Creating Data-Driven Tests With SpringUnit

Ted Velkoff
Senior Software Engineer
Integrity One Partners, Inc.
1900 Campus Commons Drive
Suite 150
Reston, VA 20191
email: ted.velkoff@ionep.com

Who Will Benefit

- Java/J2EE/Spring developers who write unit and integration tests
- Java/J2EE/Spring architects and lead developers with responsibility for development testing
- Software professionals who are serious about testing

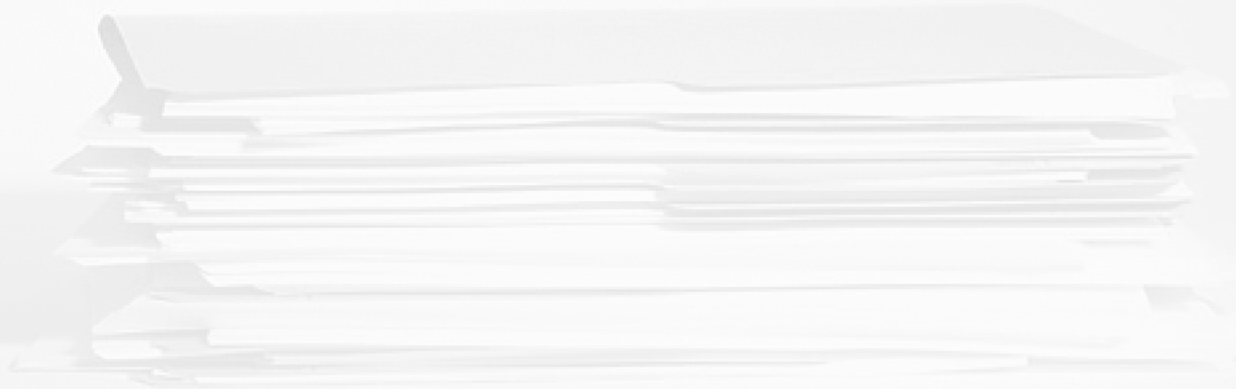
What You Will Learn

- Create data-driven tests using SpringUnit
- Create unit tests of domain model objects
- Create unit tests of business logic objects using mock objects
- Create integration tests of data access objects
- Create integration tests of systems
- Understand when and how to use SpringUnit

Agenda

- Quick Overview
- SpringUnit Test Framework
- Case Study

Part I
Quick Overview



What Is SpringUnit?

Spring* + JUnit† = SpringUnit

- Open source framework for unit and integration testing of Java software
- Marries Spring’s dependency injection container with JUnit’s test framework
- Enables data-driven testing

* 1.2.8 † 3.8.1

SpringUnit Value Proposition

- Allows test data and test code to vary independently
- Enables reuse of test data values across different tests
- Improves maintainability and readability of test code
- Leverages familiar schema for describing test data (Spring beans)

Example: JUnit Test

MyClass.java

```
public class MyClass {

  public int square(int i) {
    return i * i;
  }

}
```

MyClassTest.java

```
public class MyClassTest extends TestCase {

  public void testSquarePos() throws Exception {
    MyClass subject = new MyClass();
    Integer input = new Integer(3);
    Integer expected = new Integer(9);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }

  public void testSquareNeg() throws Exception {
    MyClass subject = new MyClass();
    Integer input = new Integer(-3);
    Integer expected = new Integer(9);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }

  /* etc. */

}
```

Example: SpringUnit Test

MyClassTest.java

```
public class MyClassTest extends SpringUnitTest {

  public void testSquarePos() throws Exception {
    runSquare();
  }
  public void testSquareNeg() throws Exception {
    runSquare();
  }
  protected void runSquare() throws Exception {
    MyClass subject = getObject(“subject”);
    Integer input = getObject(“input”);
    Integer expected = getObject(“expected”);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }








}
```

Example: SpringUnit Test

MyClassTest.java

```
public class MyClassTest extends SpringUnitTest {

  public void testSquarePos() throws Exception {
    runSquare();
  }
  public void testSquareNeg() throws Exception {
    runSquare();
  }
  protected void runSquare() throws Exception {
    MyClass subject = getObject(“subject”);
    Integer input = getObject(“input”);
    Integer expected = getObject(“expected”);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }




}
```

MyClassTest.xml

```
<beans>
<bean id=“myClassTest” class=“SpringUnitContext”>
<property name=“data”>
<map>
<entry key=“testSquarePos”>
<map>
<entry key=“subject”><bean class=“MyClass”/></entry>
<entry key=“input”><value type=“int”>3</value></entry>
<entry key=“expected”><value type=“int”>9</value></entry>
</map>
</entry>
<entry key=“testSquareNeg”>
<map>
<entry key=“subject”><bean class=“MyClass”/></entry>
<entry key=“input”><value type=“int”>-3</value></entry>
<entry key=“expected”><value type=“int”>9</value></entry>
</map>
</entry>
</map>
</property>
</bean>
</beans>
```

Example: SpringUnit Test

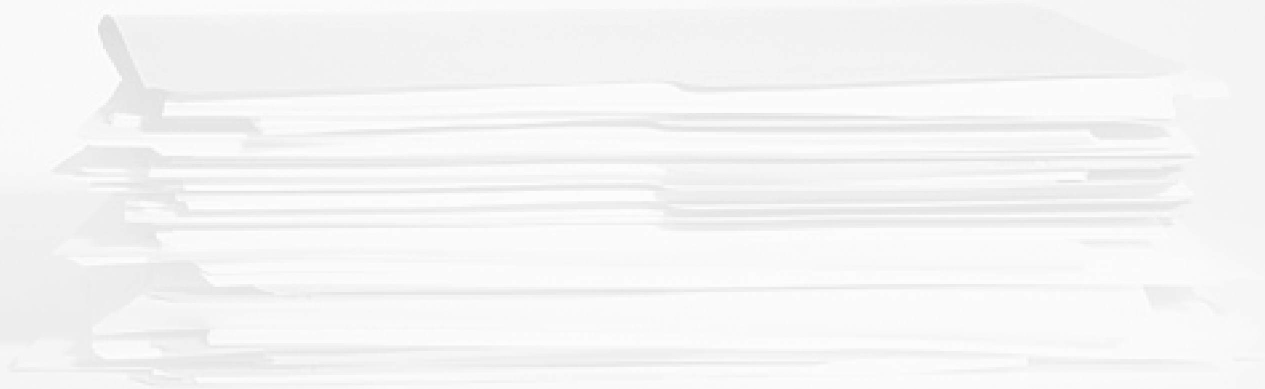
MyClassTest.java

```
public class MyClassTest extends SpringUnitTest {

  public void testSquarePos() throws Exception {
    runSquare();
  }
  public void testSquareNeg() throws Exception {
    runSquare();
  }
  protected void runSquare() throws Exception {
    MyClass subject = getObject(“subject”);
    Integer input = getObject(“input”);
    Integer expected = getObject(“expected”);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }
  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }
  private SpringUnitContext myClassTest;
}
```

