes are the simpler of the two, simply implementing the same interface as the object that they represent and returning pre-arranged responses.
- Thus a fake object merely provides a set of method stubs.

# Unit Tests and Mock Objects



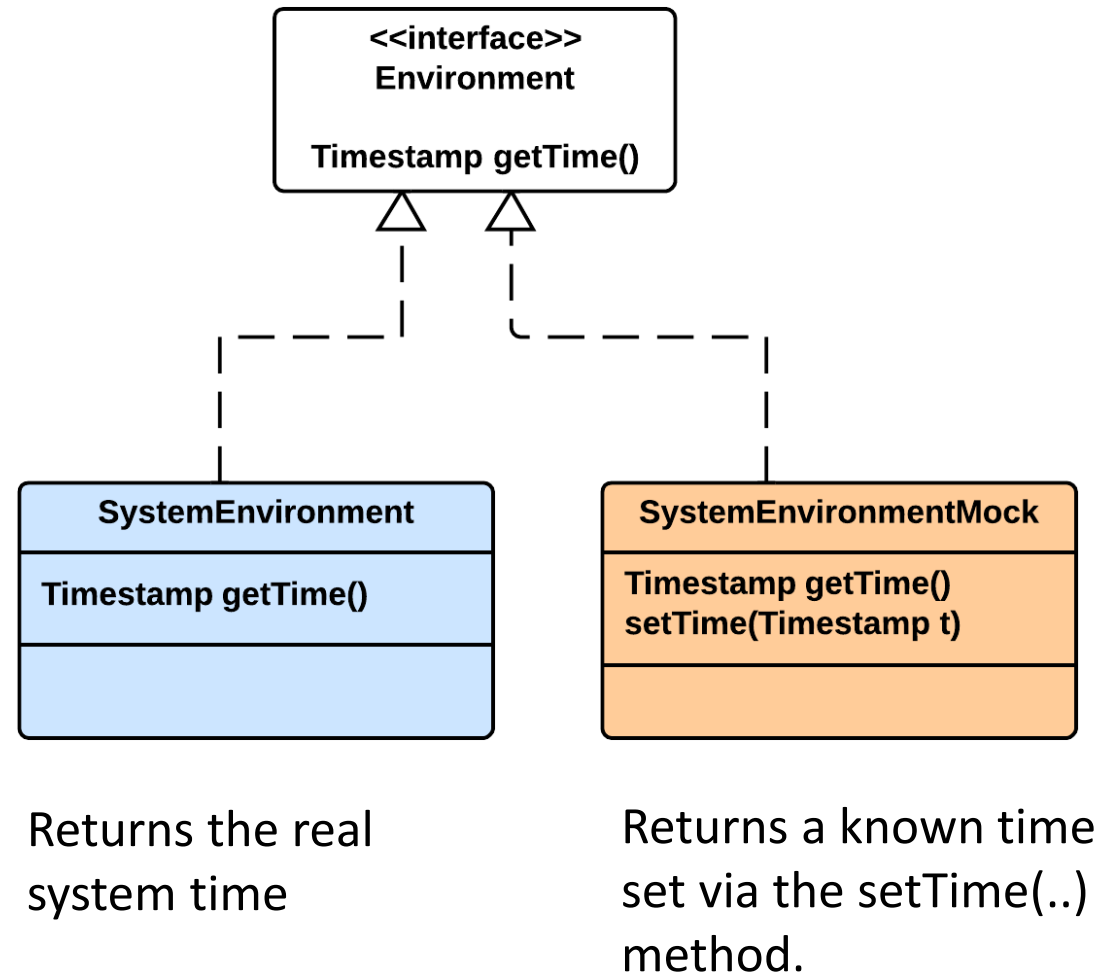
Class in focus



Mocks for the Unittest



