
# Module 18: Spring Testing

CS544: Enterprise Architecture

Spring Testing:

# BASICS OF TESTING

# Testing software

- Unit test: does one single object work?
- Integration test: do multiple object work together?
- Acceptance test: Does the application work as the client wants it to work?

# Example of unit testing

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
 }
```

# Example of unit testing

```java
package test;
import static org.junit.Assert.*;
import org.junit.*
import count.Counter;

public class CounterTest {
    private Counter counter;

     @Before
     public void setUp() throws Exception {
       counter = new Counter();
     }
     @Test
     public void testIncrement(){
        assertEquals("Counter.increment does not work correctly",1,counter.increment());
        assertEquals("Counter.increment does not work correctly",2,counter.increment());
     }
     @Test
     public void testDecrement(){
        assertEquals("Counter.decrement does not work correctly",-1,counter.decrement());
        assertEquals("Counter.decrement does not work correctly",-2,counter.decrement());
     }
}
```

Initialization

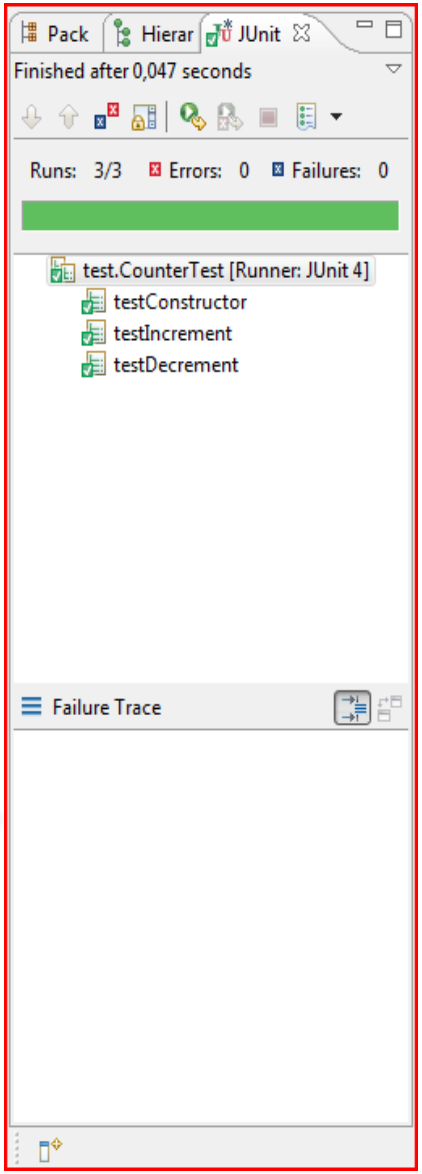
Test method

Test method

