Cours IUT CSID – Janvier 2012

Introduction to SpringFramework

Inversion Of Control (Dependency Injection) & Mock Testing

Arnaud Nauwynck

This document:
http://arnaud.nauwynck.chez-alice.fr/devPerso/Pres/Intro-SpringIOC.pdf

Plan

- Introduction
- Sample Spring Code
  - EJB with Spring – Java + annotations
  - Spring XML Core Syntax
  - Setup with maven dependency, run main + Junit
- Spring Inversion of Control
  - Sample ANTI-Patterns vs Spring solution
  - Spring = NEW Architecture principles
- Mock Test injection

Introduction

- Springframework is a java framework …
  - Developped by Rod Johnson & Juergen Hoeller

  - Version 1.0 in ~ 2002
  - Widely used since v2.5 (now 3.1)
  - Some alternatives: guice, plexus, EJB3, …
- World-wide Standard DE FACTO in J2EE

Game of the Name

- Spring + Framework …
- Framework = “Cadre de Travail”
  - = Way of working, proposed / imposed by library
- Spring = “ressort”, “printemps”, “renouveau”
  - After the cold winter of ugly EJB specs 1.0, 2.0, …

- aims to RE-invent the way of thinking devs
- tries to replace proprietary J2EE vendor implementations (weblogic, websphere, jonas, glassfish, …) and EJBs...

Before Spring existed...

- Code design history
  - Design Patttern (Gof), Model Driven Architecture
- Code was full of ANTI-Pattern:
  - Singleton (THE anti pattern)
  - EJB 1.0 specs (JNDI servicelocator + factory + …)
  - Spaghetti code

  - NO Inversion of Control (IOC)
  - NO Container
  - NO Junit tests

EJB 1.0 spec … = 1 line of real code / ~10 lines of technical noise

```
Properties props = new Properties();
// props.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.T3InitialContextF
// + user/password for first connection
InitialContext initialContext = new InitialContext(props);
MyEJBHome home;
try {
    Object ejbHome = initialContext.lookup("java:comp/env/fr/an/test/MyEJB");
    home = (MyEJBHome) PortableRemoteObject.narrow(ejbHome, MyEJBHome.class);
} catch (NamingException ex) {
    throw new RuntimeException("Error getting the home interface", ex);
}
MyEJB myEJB = home.create();


// *** do call EJB ***
myEJB.call();
```

EJB With Spring …

Small is beautiful : only annotated POJOs

```
import javax.annotation.Resource;
import org.springframework.stereotype.Component;

@Component
public class MySpringObj {

    @Resource
    protected MyEJB myEJB;

    public void callMyEJB() {
        myEJB.call();
    }
```

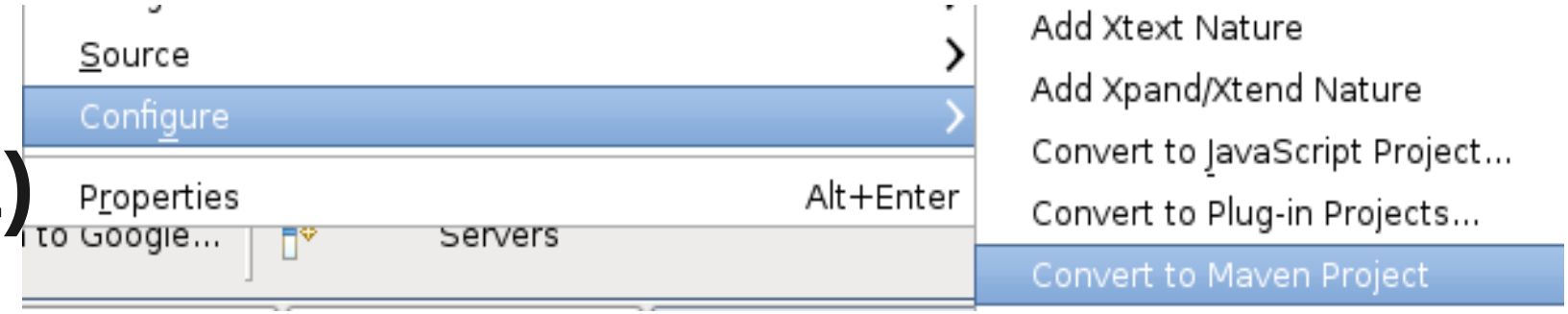
```
@Component
public class MyEJBImpl implements MyEJB {

    @Override
    public void call() {
        System.out.println("MyEJBImpl.call()");
    }

}
```