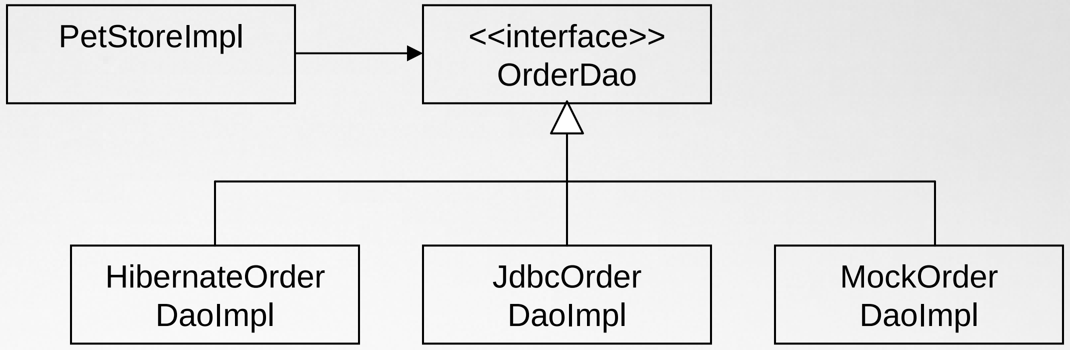
MyClassTest.xml

```
<beans>
<bean id=“myClassTest” class=“SpringUnitContext”>
<property name=“data”>
<map>
<entry key=“testSquarePos”>
<map>
<entry key=“subject”><bean class=“MyClass”/></entry>
<entry key=“input”><value type=“int”>3</value></entry>
<entry key=“expected”><value type=“int”>9</value></entry>
</map>
</entry>
<entry key=“testSquareNeg”>
<map>
<entry key=“subject”><bean class=“MyClass”/></entry>
<entry key=“input”><value type=“int”>-3</value></entry>
<entry key=“expected”><value type=“int”>9</value></entry>
</map>
</entry>
</map>
</property>
</bean>
</beans>
```

Part II
SpringUnit Framework
In Depth



Problem At Architectural Boundary



PetStoreImpl.java

```
public class PetStoreImpl {
  private OrderDao orderDao;
  public PetStoreImpl {
    this.orderDao = new ???
  }
  public void insertOrder(Order order) {
    // business logic to test
    this.orderDao.insertOrder(order);
    // etc.
  }
}
```

Dependency Injection

- Coding to interfaces at architectural boundaries is good practice
- Use of new operator hard-codes actual implementation in client of interface
- Dependency injection
  - moves object creation to framework
  - restores dependency on interface only
  - facilitates configurable implementation
  - facilitates testing

Dependency Injection in Spring

PetStoreImpl.java

```
public class PetStoreImpl {
  private OrderDao orderDao;
  public OrderDao getOrderDao {
    return this.orderDao;
  }
  public void setOrderDao(OrderDao o) {
    this.orderDao = o;
  }
  public void insertOrder(Order order) {
    // business logic to test
    this.orderDao.insertOrder(order);
    // etc.
  }
}
```

daoContext.xml

```
<beans>
  <import resource="domainContext.xml"/>
  <bean class="OrderDaoHibernate">
    <property name="sessionFactory">
      <ref bean="sessionFactory"/>
    </property>
    <property name="hibernateTemplate">
      <!-- configuration info -->
    </property>
  </bean>
</beans>
```

serviceContext.xml

```
<beans>
  <import resource="daoContext.xml"/>
  <bean class="PetStoreImpl">
    <property name="orderDao">
      <ref bean="orderDao"/>
    </property>
  </bean>
</beans>
```

Spring

- Suite of Java/J2EE frameworks (Johnson et al)
- Dependency Injection container
  - Leverages Java Beans specification
  - Beans defined in external (XML) files
  - Lightweight container wires objects at runtime
  - Benefits: configuration, testability
- Supplemental classes for integration testing
  - Support in-container testing (transactional and non-transactional)
  - Extend JUnit TestCase

JUnit

- Java framework for unit tests (Gamma, Beck)
- TestCase contains testXxx methods
- TestRunner executes testXxx methods in indeterminate order
- setUp, tearDown called before and after each test
- getName() returns “testXxx” for each test
- Tests should be stateless