Real dependency class



# Mock example

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

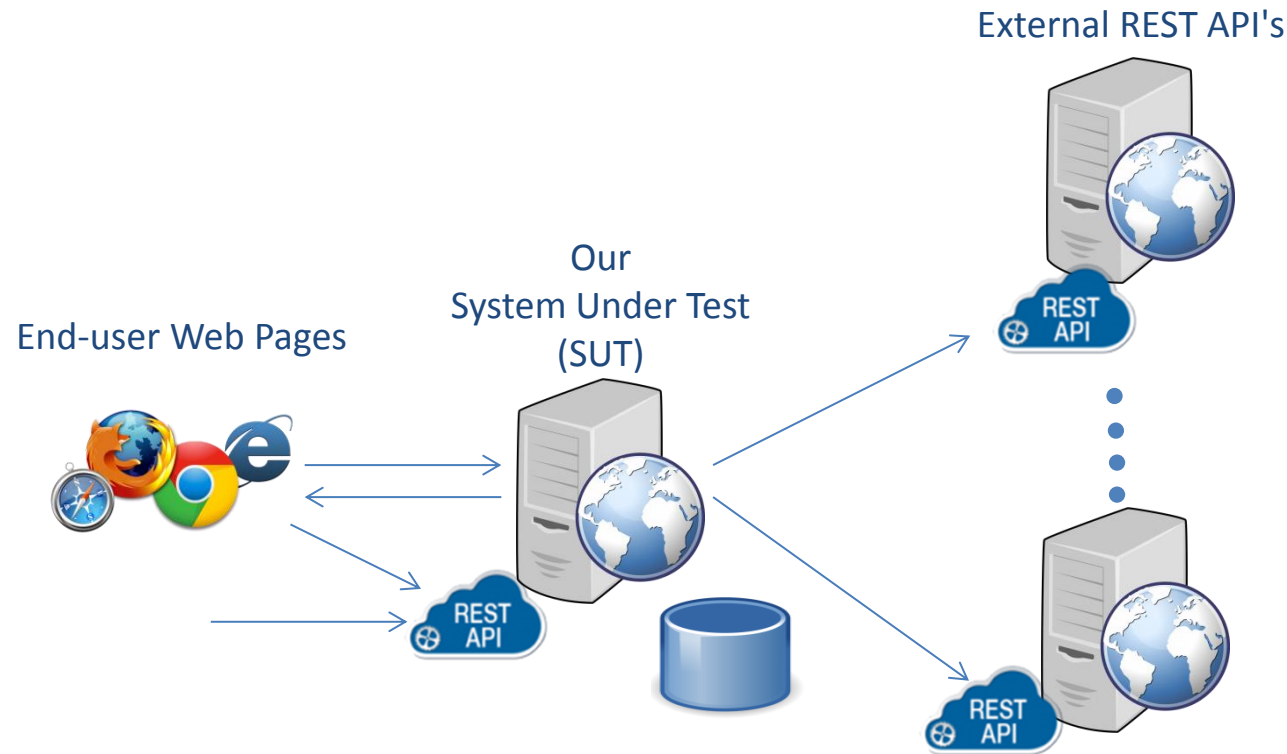
public class SystemEnvironment implements Environment {
    @Override public Timestamp
    getTime() {
        return new Timestamp(new Date().getTime());
    }
}