Spring XML Declaration for Annotations @Component / @Service and @Injected / @Resource

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <!-- enable annotation and dependency injection by @Resource -->
    <context:annotation-config />

    <!-- scan classes by @Component -->
    <context:component-scan base-package="fr.an.test" />

</beans>
```

Import Spring …
(easy with Eclipse + Maven + M2e)

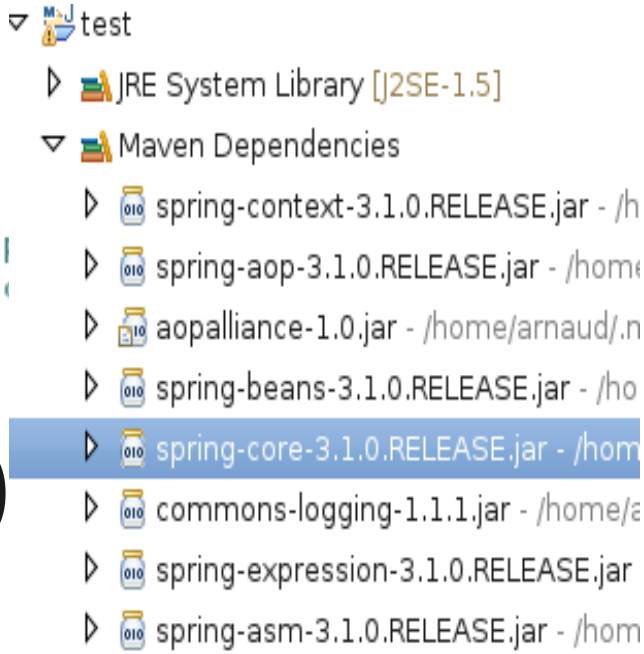
1)



2)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xm
    xsi:schemaLocation="http://maven.apache.org/POM/4
    <modelVersion>4.0.0</modelVersion>
    <groupId>fr.an.test</groupId>
    <artifactId>test-spring</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.springframework</grou
            <artifactId>spring-context</artifa
            <version>3.1.0.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

3)



The (main) Proof in the pudding...

<terminated>TestSpringMain [Java Application]

<terminated>fr.an.test.TestSpringMain at localhost:37294

test/pom.xml | applicationContext-t | TestSpringMain.java | AbstractApplicationC

```
    public static void main(String[] args) {
        // start Spring ApplicationContext
        ClassPathXmlApplicationContext ctx =
                new ClassPathXmlApplicationContext("applicationContext-test1.xml");
        ctx.start();

        // call object from outside of container
        MySpringObj obj = ctx.getBean(MySpringObj.class);
        obj.callMyEJB();
    }
```

Console | Tasks | Search | Development Mode | Call Hierarchy

```
<terminated> TestSpringMain [Java Application] /opt/devtools/jv/jdk1.6.0_27/bin/java (Jan 3, 2012 11:37:30 PM)
23:37:31.773 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instan
23:37:31.774 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instan
23:37:31.775 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instan
MyEJBImpl.call()
```

The Junit Test proof also ...

JUnit

Finished after 0.538 seconds

Runs: 1/1 Errors: 0 Failures: 0

test [Runner: JUnit 4] (0.522 s)

Failure Trace

test/pom.xml | TestSpringMain.java | MySpringTest.java

```java
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/applicationContext-test1.xml" })
public class MySpringTest {

    @Resource
    protected MySpringObj obj;

    @Test
    public void test() {
        obj.callMyEJB();
    }

}
```

Problems | Javadoc | Declaration | Console

```
<terminated> Rerun fr.an.test.MySpringTest.test [JUnit] /opt/devtools/jv/jdk1.6.0_27/bin/java (Jan 3, 2012 11:50:30 PM)
23:50:31.597 [main] DEBUG o.s.b.f.annotation.InjectionMetadata - Processing injected method of bean 'fr.an.te
23:50:31.597 [main] DEBUG o.s.b.f.s.DefaultListableBeanFactory - Returning cached instance of singleton bean
MyEJBImpl.call()
23:50:31.601 [main] DEBUG o.s.t.c.s.DirtiesContextTestExecutionListener - After test method: context [[TestCo
23:50:31.601 [main] DEBUG o.s.t.c.s.DirtiesContextTestExecutionListener - After test class: context [[TestCon
```

Spring XML Core Essentials

- Before annotations … only POJOs & Xml

```java
public class MyPOJO {

    // primitive fields
    private int fieldInt;
    private double fieldDouble;
    private String fieldStr;

