# Framework Test Strategies

Dr. Wolfgang Winter

## *Abstract*

Frameworks and applications in system integration mode have higher requirements on module and integration tests than hosted applications. Frameworks must work in various runtime environments, comprising operating systems, application servers, web containers and databases. This article presents the setup of a test environment that allows executing automated module and integration tests in multiple runtime environments. Execution of tests in these environments fits seamlessly into the Maven build process.

## *Test Scope*

Everybody knows that testing is essential for the success of a software and for keeping a good relation with the customer who ordered it. Everybody knows however, that testing is one of the first tasks that are skipped when time and resources are running out and deadlines are coming nearer. Maybe sometimes this may be acceptable when you have a really good relation to your customer but generally testing is essential for the success of a product. I see a graduation of importance and quality of testing dependent on the nature of the application:

1. Hosted applications.
   Such kind of applications could be used by a limited number of clients as for example an administration application which is used by office staff. Hosted applications could also be used over the internet or mobile by a large and uncontrollable number of users. In this case the importance and requirements of testing are even higher regarding security, scalability and multi user performance. In hosted applications the processes and responsibilities are normally well defined and include a test phase and acceptance tests. Automatic unit tests are essential, especially when the software changes a lot between releases. No doubt, tests in any form are important for such kind of projects.
2. Frameworks and applications in system integration mode
   In hosted applications the environment is normally well defined. It is clear on which platform the application is run, if the application is running in a container like Spring or EJB and which application server and database are used. All this is not defined for a framework project or an integrated application (see Fig. 1). The developers don’t know in which kind of application the framework will be integrated and in what environment the application will be run. They also may not know anything about the number of users accessing the application