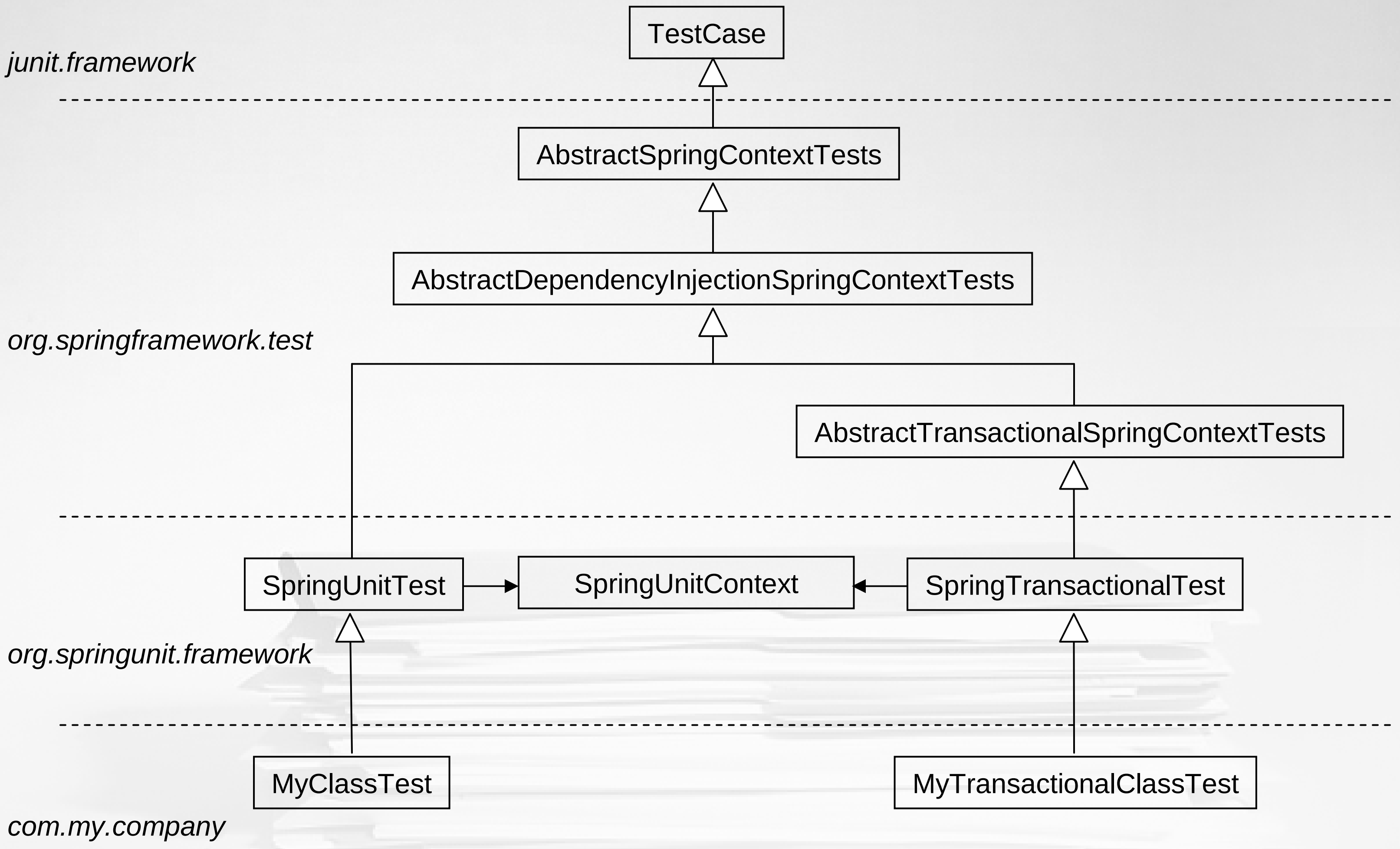
SpringUnit

- Extends Spring’s integration test classes
- Associates XML file of data values with each Java test class
- Introduces hierarchical, scoped context for test data values
- Finds data values by name in proper scope behind simple API
- Supports unit and integration testing

Design Goals and Constraints

- Simplicity for developers: convention over configuration
- No modification of JUnit test framework
- No modification of Spring extensions of JUnit test framework
- Must support inheritance of test classes

Framework Design



Recipe: How To Test MyClass

1. Create MyClassTest.java that extends SpringUnitTest
2. Create MyClassTest.xml that is Spring bean container
3. Add property myClassTest of type SpringUnitContext to MyClassTest
4. Create bean myClassTest of class SpringUnitContext in MyClassTest.xml
5. Add property data with value map to bean myClassTest in MyClassTest.xml
6. For every method testXxx in MyClassTest, add an entry to the data map in MyClassTest.xml whose value itself is a map
7. Define data values in MyClassTest.xml as entry/value pairs in maps for each testXxx
8. Retrieve data values in each testXxx method by calling getObject("some name")

Recipe: Annotated Files

① **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {
















}
```

Recipe: Annotated Files

① **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

}
```

MyClassTest.xml ②

```
<beans>

</beans>
```

Recipe: Annotated Files

1 **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; 3

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }

}
```

MyClassTest.xml 2

```
<beans>

</beans>
```

Recipe: Annotated Files

① **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; ③

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }

}
```

MyClassTest.xml ②

```
<beans>
  <bean id="myClassTest"        ④
     class="SpringUnitContext">

  </bean>
</beans>
```

Recipe: Annotated Files

1 **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; 3

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }

}
```

MyClassTest.xml 2

```
<beans>
  <bean id="myClassTest"          4
     class="SpringUnitContext">
    <property name="data">   5
      <map>

      </map>
    </property>
  </bean>
</beans>
```

Recipe: Annotated Files

1 **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; (3)

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }
                        (6)
  public void testSquarePos() throws Exception {




  }
                        (6)
  public void testSquareNeg() throws Exception {
    /* More test code */
  }

}
```

MyClassTest.xml 2

```
<beans>
  <bean id="myClassTest"            (4)
    class="SpringUnitContext">
    <property name="data"> (5)
      <map>
        <entry key="testSquarePos"> (6)
          <map>




          </map>
        </entry>
        <entry key="testSquareNeg"> (6)
          <!-- data values -->
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Recipe: Annotated Files

① **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; ③

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }
                   ⑥
  public void testSquarePos() throws Exception {





  }
                   ⑥
  public void testSquareNeg() throws Exception {
    /* More test code */
  }

}
```

MyClassTest.xml ②

```
<beans>
  <bean id="myClassTest"          ④
     class="SpringUnitContext">
    <property name="data"> ⑤
      <map>
        <entry key="testSquarePos"> ⑥
          <map>
            <entry key="subject">
              <bean class="MyClass"/>
            </entry>                ⑦
            <entry key="input">
              <value type="int">3</value>
            </entry>
            <entry key="expected">
              <value type="int">9</value>
            </entry>
          </map>
        </entry>
        <entry key="testSquareNeg"> ⑥
          <!-- data values -->
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Recipe: Annotated Files

① **MyClassTest**.java

```
public class MyClassTest extends SpringUnitTest {

  private SpringUnitContext myClassTest; ③

  public SpringUnitContext getMyClassTest() {
    return this.myClassTest;
  }
  public void setMyClassTest(
      SpringUnitContext myClassTest) {
    this.myClassTest = myClassTest;
  }
                  ⑥
  public void testSquarePos() throws Exception {
    MyClass subject = getObject(“subject”);
    Integer input = getObject(“input”);       ⑧
    Integer expected = getObject(“expected”);
    Integer actual = subject.square(input);
    assertEquals(expected, actual);
  }
                  ⑥
  public void testSquareNeg() throws Exception {
    /* More test code */
  }