    // Reference to other beans
    private Object fieldObj;

    // Special built-in types
    private List<String> fieldStrList;
    private Map<String,Object> fieldMap;
```

```xml
<bean id ="myPojo2" class="fr.an.test.MyPOJO" />

<bean id="myPojo1" class="fr.an.test.MyPOJO">
    <property name="fieldInt" value="123"/>
    <property name="fieldDouble" value="0.0456"/>
    <property name="fieldStr" value="hello"/>

    <property name="fieldObj" ref="myPojo2" />

    <property name="fieldStrList">
        <list>
            <value>hello1</value>
            <value>hello2</value>
        </list>
    </property>
    <property name="fieldMap">
        <map>
            <entry key="key1" value="value1"/>
            <entry key="key2"><value>value2</value></entry>
        </map>
    </property>
</bean>
```

SpringFramework documentation

- Spring documentation is extremely rich
  - 700 pages of nice PDFs
  - Causing a Problem ?
- Google search has billion solutions (and pbs of others...)

- Spring =
  - A Container
  - XML syntax
  - Xml helper classes
  - Java helper classes (JdbcTemplate, JMS, all J2EE libs...)

Spring Revolution
= Think code differently

- Spring is not a N+1 java library
- not only a Xml factory
  - Ok for runtime vs compile time dependencies…
  - Ok to externalize technical code outside of java
  - (never import org.springframework.*; in code)
  - … but the Xml can become worse

- Spring Dependency Injection

  = **Inversion Of Control**

  = Hollywood principle: “don't call us, we call you”

ANTI-patterns explained

… Things beginners do, things that you should NEVER do anymore

Solutions with Spring

Things you can read on smart projects

adopt the “Monkey see – Monkey do” attitude

The EVIL Singleton ANTI-Pattern

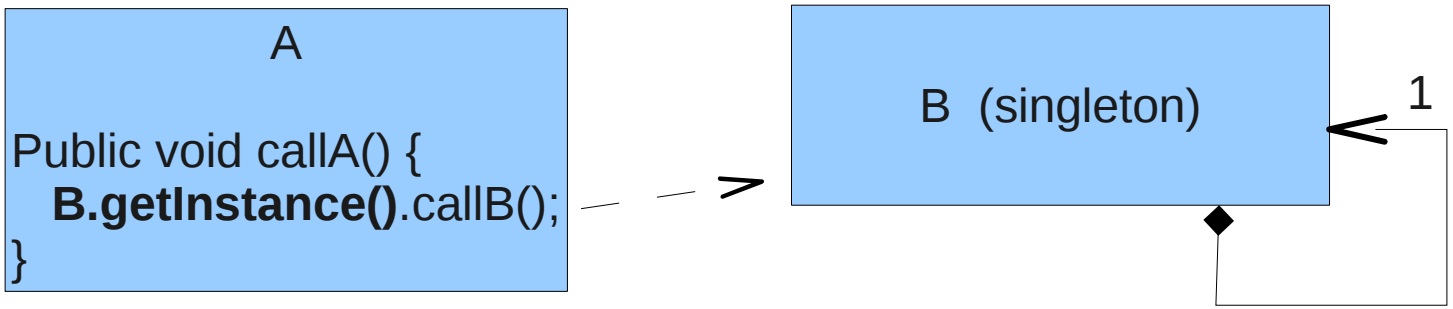
- In the GoF book,
  Singleton is one of the 23 patterns …

```
public class EvilSingleton {

    // the only instance of this class
    // no need to synchronize or lazy init + double check... java ClassLoader is OK
    private static final EvilSingleton INSTANCE = new EvilSingleton();