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

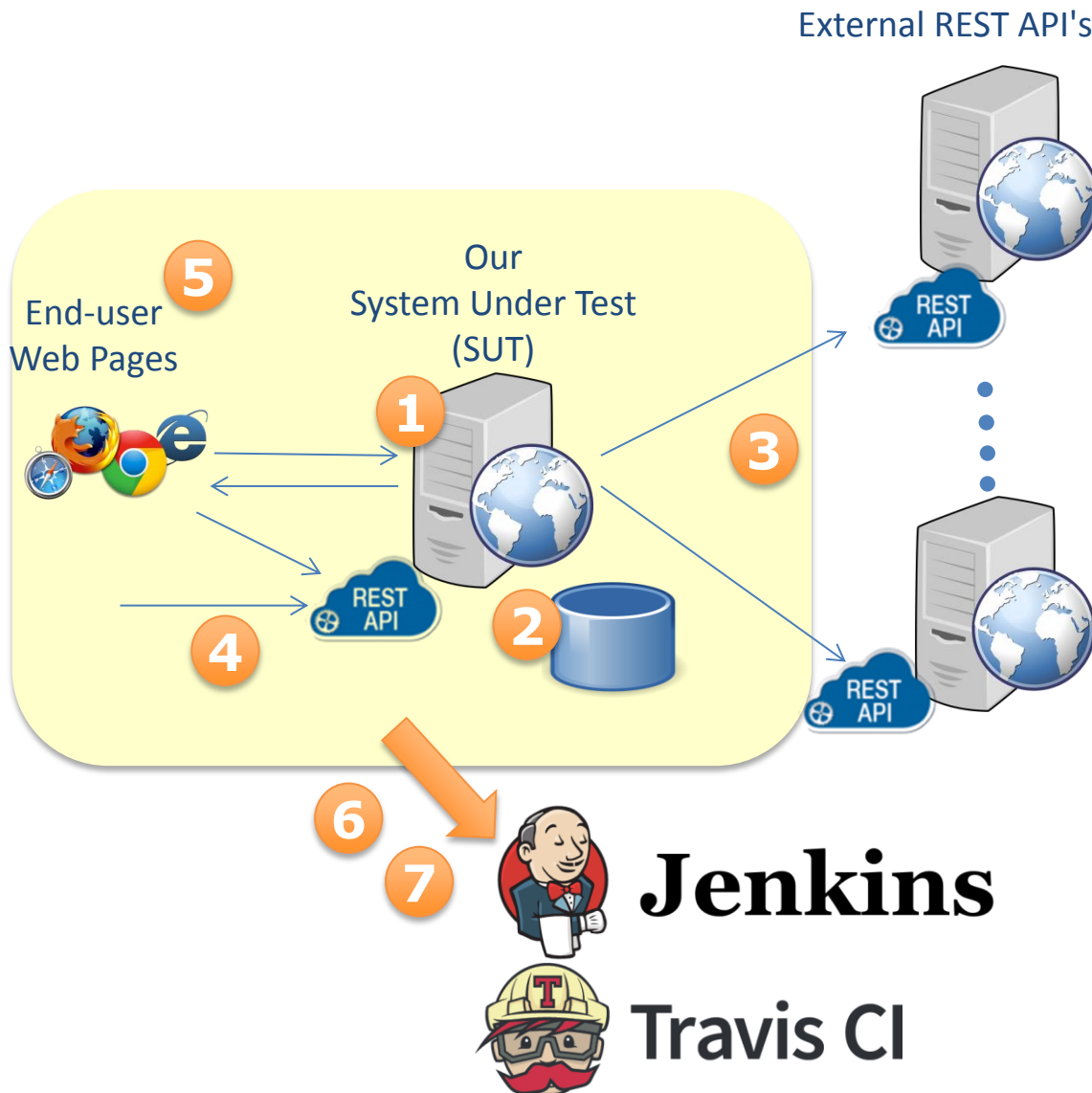
# Testing this Semester -1

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

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



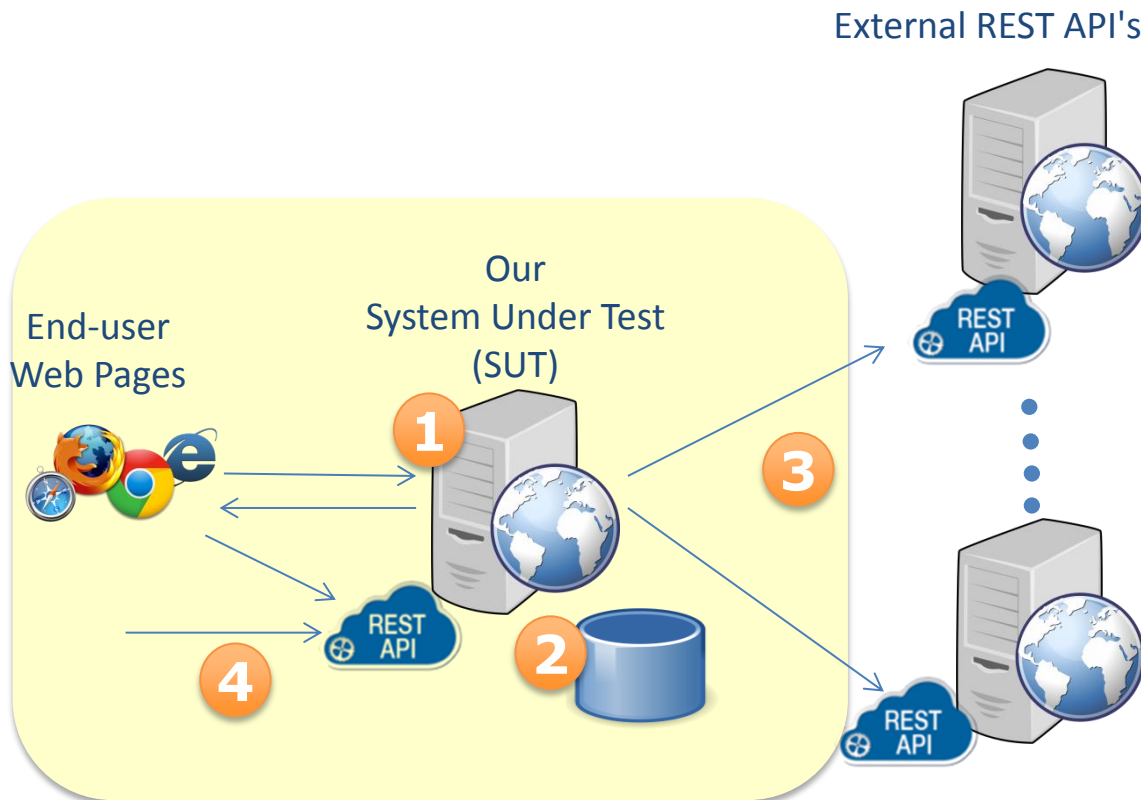
# Ideal test integrated setup for all levels