```java
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
       return ++counterValue;
    }
    public int decrement(){
       return --counterValue;
    }
    public int getCounterValue() {
       return counterValue;
    }
 }
```

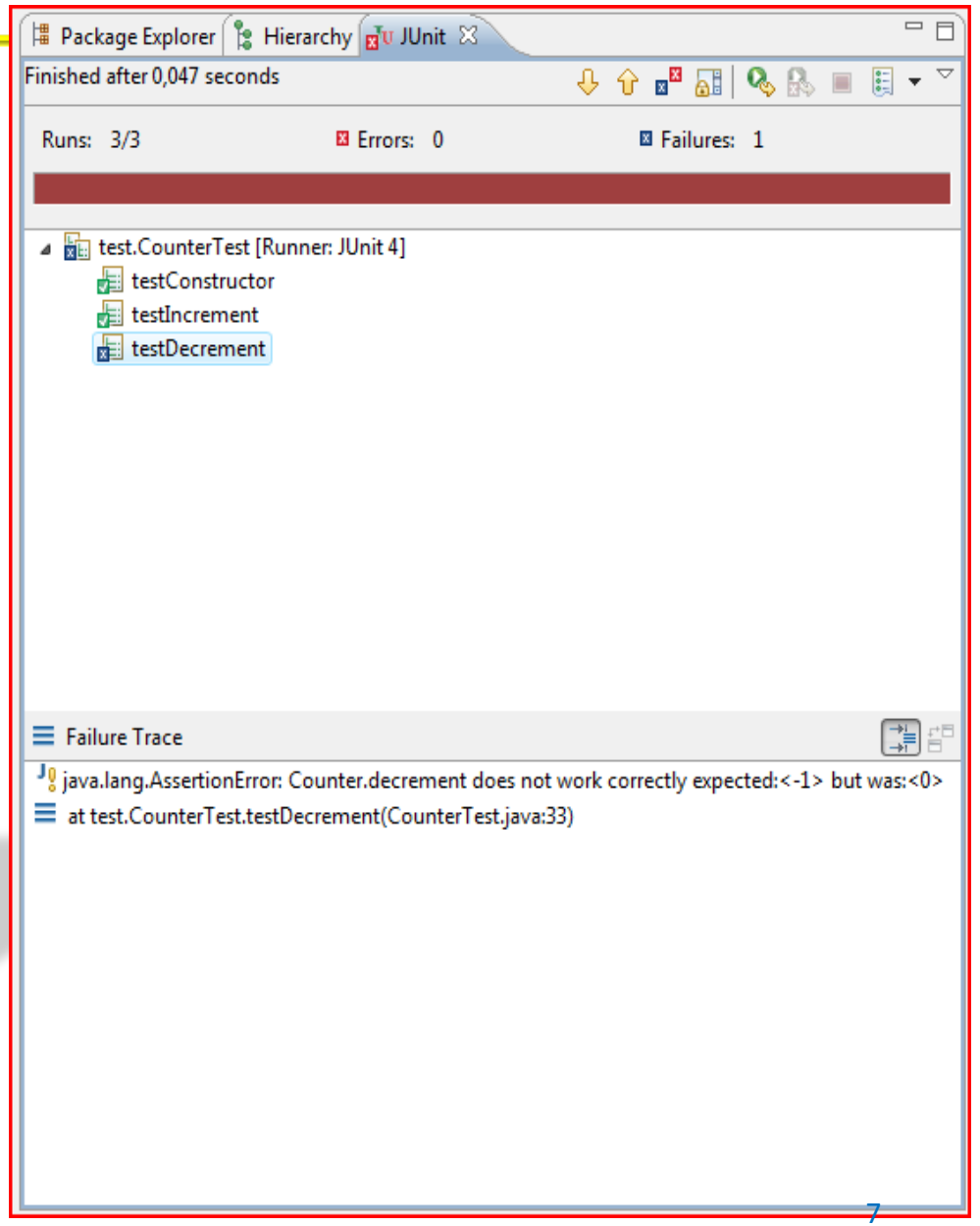
# Running the test

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
 }
```



# Running the test

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
}
```



# What is unit testing?

- A unit test is a test that tests one single class.
  - A test case tests one single method
  - A test class tests one single class
  - A test suite is a collection of test classes
- Unit tests make use of a testing framework

- A unit test
  1. Create an object
  2. Call a method
  3. Check if the result is correct

# Characteristics of unit testing

- Unit tests are written by developers
  - A test framework helps writing and executing tests
- Unit tests run automatically
  - Unit tests verify themselves
- Unit tests can be grouped into a test suite
  - A test suite can be executed in one single action
- Unit tests run quickly
  - Unit tests are run very often
- Testing should be easy

# assertX methods

- static void assertTrue(boolean ***test***)
- static void assertTrue(String **message**, boolean ***test***)

- static void assertFalse(boolean ***test***)
- static void assertFalse(String **message**, boolean ***test***)

- assertEquals(Object ***expected***, Object ***actual***)
- assertEquals(String **message**, ***expected***, ***actual***)

- assertSame (Object ***expected***, Object ***actual***)
- assertSame(String **message**, Object ***expected***, Object ***actual***)

- assertNotSame(Object ***expected***, Object ***actual***)
- assertNotSame(String **message**, Object ***expected***, Object ***actual***)

- assertNull(Object ***object***)
- assertNull(String **message**, Object ***object***)

- assertNotNull(Object ***object***)
- assertNotNull(String **message**, Object ***object***)

- fail()
- fail(String **message**)

# @Before and @After