    public static EvilSingleton getInstance() { return INSTANCE; }

    private EvilSingleton() {}
```

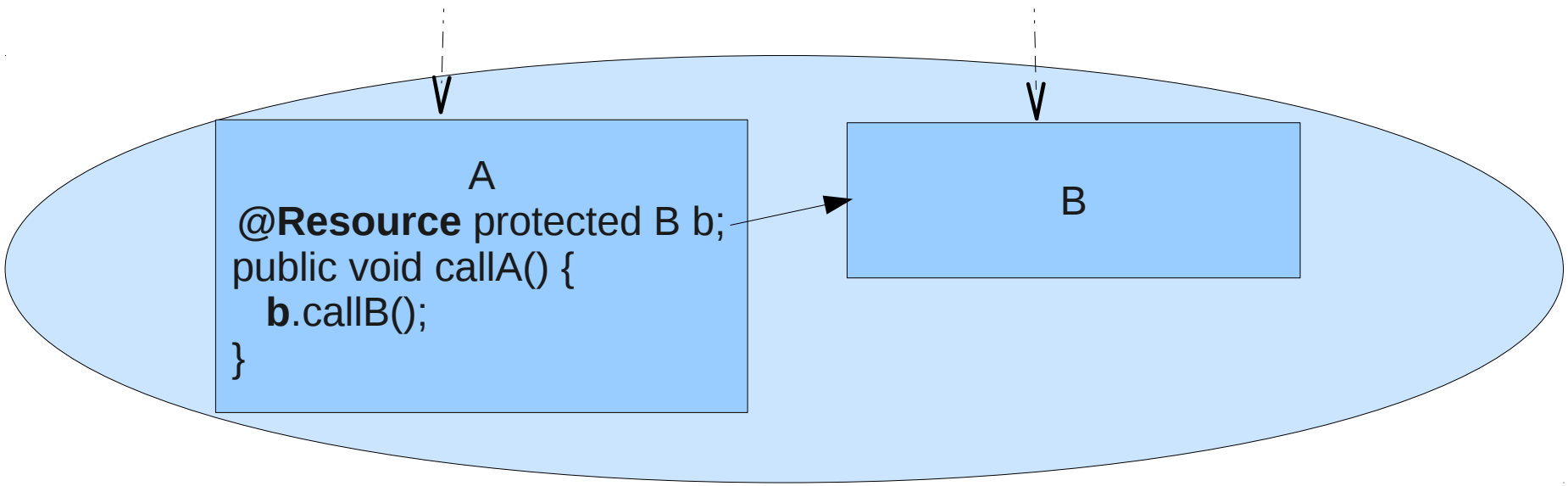
- Problems
  - Singleton contains technical code for initialization
  - Type implementation is hard-coded
  - Object is used by many others, dependency is hidden
  - Lazy init + Untestable … only at run-time

Singleton vs IOC

- Without IOC



- With Container + Dependency Injection



Container = Bootstrap + init all soon

- The container is equivalent to bootstrap code:

```
// step 1 : instanciate all objects from classes + add AOP aspects / Proxy wrappers..
Map<String,Object> ctx = new HashMap<String,Object>();
MyEJB myEJB = new MyEJBImpl();
myEJB = (MyEJB) Proxy.newProxyInstance(cl, new Class[] { MyEJB.class }, new Transacti
MySpringObj myObj = new MySpringObj();
ctx.put("myEJB", myEJB); ctx.put("mySpringObj", myObj);

// step 2: configure + link objects (ok even with cyclic deps)
myEJB.setMyObj(myObj);
myObj.setMyEJB(myEJB);

// step 3: start() all objects with lifecycle callback interface
for (Object elt : ctx.entrySet()) {
    if (elt instanceof Lifecycle) {
        ((Lifecycle) elt).start();
    }
}
```

- Pros
  - No more difficult Egg-and-Chicken problem...
  - NO lazy init problem found at runtime
  - All object can be tested by mocks (see next)

Static Fields (constants) ANTI-Patterns

- Bad …

```
private static String dbUrl = "jdbc:oracle:thin@localhost:8000:DB1";
private static String dbLogin = "admin";
private static String dbPassword = "password";

static {
    Properties props = new Properties();
    try {
        props.load(new FileInputStream("not-better-with-file.properties"));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

- OK use SpringPropertyPlaceholder
  - Externalize values in placeholder “${key}”
  - Values comes from “key=value” properties file
  - Choose which file to inject in PropertyPlaceHolder

PropertyPlaceHolder

- Xml declaration

```
<bean id="propertyConfigurer" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location">
        <value>classpath:db-env.properties</value>
    </property>
</bean>
```

- Use values in Xml beans

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="${dbDriver}"/>
    <property name="url" value="${dbUrl}"/>
</bean>
```

- Use values in Java code

```
@Component
public class OkProperties {

    @Value("${dbUrl}") private String dbUrl;

    @Value("${dbUrl}") private String dbLogin;

    @Value("${dbPassword}") private String dbPassword;
```

explicit create class instance (hard-code class + wrapper)

- BAD : instanciating objects may be tedious
  - contains technical boiler-plate code!

