Testing Java EE applications

Karel Piwko
JBoss WFK QA

November 2010

Outline

Testing applications

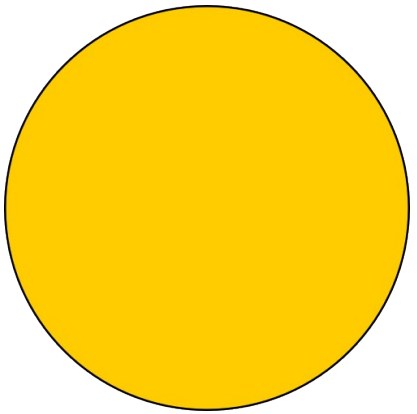
- Why do we test applications?
- How do we test applications?

Testing Java EE applications

- Problems
- Useful tools
- Testing Java EE the JBoss way

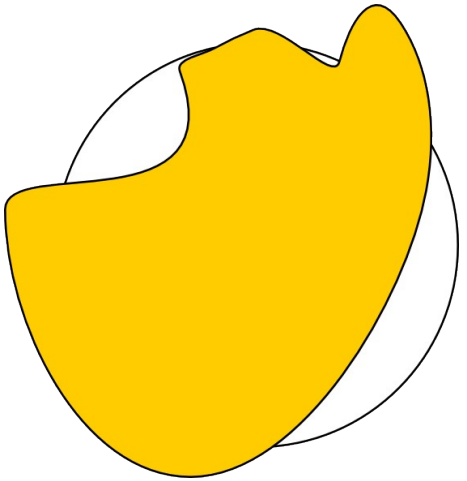
Why do we test applications?

- Developers tend to see their application often as a perfect piece of code



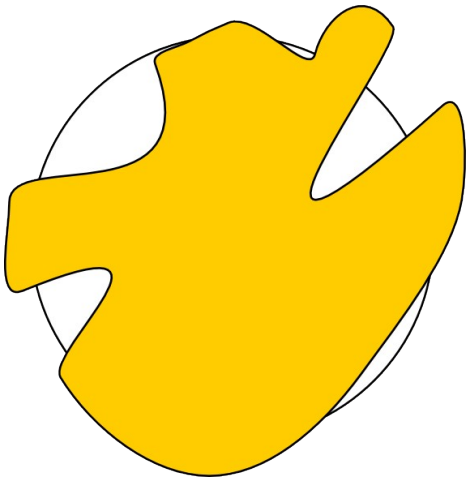
Why do we test applications?

- But often...



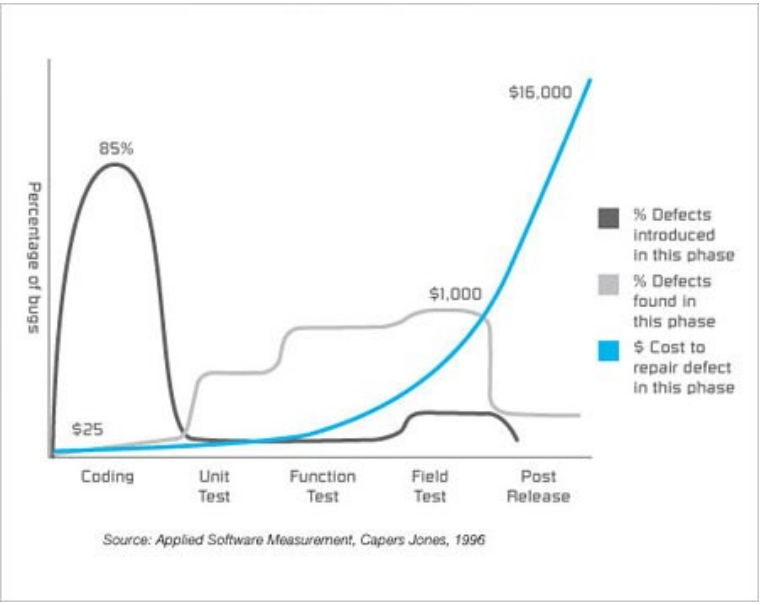
Why do we test applications?

- Last fix was a two-liner...



Why do we test applications?

- Ensure the software contains the least bugs possible
- Verification vs. validation
  - complies with specifications and conditions specified in a development phase
  - accomplishes expected requirements
- Sooner means cheaper



How do we test applications?