}
```

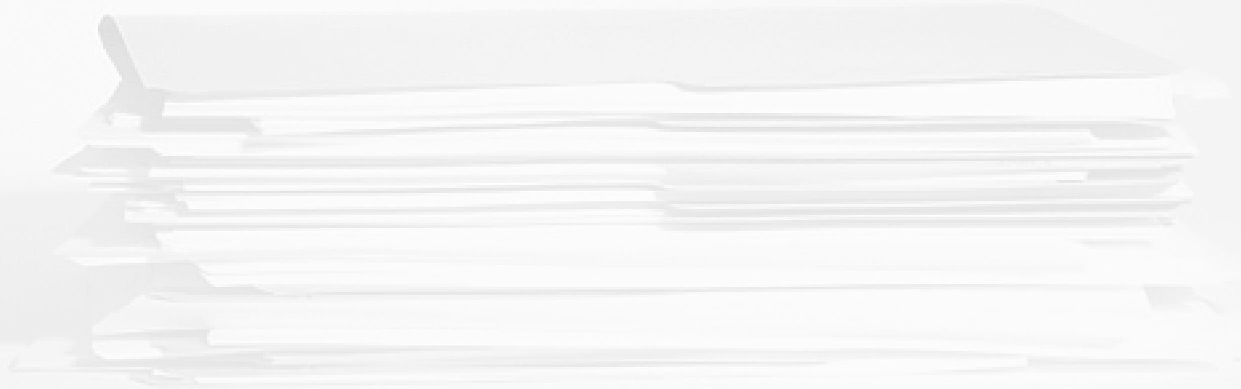
MyClassTest.xml ②

```
<beans>
  <bean id=“myClassTest”          ④
     class=“SpringUnitContext”>
    <property name=“data”> ⑤
      <map>
        <entry key=“testSquarePos”> ⑥
          <map>
            <entry key=“subject”>
              <bean class=“MyClass”/>
            </entry>              ⑦
            <entry key=“input”>
              <value type=“int”>3</value>
            </entry>
            <entry key=“expected”>
              <value type=“int”>9</value>
            </entry>
          </map>
        </entry>
        <entry key=“testSquareNeg”> ⑥
          <!-- data values -->
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

SpringUnit Web Site

- SpringUnit on Sourceforge
  - http://springunit.sourceforge.net
  - http://sourceforge.net/projects/springunit
- Information on the web site
  - Getting Started guide
  - Tutorial
  - Common Errors
  - Eclipse Plug-in

Demo #1:
SpringUnit Web Site



SpringUnit in Maven Repository

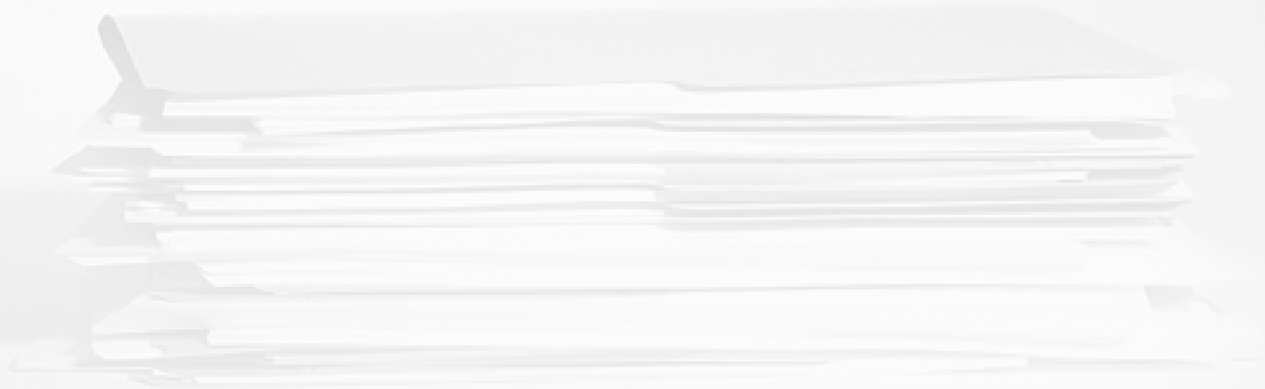
pom.xml

```
<project>
  <!-- other stuff -->
  <dependencies>
    <dependency>
      <groupId>org.springunit</groupId>
      <artifactId>springunit</artifactId>
      <version>0.5</version>
    </dependency>
    <!-- other dependencies -->
  </dependencies>
</project>
```

SpringUnit Eclipse Plug-in

- Jump starts the creation of new tests
- Extends the familiar JUnit Eclipse plug-in
- Creates skeleton Java code and XML file
- Ensures all SpringUnit naming conventions enforced
- Supports subclassing and superclassing of existing SpringUnit tests

Demo #2:
SpringUnit Eclipse Plug-in



SpringUnit Pros

- Create data-driven tests with reusable data values
- Seamless JUnit-based tests from unit to integration to system test
- Leverages well-known XML-based data description language (Spring beans)
- Eliminates need to use (vendor-specific) SQL and DDL to create persistent data values
- Separation of code and data makes tests easier to understand

SpringUnit Cons

- Separation of code and data makes tests harder to understand
- XML is a verbose DDL
- Spring beans DDL introduces level of indirection