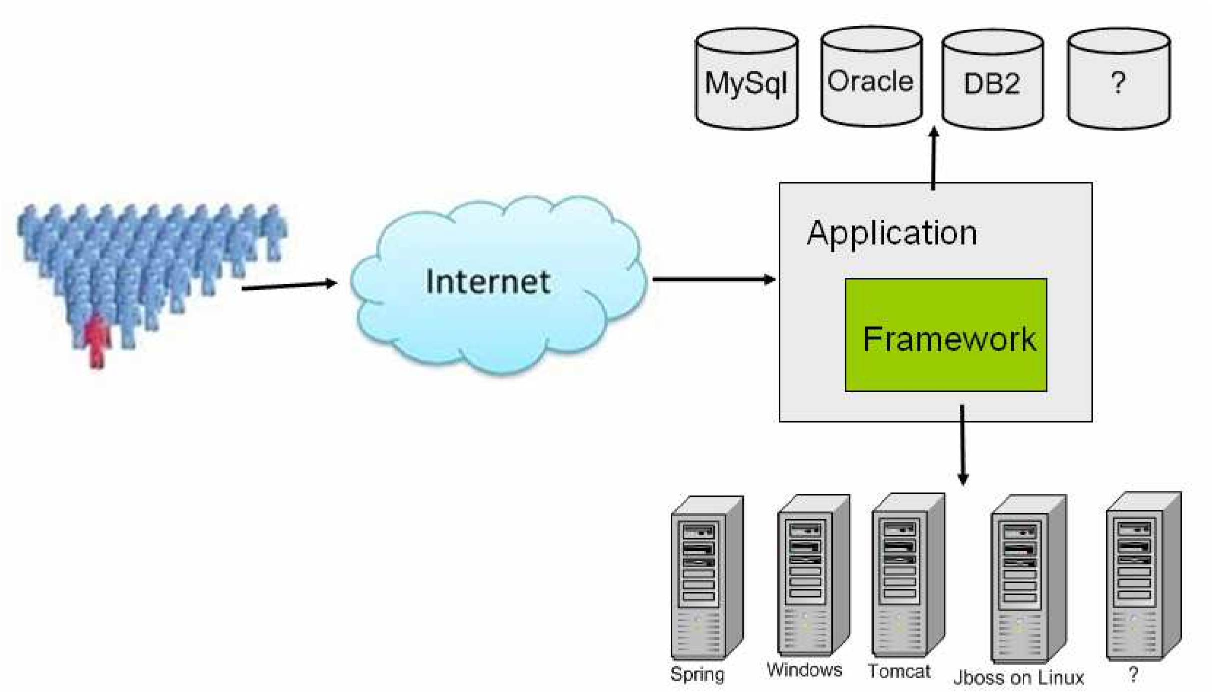


**Fig. 1: Integration of a framework in different environments**

Thus frameworks have the highest requirements for testing. All the arguments of hosted applications apply as well but the problem for frameworks is that nothing is known about possible usages and environments. The only chance a framework developer has is to restrict its application to certain conditions, possibly the conditions that have been thoroughly tested. A restriction may exist for example for the Java version or dependencies to other third party libraries. Frameworks may set restrictions on the database or application server types and versions. On the other hand, a framework is built to be used in different environments and circumstances; it lives from a broad distribution on many platforms. Therefore it is desirable to test a framework in as many environments as possible. In the following it will be shown how the testing on multiple runtime environments has been automated for the Cibet framework.

## *Cibet Framework*

The Cibet framework (http://www.logitags.com/cibet) allows setting controls on business processes like method invocations, persistence operations on domain objects and http requests. Controlling actions could be applying dual control principles like four-eyes control, audit-proof archiving or access controls. The functionality is achieved by integrating an open loop controller structure into an application in a nearly non-intrusive manner.

The main principle is shown in Fig. 2. An event is observed by an appropriate sensor. Cibet provides sensors for EJB service invocations, JPA persistence, method invocations and http requests. The controller checks this signal, compares it with the configuration and actual context and decides which actions to take and which actuator to apply. The actions of the actuators are then an input to the system.