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

# Required Testing for the Semester Project



As a minimum we expect that you will:

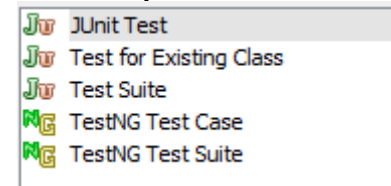
- 1 Unit Test relevant methods
- 2 Mock Away External DataBase Dependencies
- 3 Mock Away External HTTP Dependencies
- 4 Test your REST API



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>
```

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
- "Cheat" a bit ;-), and use and **in-memory database** to Mock away external Database dependencies

# Mocking framework for unit tests in Java



From the [mockito.org](https://mockito.org) front page:

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

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

In the POM.XML:

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

# Mockito example

Example taken from: [https://www.tutorialspoint.com/mockito/mockito\\_overview.htm](https://www.tutorialspoint.com/mockito/mockito_overview.htm)

```
//Creates a list of stocks to be added to the portfolio
List<Stock> stocks = new ArrayList<Stock>();
Stock googleStock = new Stock("1","Google", 10);
Stock microsoftStock = new Stock("2","Microsoft",100);

stocks.add(googleStock);
stocks.add(microsoftStock);

//Create the mock object of stock service
StockService stockServiceMock = mock(StockService.class);

// mock the behavior of stock service to return the value of various stocks
when(stockServiceMock.getPrice(googleStock)).thenReturn(50.00);
when(stockServiceMock.getPrice(microsoftStock)).thenReturn(1000.00);

//add stocks to the portfolio
portfolio.setStocks(stocks);

//set the stockService to the portfolio
portfolio.setStockService(stockServiceMock);

double marketValue = portfolio.getMarketValue();

//verify the market value to be
//10*50.00 + 100* 1000.00 = 500.00 + 100000.00 = 100500
System.out.println("Market value of the portfolio: "+ marketValue);
```

See the example project here:

<https://github.com/CphBusCosSem3/testing/tree/master/mockitoDemo/src/main/java/dk/webtrade/mockitodemo>

# 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.

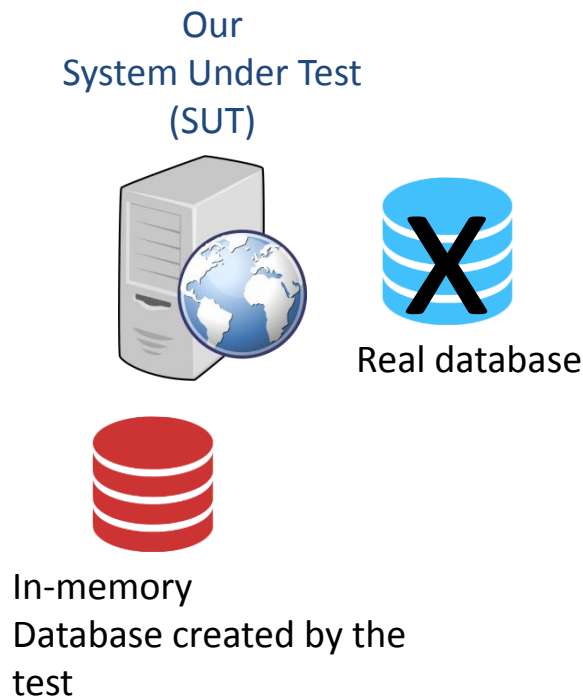
You can create you own facades and mock those during testing

# Mocking the Database

## Using an in-memory database

Mocking the 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



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

Is it difficult to get and setup?

Yes, you need to put this in your POM ;-)