```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import count.Counter;

public class CounterTest {
    private Counter counter;

    @Before
    public void setUp() throws Exception {
        counter = new Counter();
    }

    @After
    public void tearDown() throws Exception {
        counter=null;
    }
    @Test
    public void testConstructor(){
        assertEquals("Counter constructor does not set counter to 0",0,counter.getCounterValue());
     }
}
```

This method is called before every testmethod

This method is called after every testmethod

# @BeforeClass and @AfterClass

```
package test;
import static org.junit.Assert.*;
import org.junit.*;
import count.Counter;

public class CounterTest {
    private static Counter counter;

    @BeforeClass
    public static void setUpOnce() throws Exception {
        counter = new Counter();
    }

    @AfterClass
    public static void tearDownOnce() throws Exception {
        counter=null;
    }

    @Test
    public void testConstructor(){
        assertEquals("Counter constructor does not set counter to 0",0,counter.getCounterValue());
     }
}
```

This method is called once, before the testmethods are called

This method is called once, after the testmethods are called

# Advantages of unit testing

- Get more confidence in your code
- Makes it possible to change (refactor) code
- Tests are good documentation
- Find bugs fast
- Testing forces you to think twice
- Faster development
- Collective ownership of the code
- Testability and good architecture go hand in hand

Spring Testing:

# TESTING SPRING APPLICATIONS

# Spring makes unit testing easier

- Spring beans can be unit tested without Spring
  - Spring beans are clean POJOs not tied to the Spring framework

# Spring and integration testing

- Test the Spring configuration
- Test DAOs (configured by Spring)
  - OR mapping
  - Queries: HQL, Criteria, etc
  - Database performance
  - Transactions
- Test enterprise service objects (configured by Spring)
  - EmailSender
  - JMSSender
  - …

# The Service class to test

```
public class ShoppingServiceImpl implements ShoppingService {
  private WarehouseService warehouseService;
  private ShoppingDAO shoppingDAO;

  public ShoppingServiceImpl(WarehouseService warehouseService, ShoppingDAO shoppingDAO) {
    this.warehouseService = warehouseService;
    this.shoppingDAO = shoppingDAO;
  }
  ...
}
```

```
<bean id="shoppingService" class="shopping.ShoppingServiceImpl">
    <constructor-arg index="0" ref="warehouseService" />
    <constructor-arg index="1" ref="shoppingDAO" />
</bean>
<bean id="warehouseService" class="warehouse.WarehouseServiceImpl"/>
<bean id="shoppingDAO" class="dao.ShoppingDAOImpl"/>
```

# Test the Spring configuration

Create an ApplicationContext and get the shoppingService from the ApplicationContext

```
public class ShoppingServiceTest {
static ShoppingService shoppingService;

  @BeforeClass
  public static void setUpOnce() throws Exception {
   ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
   shoppingService = context.getBean("shoppingService", ShoppingService.class);
  }

  @Test
  public void testInjectionOnNull(){
    assertNotNull("injected wharehouse service is NULL",shoppingService.getWarehouseService());
    assertNotNull("injected shoppingDAO is null",shoppingService.getShoppingDAO());
  }