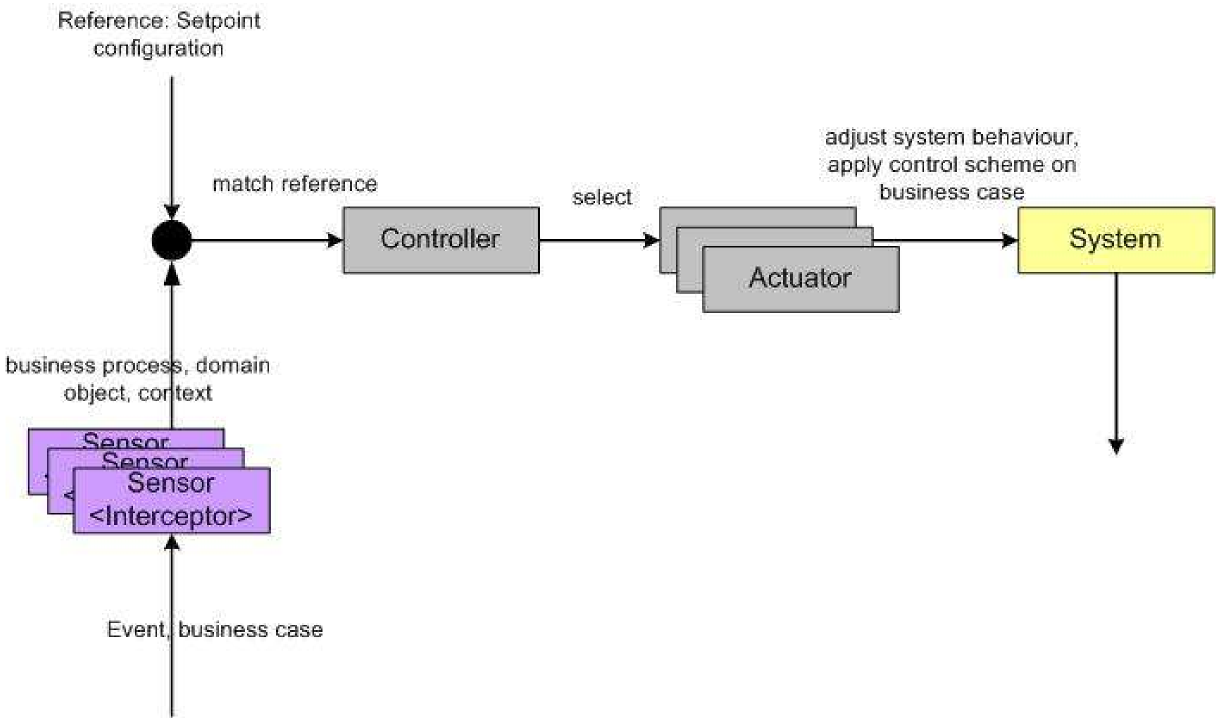

Reference: Setpoint configuration
match reference
Controller
select
adjust system behaviour, apply control scheme on business case
Actuator
System
business process, domain object, context
Sensor
<Interceptor>
Event, business case

**Fig. 2: Open loop controller structure of Cibet**

Built-In actuators are a four-eyes and a six-eyes actuator which postpone the event and execute it only when a second or third user has checked and approved it. An archiving actuator stores the event so that it could be restored or redone at a later time. The integration with Spring Security implemented in an actuator allows controlling the execution of the event according to user permissions.

## *Test Requirements of Cibet*

The development environment for Cibet is based on Maven. Therefore a test environment is set up that fits well into the Maven build process and relies on the Maven Surefire plugin and JUnit as basic test framework. The Maven Failsafe plugin that separates execution of unit tests from integration tests in the Maven build lifecycle is not used but could be an option. For the Cibet framework we always want to run all tests automatically to ensure that the code is always working as expected. The Failsafe plugin allows also making use of the Maven pre- and post- integration test phases to set up and tear down the test environment. In the Cibet tests the same is achieved with the JUnit annotations @BeforeClass and @AfterClass. Therefore we keep it as simple as possible and do not add another level of configuration.

Now what do we want to test in the Cibet framework?

Basic tests are naturally unit tests which verify the functionality of isolated methods. Some functionality of Cibet like four-eyes control requires database persistence actions, thus integration tests must involve a database. Cibet allows not only controlling method invocations and entity persistence but also controlling of http requests. Therefore it is necessary to make also integration tests against a web container. Cibet controls method invocations on pojo objects but also on EJBs. In order to test the Java EE functionality we need integration tests with an EJB container.

Cibet has no GUI, so we don’t have to think about automation of user interface tests. We also have no requirements to test other enterprise resources like JMS or integration with other tools or third-party applications.

## *Unit Tests*

Unit tests verify if individual units of source code or modules, normally a single class or a method within a class, execute as expected. Ideally, the tested class should be independent on other classes, resources, database or container as only the internal working of the class is tested, not the interfaces. Such dependencies are often simulated by stubs and mock objects. There are several mock object frameworks for this purpose on the market like jMock (http://www.jmock.org/), EasyMock (http://easymock.org/) or Mockito (http://code.google.com/p/mockito/) which all provide nearly the same functionalities. The choice may be a matter of personal gusto. For the Cibet framework we choose Mockito for it’s easy to understand and intuitive API. It can be applied for example like shown in Listing 1 in a JUnit test class:

**Listing 1**
```
@RunWith(MockitoJUnitRunner.class)
public class CibetEntityManagerTest {

   @Mock
   protected EntityManager em;

   private EntityManager cib = new CibetEntityManager();

   @Test
   public void find() {
      TEntity te = new TEntity("Hansi", 99, "owned by x");
      te.setId(5);
      Mockito.when(em.find(TEntity.class, (Long) 5l)).thenReturn(te);

      TEntity result = cib.find(TEntity.class, 5);
      Assert.assertNotNull(result);
      Assert.assertEquals(99, result.getCounter());
   }
}
```

Mockito is very easy to apply:
- Allow usage of Mockito annotations by running the test with MockitoJUnitRunner
- Create a mock object with @Mock annotation
- Define expected behaviour with the static methods of Mockito class

However, there are limitations for the usage and benefit of mocking and a mock framework. Usually static and private methods cannot be mocked. Okay, private methods should be invisible for testing and only the public API is normally tested but imagine the case where a

public method calls a private method which in turn performs a database access. It will be very hard to test the public method without a real database. We also don’t want to design production code only to fit it into the testing environment.
Sometimes it is also so complex to set up and stuff a whole garden of mock objects for a single unit test that it is much easier to skip such a unit test and go directly for the integration test.