- Test approach
  - white box testing
  - black box testing
  - gray box testing

- Test type
  - code analysis
  - unit test
  - integration test
  - functional test
  - system test

White box testing

- Tests internal structure of the application
  - branching, control flow, data flow
- Usually unit level

- Drawback
  - can't test code which is not written

White box testing

```java
public interface Quest
{
   void embark() throws QuestException;
}
```

```java
public class Knight
{
   private Quest quest;

   public Knight(Quest quest)
   {
      this.quest = quest;
   }

   void embarkOnQuest() throws QuestException
   {
      quest.embark();
   }
}
```

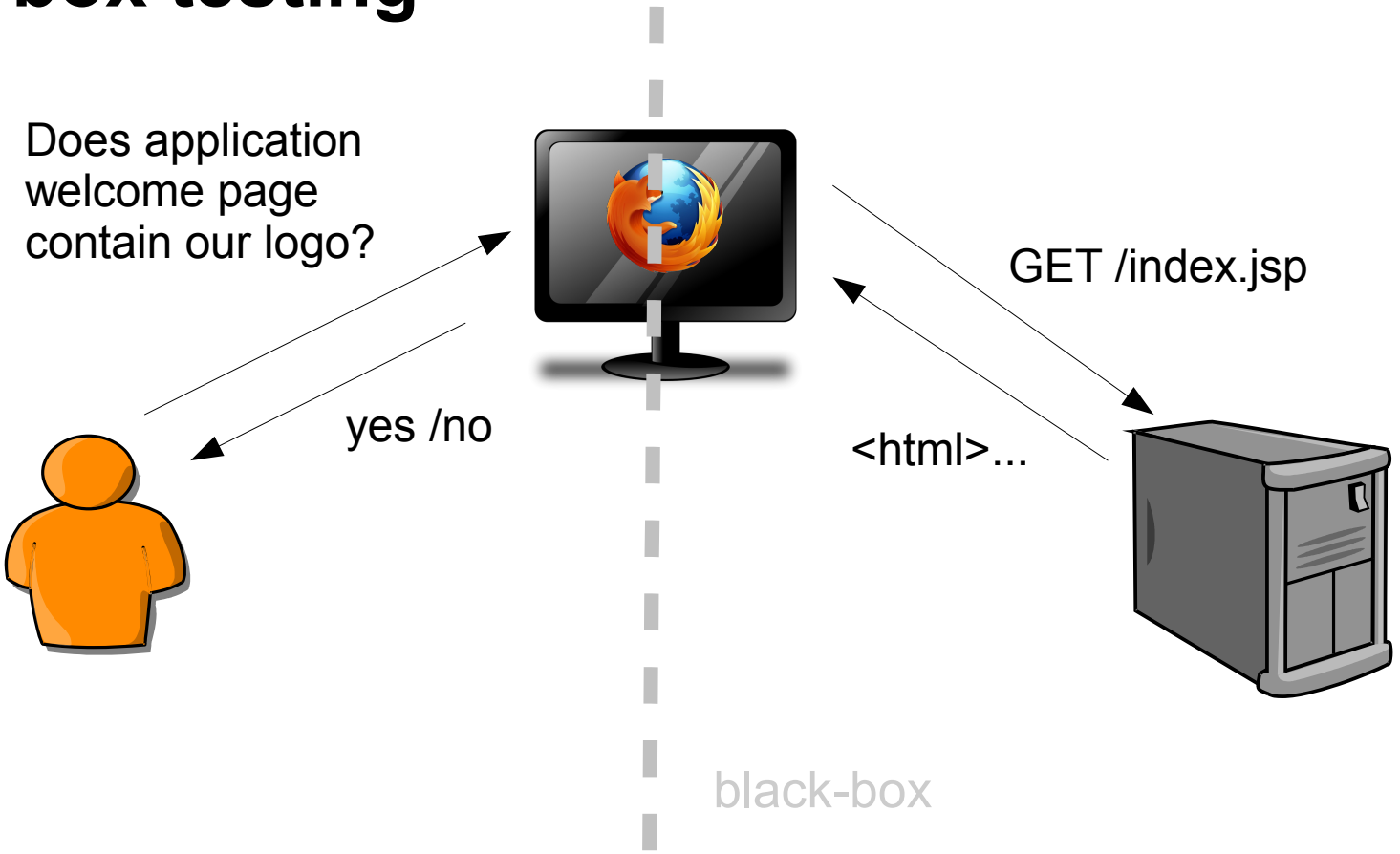
```java
public class KnightTest
{
   @Test
   public void testQuestEmbark() throws QuestException
   {
      Quest mockQuest = Mockito.mock(Quest.class);
      Knight knight = new Knight(mockQuest);
      knight.embarkOnQuest();

      Mockito.verify(mockQuest, Mockito.times(1)).embark();
   }
}
```