```
<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_development" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entity.Person</class>
    <properties>
      .....
      <property name="eclipselink.canonicalmodel.subpackage" value="development"/>
    </properties>
  </persistence-unit>

  <persistence-unit name="pu_test" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>entity.Person</class>
    <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>
```

# Testing our REST services

## An introduction to REST assured

- 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

Testing a RESTful Service include the following checks:

- That URL addresses are constituted correctly based on the service deployment end-point and the method annotations.
- That the generated server requests calls the correct methods.
- That the API returns acceptable data. (That the data you want to examine in your PUT/GET/POST methods is valid and correct)
- That the API returns acceptable data also for **erroneous scenarios**

Since a REST API is accessed via HTTP, we need a way to programmatically perform GET, POST, PUT, DELETE etc. up against the API. We have done this manually using Postman, **but we want tests that can execute automatically.**

# Testing our REST services

## An introduction to REST assured

REST Assured is such a simple Java library for programmatically performing requests up against REST services with focus on testing.

It simplifies testing by eliminating the need for boiler-plate code to test and validate complex responses. It also supports XML and **JSON** Request/Responses.

REST Assured supports the *POST*, *GET*, *PUT*, *DELETE*, *OPTIONS*, *PATCH* and *HEAD* http methods and let us specify and validate parameters, headers, cookies and body easily.

REST Assured can be used together with JUnit.

# REST Assured

## Getting started - 1

Add the following to your pom.xml file:

```
<dependency>  
  <groupId>com.jayway.restassured</groupId>  
  <artifactId>rest-assured</artifactId>  
  <version>2.9.0</version>  
</dependency>
```

In you test-class, add the following to your imports:

```
import io.restassured.RestAssured;  
import static io.restassured.RestAssured.*;  
import io.restassured.parsing.Parser;  
import static org.hamcrest.Matchers.*;
```

This way the syntax of the framework gets cleaner to work with.

# REST Assured

## Getting started - 2

Next; you want to specify your base URI (what host you are targeting), default Parser (how is data sent to you, ie. JSON), and base path (what is the root of the REST API).

```
@BeforeClass
```

```
public static void setUpBeforeAll() {  
    RestAssured.baseURI = "http://localhost";  
    RestAssured.port = 8080;  
    RestAssured.basePath = "/Test1";  
    RestAssured.defaultParser = Parser.JSON;  
}
```

# REST Assured

## Writing Test Cases

This very simple test will verify that the REST server is running

```
@Test
public void serverIsRunning() {
    given().
    when().get().
    then().
    statusCode(200);
}
```

This works because of the `setUpBeforeAll()` method given above. We could also have written it like this:

```
@Test
public void serverIsRunningV2() {
    given().when().get("http://localhost:8080/Test2/").then().statusCode(200);
}
```



# REST Assured

## The Syntax

*given()*

Describes:

What payload our method is going to have (JSON data representing a Student)

What kind of ContentType our payload has (XML, PlainText, JSON, etc.)

What kind of authentication we are having (basic,token,Oauth, etc.)

*when()*

The method we are going to call as well as the path

*POST, GET, PUT, DELETE, OPTIONS, PATCH or HEAD*

URL path (eg. /user , /teacher/sal)

*then()*

After we executed our method, we need to test on the results we get back

Statuscode (200OK,401NotFound, etc)

Actual messages we get back. JSON,XML, etc.

# REST Assured

## The Syntax

*given()*

Describes:

- What payload our method is going to have (JSON data representing a Student)
- What kind of ContentType our payload has (XML, PlainText, JSON, etc.)
- What kind of authentication we are having (basic,token,Oauth, etc.)

*when()*

The method we are going to call as well as the path

*POST, GET, PUT, DELETE, OPTIONS, PATCH or HEAD*

URL path (eg. /user , /teacher/sal)

*then()*

After we executed our method, we need to test on the results we get back

Statuscode (200OK,401NotFound, etc)

Actual messages we get back. JSON,XML, etc.

Complete the exercise (make sure to read all the text):

## **REST testing with Rest-Assured**