## *Database Integration Tests*

One requirement for a good integration test is that the database is set up before the test with required test data and cleaned after the test from all artifacts that have been inserted. Of cause there is the DbUnit (www.dbunit.org/) framework which supports these tasks but for the Cibet tests it would be too much overhead because we have only very few test data to insert beforehand and the cleaning is easy in a Java SE environment: We put the whole test into a transaction and rollback after the test. Our abstract test class looks like in Listing 2:

**Listing 2**

```
public class AbstractTestIntegration {

  @Before
   public void doBefore() {
      entityManager.getTransaction().begin();
   }

   @After
   public void doAfter() throws Exception {
      if (entityManager.getTransaction().isActive()) {
         if ("true".equals(System.getProperty("test.commit"))) {
            entityManager.getTransaction().commit();
         } else {
            entityManager.getTransaction().rollback();
         }
      }
   }

}
```

The problem with automatic rollback after the tests is that database modifications can not be checked visually. This is especially important during the implementation of the test, when it is tested itself and when tests don’t work anymore to analyze the problem. Therefore, the transaction could be committed simply by executing the test with a system property ‘test.commit’ set to true.

Another requirement is to automate the tests against different databases. The database connections that should be tested are configured in an XML file (see Listing 3):

**Listing 3**

```
<?xml version="1.0" encoding="UTF-8"?>
<testEnvironment xmlns="http://www.logitags.com/testEnvironment"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.logitags.com/testEnvironment
        testEnvironment.xsd">

        <database>
                <name>RemoteOracle</name>
                <url>jdbc:oracle:thin:@192.168.1.3:1521:xe</url>
                <driver>oracle.jdbc.OracleDriver</driver>
                <user>cibet</user>
                <password>cibet</password>
                <dialect>org.hibernate.dialect.Oracle10gDialect</dialect>
                <active>true</active>
        </database>
        …
```

The automation of running the same tests against the defined databases is achieved applying the parameter controlling feature that is offered by the JUnit framework. In Listing 4 it is shown how this is done in an abstract test class:

**Listing 4**

```
@RunWith(Parameterized.class)
public abstract  class AbstractTestIntegration {

  @Parameters
  public static Collection<String[]> configs() throws Exception {
    Collection<String[]> col = new ArrayList<String[]>();
    TestEnvironment env = loadTestEnvironment();
    for (Database db : env.getDatabase()) {
      if (!db.isActive()) continue;
      Map<String, String> props = new HashMap<String, String>();
      props.put("hibernate.dialect", db.getDialect());
        …
        EntityManagerFactory emf = Persistence.createEntityManagerFactory(
           "cibetTestDB", props);
        emanFacMap.put(db.getName(), emf);
        col.add(new String[] { db.getName() });
    }
    return col;
  }

  protected AbstractTestIntegration(String dbName) {
    databaseName = dbName;
  }
```

By applying the test runner Parameterized the method annotated with @Parameters is executed before the test. In the above case the method loads the XML configuration file and creates and caches an EntityManagerFactory for each configured and active database connection. The method must return a collection of arrays. The collection contains one element for each database connection and the array contains the parameters which are passed to the test class constructor. We define only one parameter which is the configured name of the database. The Parameterized test runner executes the test now once for each element in the collection and passes the database name to the constructor which uses it to find the corresponding EntityManagerFactory in the map.

## *EJB and Servlet Container Integration Tests*

For testing Java EE functionalities and servlet requests we want to run the tests against different containers. As we want a full automated process we use embedded containers. There have been already a handful of embedded containers on the market and since EJB 3.1 the embedded server API is part of the specification. First thing to do in order to use embedded containers is to write wrapper classes which are responsible for starting, stopping and initialising the server and handling jndi names and lookup of EJBs.
The wrappers of all tested containers are then configured in the same XML file as the databases (see Listing 5)

**Listing 5**

```
<container>
   <name>EmbeddedTomcat7</name>
   <class>com.logitags.cibet.helper.env.EmbeddedTomcat7</class>
   <ejb>false</ejb>
   <servlet>true</servlet>
   <active>true</active>
</container>
```