  @Test
  public void testInjectionOnType(){
    if (! (shoppingService.getWarehouseService() instanceof warehouse.WarehouseService) )
      fail("injected wharehouse service is not of type WarehouseService");
    if (! (shoppingService.getShoppingDAO() instanceof dao.ShoppingDAO) )
      fail("injected shoppingDAO  is not of type ShoppingDAO");
    }
}
```

# Autowiring by type using field injection

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/springconfig.xml"})
public class ShoppingServiceTest {
  @Autowired
  private ShoppingService shoppingService;

  @Test
  public void testInjectionOnNull(){
    assertNotNull("injected wharehouse service is NULL",shoppingService.getWarehouseService());
    assertNotNull("injected shoppingDAO is null",shoppingService.getShoppingDAO());
  }

  @Test
  public void testInjectionOnType(){
    if (! (shoppingService.getWarehouseService() instanceof warehouse.WarehouseService) )
      fail("injected wharehouse service is not of type WarehouseService");
    if (! (shoppingService.getShoppingDAO() instanceof dao.ShoppingDAO) )
      fail("injected shoppingDAO  is not of type ShoppingDAO");
  }
}
```

The Spring configuration file

The ShoppingService is autowired by type using field injection

Spring Testing:

# TESTING DAO CLASSES

# Testing of DAO classes

- What do you want to test?
  - OR mapping
  - Queries: HQL, Criteria, etc
  - Database performance
  - Transactions
- DAO testing:
  - Bring the database in a well-known state
  - Execute the database action
  - Check if the database action is done correctly
  - Cleanup the database
    - Drop the database
    - Rollback the previous action

# DOA testing with Spring

- Spring takes care of:
  1. Context loading and caching
     - The spring context is loaded only once for all DAO tests
       - The Hibernate mapping is read only once
  2. Automatic transaction creation and rollback
     - The tests don’t clutter the database
- This results in high performance during testing

# Testing DAO classes

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/springconfig.xml"})
public class CustomerDaoTest {

  @Autowired
  private CustomerDAO customerdao;

  @Test
  public void testPersist(){
      String query = "select count(*) from customer";
      Integer count=null;

      // check if the DB is empty
      Session session= customerdao.getSessionFactory().openSession();
      count = (Integer)session.createSQLQuery(query).uniqueResult();
      session.close();
      assertEquals("A customer already exists in the DB", 0, count.intValue());

      // save a customer
      Customer customer = new Customer("Frank","Brown");
      customerdao.saveCustomer(customer);

      // check if a customer is added to the DB
      session= customerdao.getSessionFactory().openSession();
      count = (Integer)session.createSQLQuery(query).uniqueResult();
      session.close();
      assertEquals("User was not found in the DB", 1, count.intValue());
  }
}
```

The DAO is injected

# Automatic rollback

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/springconfig.xml"})
@Transactional
public class CustomerDaoTest {

  @Autowired
  private CustomerDAO customerdao;

  @Test
  public void testPersist(){
      String query = "select count(*) from customer";
      Integer count=null;

      // check if the DB is empty
      Session session= customerdao.getSessionFactory().openSession();
      count = (Integer)session.createSQLQuery(query).uniqueResult();
      session.close();
      assertEquals("A customer already exists in the DB", 0, count.intValue());

      // save a customer
      Customer customer = new Customer("Frank","Brown");
      customerdao.saveCustomer(customer);

      // check if a customer is added to the DB
      session= customerdao.getSessionFactory().openSession();
      count = (Integer)session.createSQLQuery(query).uniqueResult();
      session.close();
      assertEquals("User was not found in the DB", 1, count.intValue());
  }
}
```

Automatic rollback after every test method

# Control transactional behavior

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/springconfig.xml"})
@Transactional
public class ClassTest {

  @BeforeTransaction
  public void verifyDatabaseStateBeforeTx() {  }

  @AfterTransaction
  public void verifyDatabaseStateAfterTx() {  }

  @Test
  @Rollback(false)
  public void testmethodA(){ }

  @Test
  @NotTransactional
  public void testmethodB(){ }
}
```

Verify the db state before a transaction is started

Verify the db state after a transaction is rolled back

Do not automatically rollback this transaction

Do not run this method within a transaction

# Summary

- Unit testing should be done without Spring
- The Spring test framework helps you to do integration testing
  - Test the Spring configuration
  - Test DAO classes
- The Spring test framework provides
  - Context loading and caching
  - Automatic transaction creation and rollback