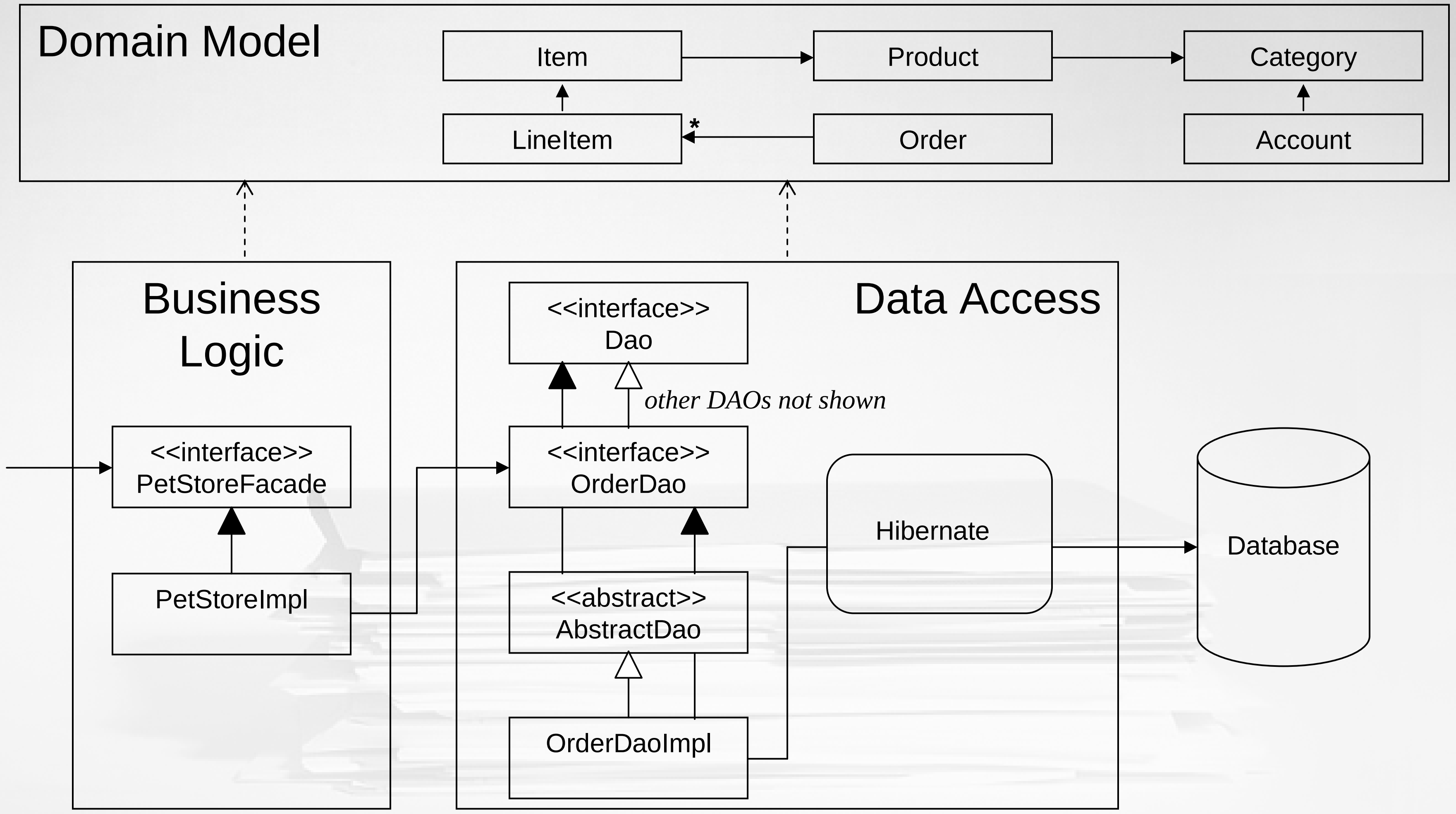
Is SpringUnit Right For You?
Beauty Lies in the Eye of the Beholder

Part III
Case Study: Testing the JPetStore Sample Application With SpringUnit

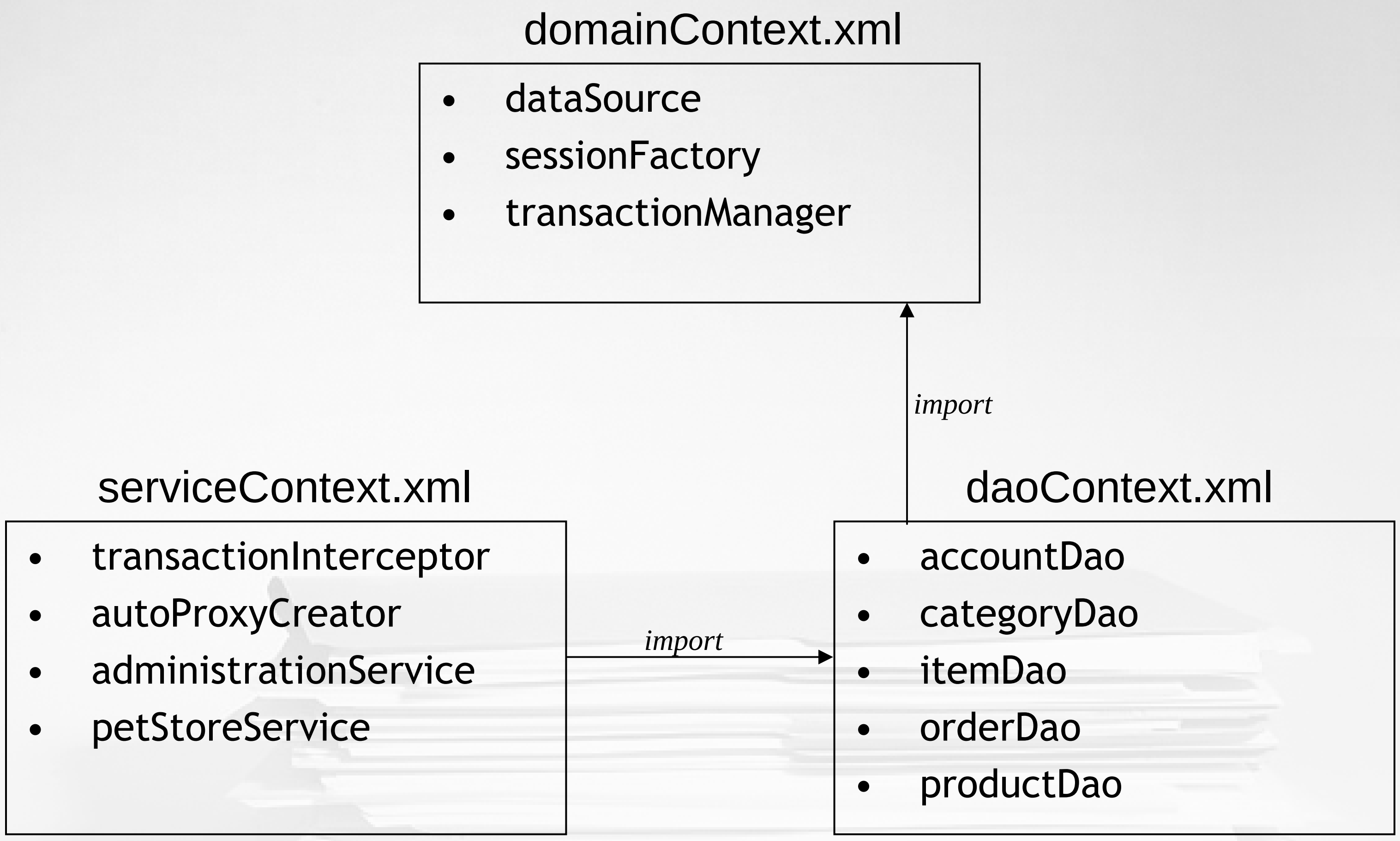
Case Study Overview

- JPetStore adapted from Spring Framework
- Domain Model Objects
- Business Logic Façade
- Data Access Objects
- O/R mapping using Hibernate
- Transactions introduced in façade using Spring AOP
- System wired together using Spring Dependency Injection