Now we use again the Paramerized test runner to set up the test classes (see Listing 6)

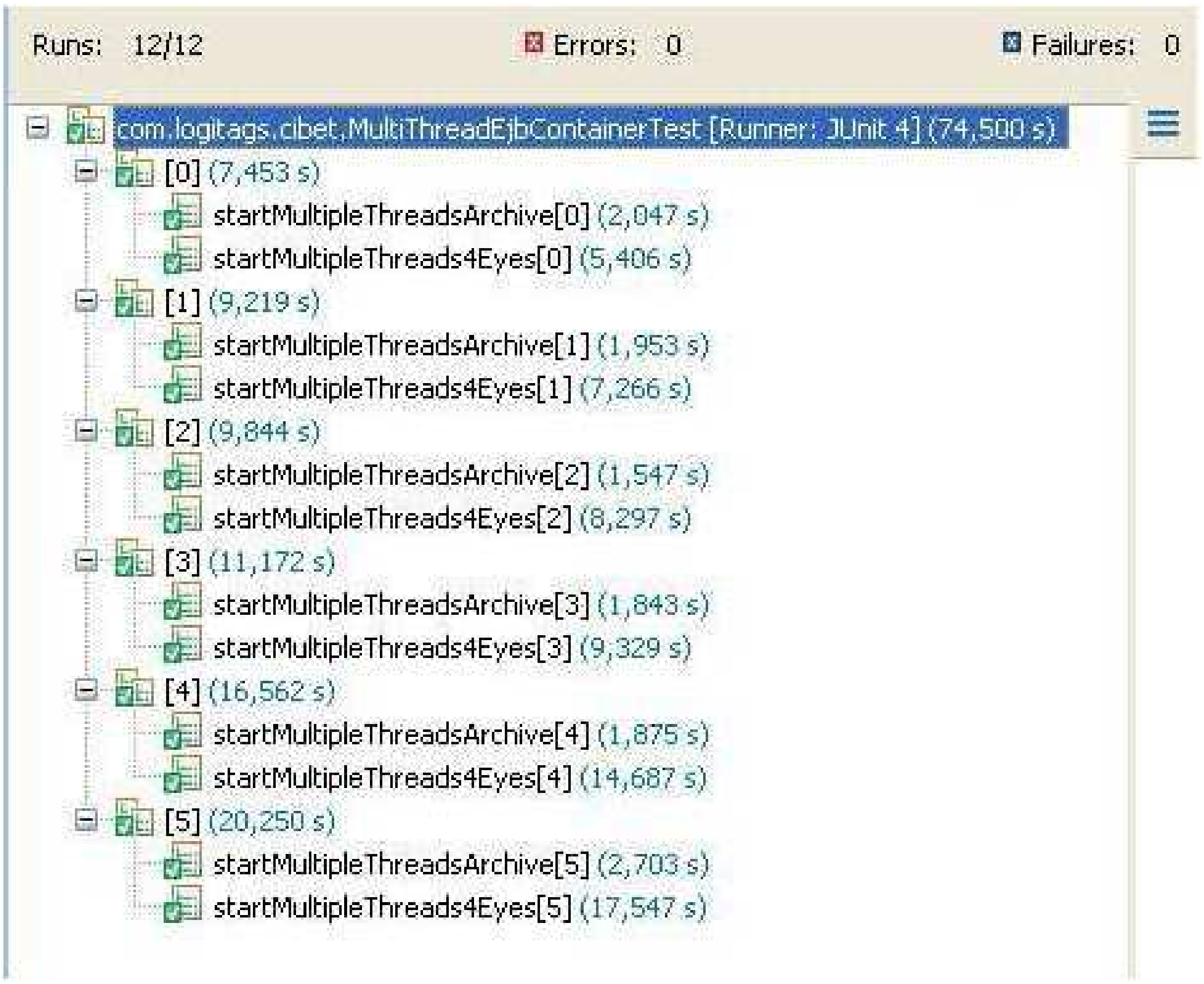
You will note that the array in the returned collection is an object array and contains two elements this time. As we want to test all possible combinations of containers and databases the constructor takes the container wrapper and the database configuration class as parameters.