Black box testing

- Internal structure of the application not known or not required
- Specification and requirements are used to validate functional behavior
- Usually integration or functional level

- Drawback
  - results can be influenced by state of the black-box component

Black box testing



Code analysis

- Code verification
  - Static analysis
    - type analysis, bug pattern searching
  - Dynamic analysis
    - code coverage
    - debuggers, profilers
  - Formal methods
    - based on mathematical theories
    - full automation, soundness, completeness, termination

Code coverage

- Determine how much of the code is tested
  => use the information to add test cases

- Tool: EMMA, Cobertura
  - branch, live, method, class, package coverage reports

- Unit versus integration tests
  - generally the possible coverage result will decrease with test level
  - test coverage results can be misleading if we sum different levels

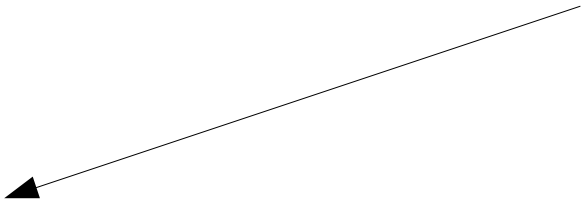
Unit tests

- Tests individual units of source code in isolation
  - enforces code style
  - stubs and mock objects
- Usually created by programmers
  - test driven development possible

- Can run in an IDE
- The granularity of unit matters
- It is difficult to cover all execution paths of the application

Unit test granularity

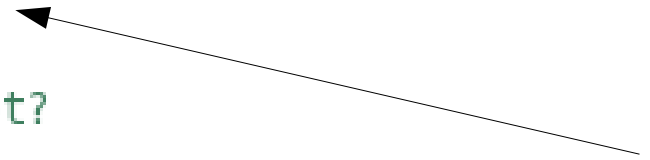
```
public class RescueQuest implements Quest
{
   private String princess;

   public RescueQuest(String princess) {
      this.princess = princess;
   }

   public void embark() throws QuestException
   {
      System.out.println("Princess " + princess  + " was rescued!");
   }
}

public class JediKnight
{
   private Quest quest;

   public JediKnight()
   {
      this.quest = new RescueQuest("Leia");
   }

   void embarkOnQuest() throws QuestException
   {
      quest.embark();
   }
}
```



We coupled two units together!

Unit test granularity

```
public class JediKnightTest
{
    @Test
    public void testQuestEmbark() throws QuestException
    {
        // the class is coupled with RescueQuest
        JediKnight obiWan = new JediKnight();
        obiWan.embarkOnQuest();

        // was Leia rescued or not?
    }
}
```



We cannot easily find out what happened inside!

- Solution
  - Decouple contracts and its implementation (constructor)
  - Provide better interface for Quest

Integration and functional tests

- Tests groups of verified units together
- Complex
- Cannot be easily run in an IDE

- Continuous integration testing
  - run unit and integration tests after each modification
  - version control system (SVN, Git, Hq, ...)
  - automation of the process (Hudson)

  => feature and nightly builds

System tests

- Compliance of the system to its specified requirements
- Smoke tests
  - Verification of the system before performance tests
- Load tests
  - Behavior under load
- Stress tests
  - Behavior under load beyond usual expectations
- Soak tests
  - Behavior with a long period of the time

Testing Java EE applications

- Problems
  - Java EE applications are complex, thus it is difficult to isolate components
    - application server (JBoss AS, GlassFish, WebSphere, ...)
    - communication (JMS, HornetQ, ...)
    - UI (web based - JSF, JSP, RichFaces, ...)
    - database layer (JPA, Hibernate, ...)
    - ...
  - Testing is highly time consuming, not enjoyable and hard to be done properly

=> Leads to even more stubbing, mocking and innovative approaches

What do we need to test Java EE applications?

- Build tool
  - Maven, Ant, Ivy, Gradle
- Test framework
  - TestNG, JUnit
- Mock framework
  - Mockito, jMock, JMockit, EasyMock
- UI testing frameworks
  - Selenium, WebDriver, JSFUnit, Ajocado, HTMLUnit

... and lot of others

Testing Java EE the JBoss way

- Goal
  - make active mocks easier to use
  - configure applications to use test data sources
  - deal with classpath isolation in container
  - create/deploy application archive
  - handle “too many frameworks involved” problem

=> give developers tools to make Java EE testing fun again

ShrinkWrap

- Simple API to assemble archives like JARs, WARs and EARs
  - allows building integration bits directly in the code
  - keeps the isolation in test execution
- Used by Arquillian internally

http://community.jboss.org/wiki/Shrinkwrap

Skip the Build!