System Under Test



Demo #3:
How the Case Study Is Wired



Test Strategy For the Case Study

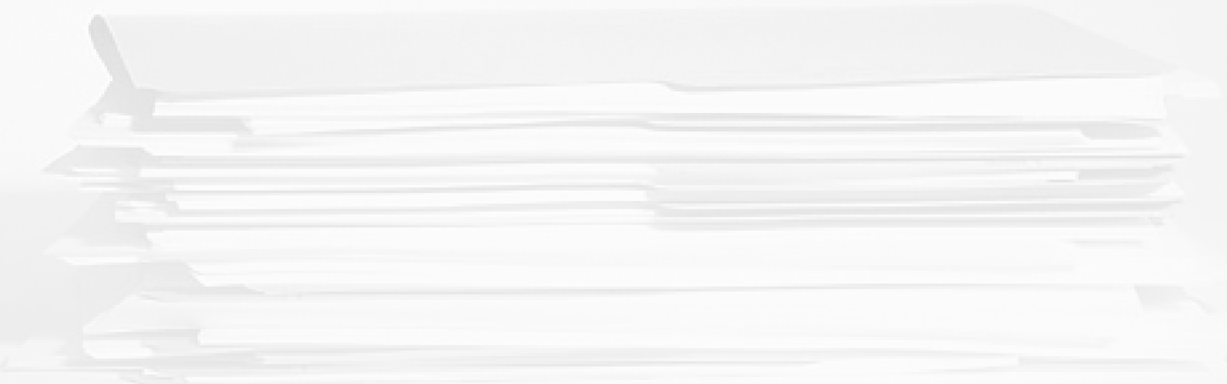
- Create unit tests of domain model
  - reusable data values can be developed here
- Create unit tests of business logic
  - use mock objects for DAOs
- Create integration tests of persistence layer
  - rollback transactions for stateless testing
- Create system tests of everything together
  - ensure that transactions complete
  - use compensating transactions for statelessness

Demo #4:
Goal: Complete Unit and Integration Testing of the Case Study

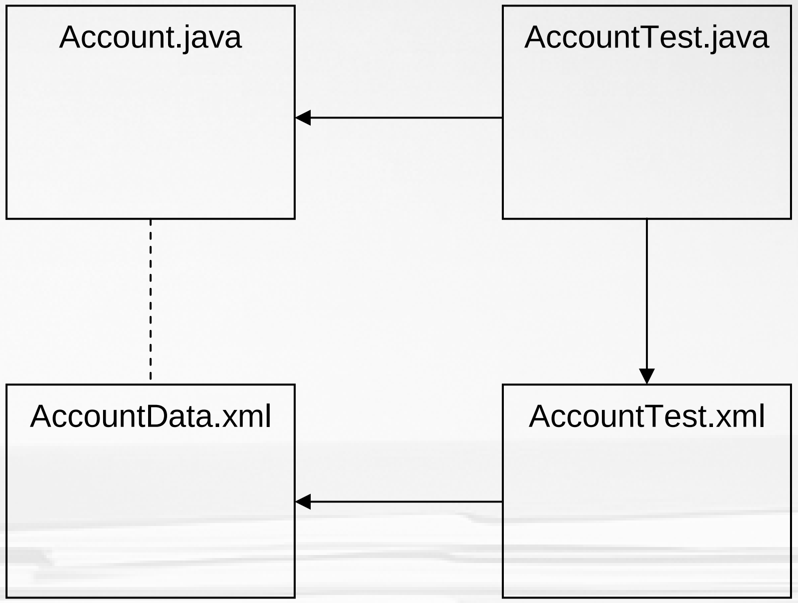
Maven-generated web site with test results

Unit Test of Domain Model Objects

- Apply testing techniques already shown
- Opportunity to begin building set of reusable test data



Demo #5:
Domain Model Unit Tests



Unit Test of Business Logic Objects

- Create unit test of façade implementation without requiring database, O/R mapping, transactions
- Use EasyMock to simulate behavior of DAOs
- Use Spring dependency injection to associate mock objects with façade implementation
- Use SpringUnit to obtain data values for testing

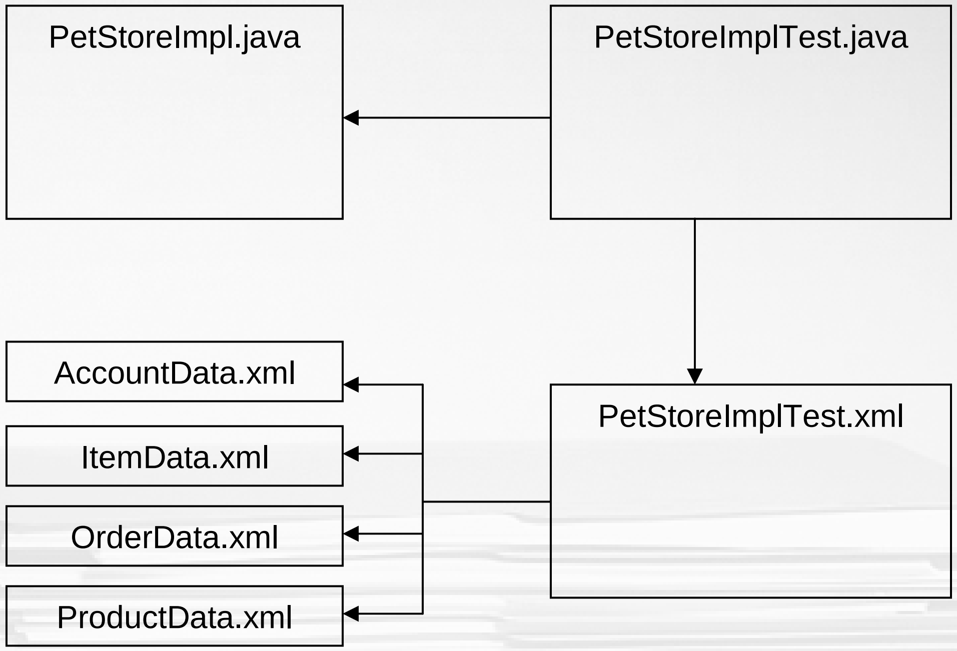
SpringUnit test with EasyMock

MyClassTest.java

```
public void testInsertOrder() throws Exception {
  PetStoreImpl subject = getObject(“subject”);
  Order order = getObject(“order”);
  ItemDao<Item> itemDao = subject.getItemDao();        1
  OrderDao<Order> orderDao = subject.getOrderDao();
  Object[] mocks = new Object[]{itemDao, orderDao};

  orderDao.insertOrder(order);              2
  itemDao.updateQuantity(order);
  EasyMock.replay(mocks);                   3

  subject.insertOrder(order);               4

  EasyMock.verify(mocks);                   5
}
```