**Listing 6**

```
@RunWith(Parameterized.class)
public abstract class AbstractTestEjbContainer {

   @Parameters
   public static Collection<Object[]> configs() throws Exception {
      Collection<Object[]> col = new ArrayList<Object[]>();
      TestEnvironment env = loadTestEnvironment();
      Database defaultDB = findDefaultDatabase(env);
      for (Container cont : env.getContainer()) {
         if (!cont.isActive() || !cont.isEjb()) continue;
         Class<EjbContainer> clazz = (Class<EjbContainer>)
             Class.forName(cont.getClazz());
         EjbContainer container = clazz.newInstance();
         container.initDatabaseConnections(env.getDatabase(), defaultDB);
         container.start();

         for (Database db : env.getDatabase()) {
            if (!db.isActive()) continue;
            Object[] test = new Object[] { container, db };
            col.add(test);
         }
      }

      return col;
   }

   protected AbstractTestEjbContainer(EjbContainer embc, Database db)
         throws Exception {
      container = embc;
      database = db;
   }
```

The Cibet  tests are using this strategy in the following environment:
Databases: Derby 10.4, Oracle 10g, MySql 5.5
Containers: JBoss-embedded beta3, Jetty 6.1, OpenEJB 3.1, Tomcat 7

If a test class is executed as JUnit test the output in Eclipse looks now like in Fig. 3. The same test is run six times with various combinations of databases and containers.



```
Runs:   12/12                 Errors:   0                 Failures:   0

com.logitags.cibet.MultiThreadEjbContainerTest [Runner: JUnit 4] (74,500 s)
  [0] (7,453 s)
    startMultipleThreadsArchive[0] (2,047 s)
    startMultipleThreads4Eyes[0] (5,406 s)
  [1] (9,219 s)
    startMultipleThreadsArchive[1] (1,953 s)
    startMultipleThreads4Eyes[1] (7,266 s)
  [2] (9,844 s)
    startMultipleThreadsArchive[2] (1,547 s)
    startMultipleThreads4Eyes[2] (8,297 s)
  [3] (11,172 s)
    startMultipleThreadsArchive[3] (1,843 s)
    startMultipleThreads4Eyes[3] (9,329 s)
  [4] (16,562 s)
    startMultipleThreadsArchive[4] (1,875 s)
    startMultipleThreads4Eyes[4] (14,687 s)
  [5] (20,250 s)
    startMultipleThreadsArchive[5] (2,703 s)
    startMultipleThreads4Eyes[5] (17,547 s)
```

**Fig. 3: Executing a JUnit test in multiple environment**

## *Conclusion*

Though it is possible to run automated tests in multiple environments there are limitations of this approach. Especially using embedded containers is problematic. The container libraries do not contain only their own classes but often also classes of other third party libraries and the API of the Java EE specification are included in the jars. As all containers are started in the same virtual machine very quickly class path problems of doubled classes in different versions arise. For example we did not succeed in running the tests also against Glassfish. Testing different databases with this approach however bears no problems as only the jdbc drivers must be in the class path

An alternative for running integration tests in containers is to use Arquillian (http://www.jboss.org/arquillian). The JBoss community is developing a very interesting framework for testing in a container. In theory it should be very easy to execute tests with Arquillian in various containers according to the documentation. A whole bunch of containers is supported, JBoss, OpenEjb, Tomcat, Glassfish, both in stand alone and embedded modes. Arquillian is also well integrated with the Maven build process and test frameworks like JUnit

and TestNG. However, the latest version of the project is still an alpha release and unfortunately the status makes it not very usable. After two days struggling I gave up to build an Arquillian environment that fulfils the requirements described above. This was mainly due to the miserable state of the JBoss Maven repository (inconsistencies, missing dependencies, missing versions, missing pom.xml) but also to functionalities not working as described in the documentation.
Nevertheless, Arquillian is a very interesting project and worth to observe. I will definitely come back to it once it is in a more settled status.