ShrinkWrap

- How to build WAR in application?

```
ShrinkWrap.create(WebArchive.class, "weld-login.war")
.addClasses(Credentials.class, LoggedIn.class, Login.class, User.class, Users.class)
.addWebResource(new File("src/main/webapp/WEB-INF/beans.xml"), "beans.xml")
.addWebResource(new File("src/main/webapp/WEB-INF/faces-config.xml"), "faces-config.xml")
.addWebResource(new File("src/main/resources/import.sql"), ArchivePaths.create("classes/import.sql"))
.addResource(new File("src/main/webapp/index.html"), ArchivePaths.create("index.html"))
.addResource(new File("src/main/webapp/home.xhtml"), ArchivePaths.create("home.xhtml"))
.addResource(new File("src/main/webapp/template.xhtml"), ArchivePaths.create("template.xhtml"))
.addResource(new File("src/main/webapp/users.xhtml"), ArchivePaths.create("users.xhtml"))
.addManifestResource(new File("src/main/resources/META-INF/persistence.xml"))
.setWebXML(new File("src/main/webapp/WEB-INF/web.xml"));
```

- Many other ways how to include files in an Archive
  - by package, class name, file, stream, zip

ShrinkWrap extensions

- ShrinkWrap dependencies
  - Resolves dependencies from Maven repositories
  - Can reuse information in POM file to reduce verbosity

```
WebArchive war = ShrinkWrap.create(WebArchive.class, "test.war")
      .addLibraries(Dependencies.artifacts("org.apache.maven.plugins:maven-help-plugin:2.1.1",
                                           "org.apache.maven.plugins:maven-patch-plugin:1.1.1")
                                .resolve());
```


Arquillian

- Brings you the way to write integration tests in a same way as you do for unit tests
  - manages lifecycle of a container
  - bundles and deploys test archive
  - enriches test classes
  - captures test results and
- Does not bind a build to the test, configuration is kept externally

Arquillian makes integration testing a breeze!

Arquillian

- Can be used within multiple build tools, containers and test frameworks, specialized on EE testing

- Expendable via SPI interface

Arquillian and @Inject (In-container testing)

```
@RunWith(Arquillian.class)
public class InjectionTestCase
{
   @Deployment
   public static JavaArchive createDeployment() {
      return ShrinkWrap.create(JavaArchive.class, "test.jar")
               .addClasses(GreetingManager.class, GreetingManagerBean.class)
               .addManifestResource(new ByteArrayAsset("<beans/>".getBytes()),
                     ArchivePaths.create("beans.xml"));
   }

   @Inject GreetingManager greetingManager;

   @Test
   public void shouldBeAbleToInjectCDI() throws Exception {

      String userName = "MUNI";
      Assert.assertEquals("Hello " + userName, greetingManager.greet(userName));

   }
}
```


Arquillian and @EJB (In-container testing)

```
@RunWith(Arquillian.class)
public class InjectionTestCase
{
   @Deployment
   public static JavaArchive createDeployment() {
      return ShrinkWrap.create(JavaArchive.class, "test.jar")
                .addClasses(GreetingManager.class, GreetingManagerBean.class);
   }

   @EJB
   private GreetingManager greetingManager;

   @Test
   public void shouldBeAbleToInjectEJB() throws Exception {
      String userName = "MUNI";
      Assert.assertEquals("Hello " + userName, greetingManager.greet(userName));
   }
}
```

Arquillian As-Client Testing (Out of container)

```
@Run(AS_CLIENT)
public class CommonLoginTest extends Arquillian
{
   @Selenium DefaultSelenium selenium;

   @Test
   public void loginTest()
   {
      selenium.open("http://localhost:8080/weld-login/home.jsf");

      assertFalse(selenium.isElementPresent(LOGGED_IN), "User should not be logged in!");
      selenium.type(USERNAME_FIELD, "demo");
      selenium.type(PASSWORD_FIELD, "demo");
      selenium.click(LOGIN_BUTTON);
      selenium.waitForPageToLoad("60000");
      assertTrue(selenium.isElementPresent(LOGGED_IN), "User should be logged in!");
   }
   // ...
```


kpiwko@redhat.com | www.redhat.com