1. For convenience, retrieve DAOs from subject and create array
2. Execute expected calls on mocked DAOs
3. Replay mocks to record expected behavior
4. Call method under test
5. Verify that actual behavior of mocks matches expected behavior

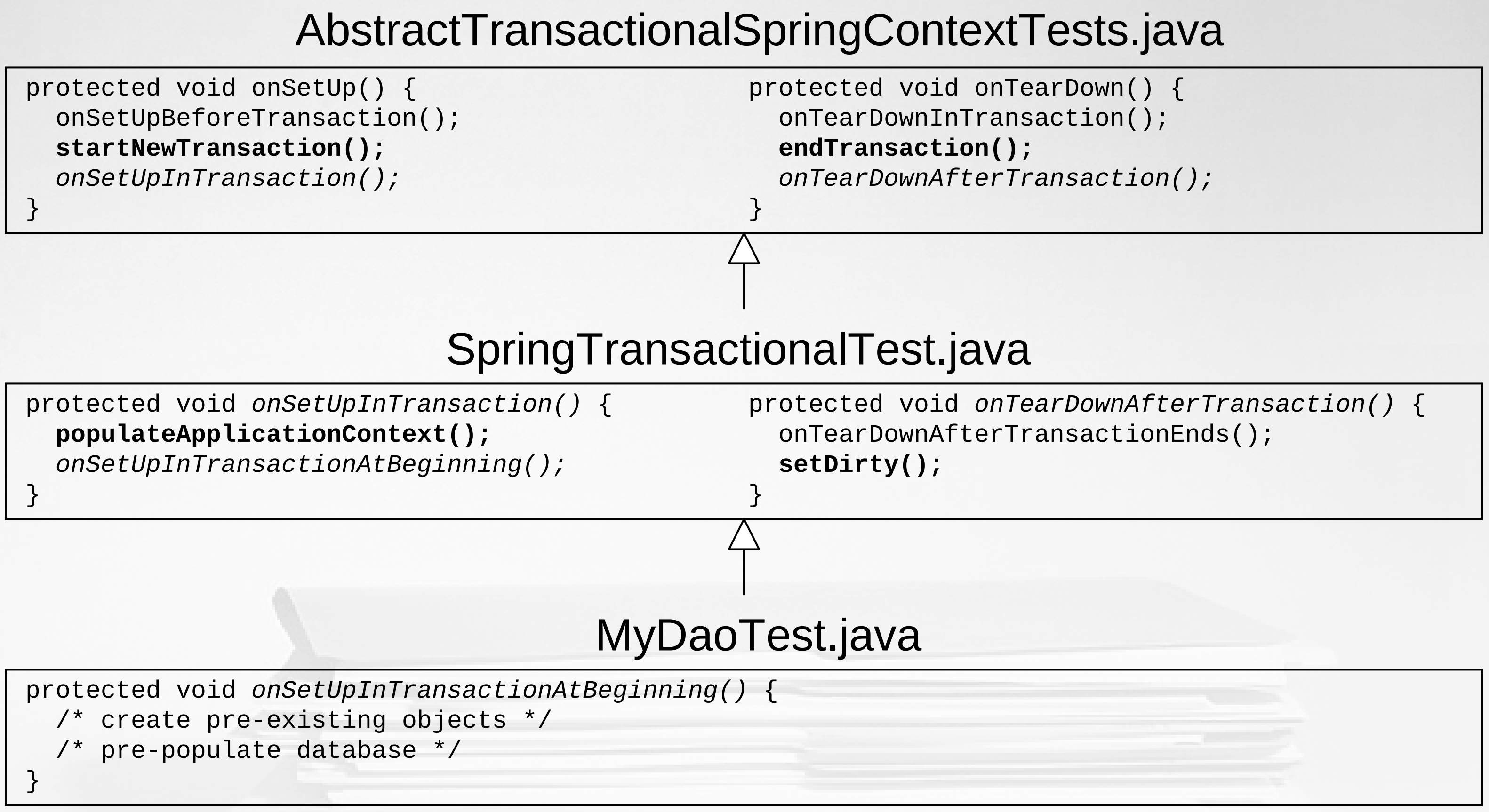
Demo #6:
Business Logic Unit Tests



Integration Test of Data Access Objects

- Execute DAO methods inside transaction, then roll back
- Create and delete dependencies, pre-populated objects inside same transaction
- Tests of CRUD operations shared by all DAOs
- Tests of finders and DAO-specific methods unique to each DAO

SpringTransactionalTest Design



AbstractTransactionalSpringContextTests.java

```
protected void onSetUp() {
  onSetUpBeforeTransaction();
  startNewTransaction();
  onSetUpInTransaction();
}
```

```
protected void onTearDown() {
  onTearDownInTransaction();
  endTransaction();
  onTearDownAfterTransaction();
}
```