```
private static MyEJB INSTANCE = createInstance();
public static MyEJB getInstance() { return INSTANCE; }
private static MyEJB createInstance() {
    MyEJB myEJB = new MyEJBImpl();
    // wrap object with Transaction proxy (AOP aspect) + logs aspect + security + ...
    ClassLoader cl = EvilSingletonXA.class.getClassLoader();
    myEJB = (MyEJB) Proxy.newProxyInstance(cl, new Class[] { MyEJB.class },
            new TransactionInvocationHandler(myEJB));
    return myEJB;
}
```

- OK: use Spring facilities, like AOP, annotations..

```
<tx:annotation-driven transaction-manager="txManager"/>

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager" />
```

Use Interface instead of Classes … but not too much (no EJB stutter)

- Use interface when appropriate, example:
  use javax.sql.DataSource
  … not com.oracle.jdbc.OracleDatasource

- define a summary interface for contract between client-server objects

- For server-server code (example DAO)
  ... not need for interface
  spring use CGLIG / AopAlliance (with restrictions)

Common Architecture Layers

- 3-Tiers Architecture principles:
- Tiers 1 = Exported Protocol Services
  - Example for Web: (Web tiers = tomcat...)
    - Servlet, Jsp, Rest, WebService (jaxws)...
  - RMI, JMS, Corba, Hessian, t3 (weblogic:), …
- Tier 2 = Service Layer (business code)
  - Contains “EJB” + DAO + helper
  - Access to jdbc from JTA + Orm (eclipselink, hibernate...)
- Tier 3 = DataBase layer

3-Tiers with Spring

- Tiers 1 = expose explicitely connectors in XML
  - Ex 1 webservice:
    ```
    <wss:binding url="/myWebService" service="#myObj" />
    ```
  - Ex 2 JMX:
    ```
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
      <property name="beans">
        <map>
          <entry key="bean:name=testBean1" value-ref="testBean"/>
        </map>
      </property>
    </bean>
    ```
- Tiers 2 = only @Component in java POJO
- Tiers 3 = Hibernate + JTA fully supported by spring...

Spring is modular : write once – run many

- code is NOT dependent of Spring
- code is NOT dependent of anything technical
  - Can choose at deployment time which protocol...

- Strength of Spring : the same code can run
  - in weblogic (production mode)
  - In Eclipse (standalone mode with DB)
  - In Junit integration Test with DB
  - In real Unit-test with Mocks (no DB)

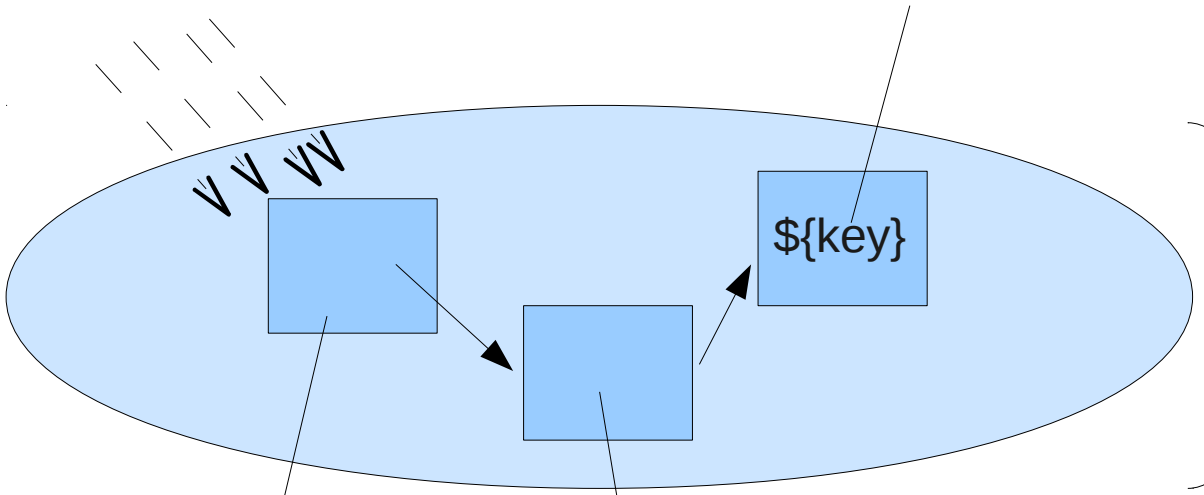
Program Aspect Splits
Java – Xml - Properties

Xml Architecture Setup
(= plombing)
Protocol exported + internals

```
<context:component-scan base-package="fr.an.test" />

<bean id ="myPojo2" class="fr.an.test.MyPOJO" />
```

Externalized Properties file
= environment specific
(prod/int/dev..)

```
# key = value  properties file

dbUrl = jdbc:oracle:thin@localhost:8000
dbLogin = admin
dbPassword = password
```



Stateless
Service Objects
In container

```
addOrder() {
  SendEmail();
  Save();
}
```

```
reportPdf() {
  find();
  exportPdf();
}
```

Functionnal code

Mock Testing

- Example of Mock framework libraries : EasyMock, Mockito
- Goals : test a POJO
  … but replace all its (spring) dependencies by dummy mock objects
- Mock are implemented at runtime from interface + dynamic Proxy
- They record + replay + check method calls

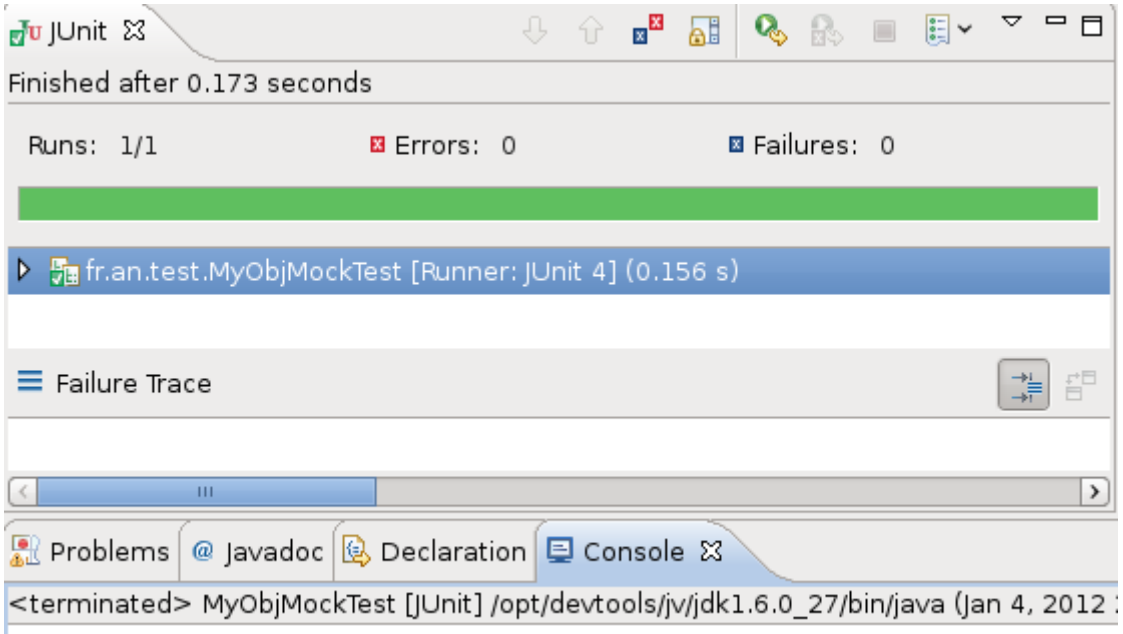
Sample Mock Test

- Annotations:
  - @Mock to instanciate mock proxy on interfaces
  - @InjectMock to inject dependency into object

```
public class MyObjMockTest {

    // @Mock is equivalent to create a mock proxy
    // init from MockitoAnnotations.initMocks(this);
    @Mock private MyEJB myEJB;

    // SUT = System Under Test
    // @InjectMocks is equivalent to autowiring setter "sut.set(...)"
    //  init from MockitoAnnotations.initMocks(this);
    @InjectMocks
    private MySpringObj sut = new MySpringObj();

    @Before public void setup() {
        // equivalent to @RunWith(MockitoJUnitRunner.class) in test class
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void test1() {
```

Expect-Run-Verify Mock Test

```
@Test
public void test1() {
    // step 1: when
    Mockito.when(myEJB.call(1)).thenReturn(2);
    Mockito.when(myEJB.call(2)).thenReturn(4);

    // step 2: run
    int res2 = sut.callMyEJB(1);
    int res4 = sut.callMyEJB(2);

    // step 3: verify
    Assert.assertEquals(2, res2);
    Assert.assertEquals(4, res4);
    Mockito.verify(myEJB).call(1);
    Mockito.verify(myEJB).call(2);
}
```



Real object MyEJBImpl is not called
(no System.out.println())

Questions ?

Alors Tps !

This document :
http://arnaud.nauwynck.chez-alice.fr/devPerso/Pres/Intro-SpringIOC.pdf