SpringTransactionalTest.java

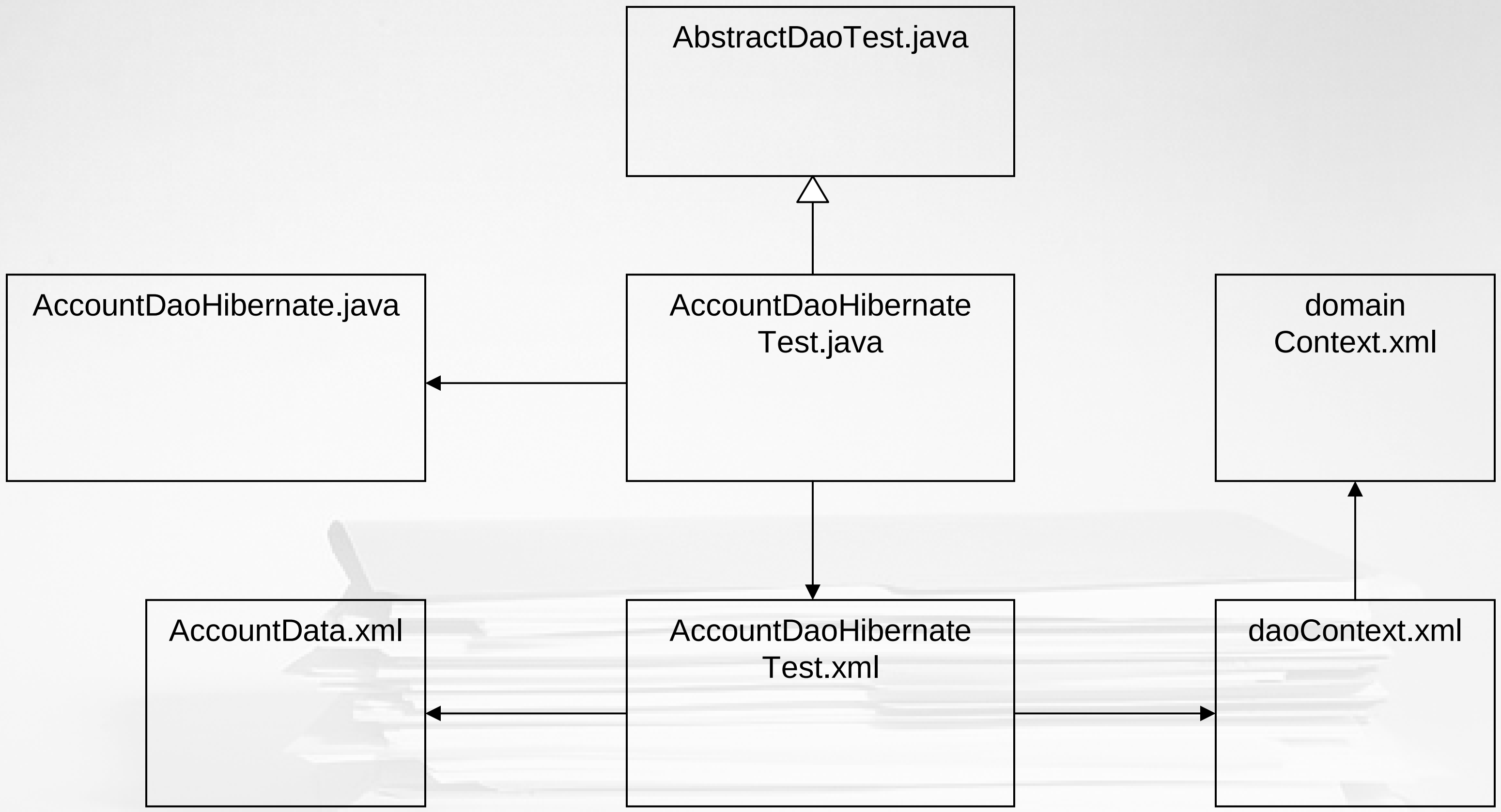
```
protected void onSetUpInTransaction() {
  populateApplicationContext();
  onSetUpInTransactionAtBeginning();
}
```

```
protected void onTearDownAfterTransaction() {
  onTearDownAfterTransactionEnds();
  setDirty();
}
```

MyDaoTest.java

```
protected void onSetUpInTransactionAtBeginning() {
  /* create pre-existing objects */
  /* pre-populate database */
}
```

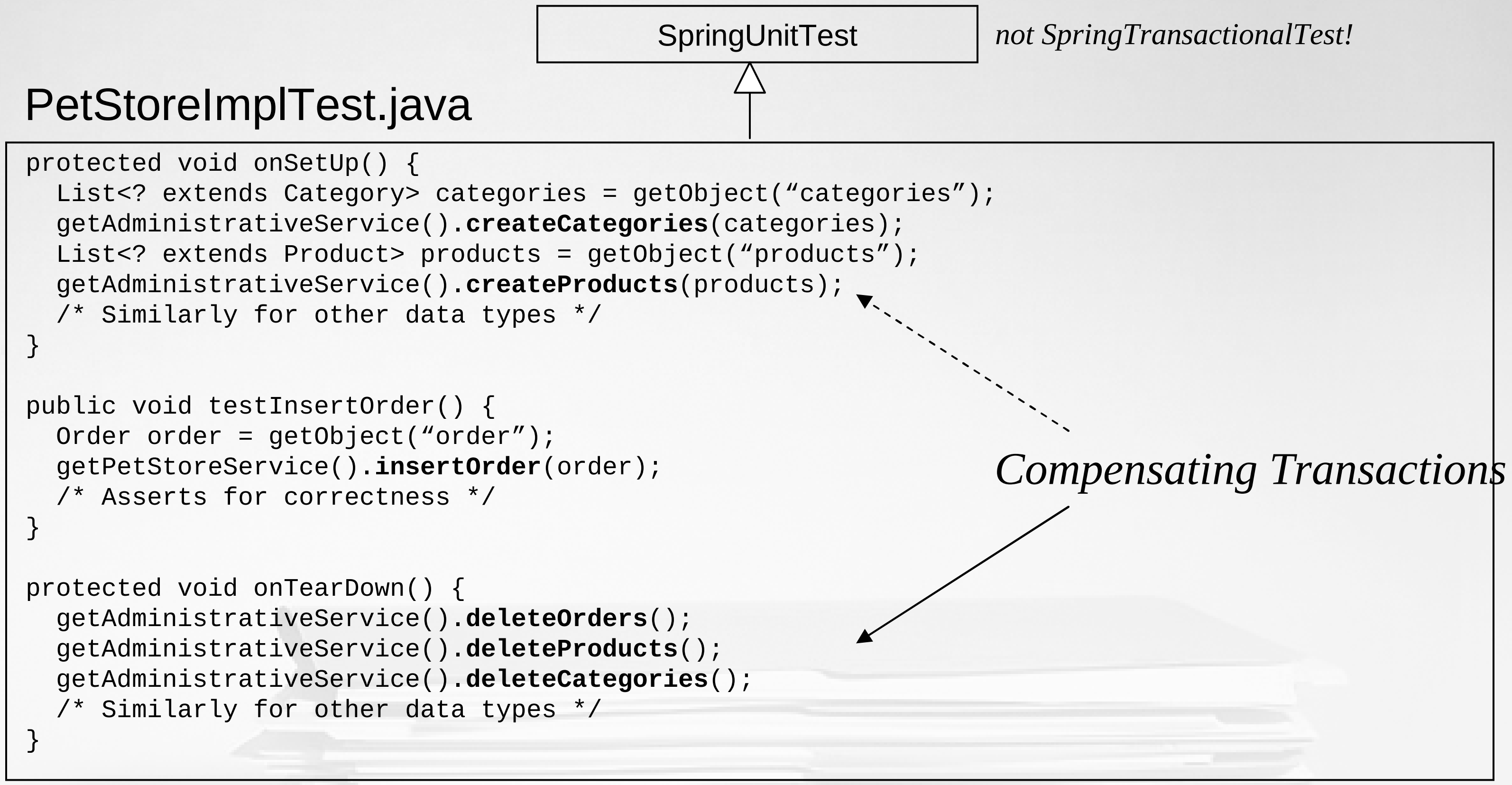
Demo #7: Persistence Layer Integration Tests



Integration Test of the System

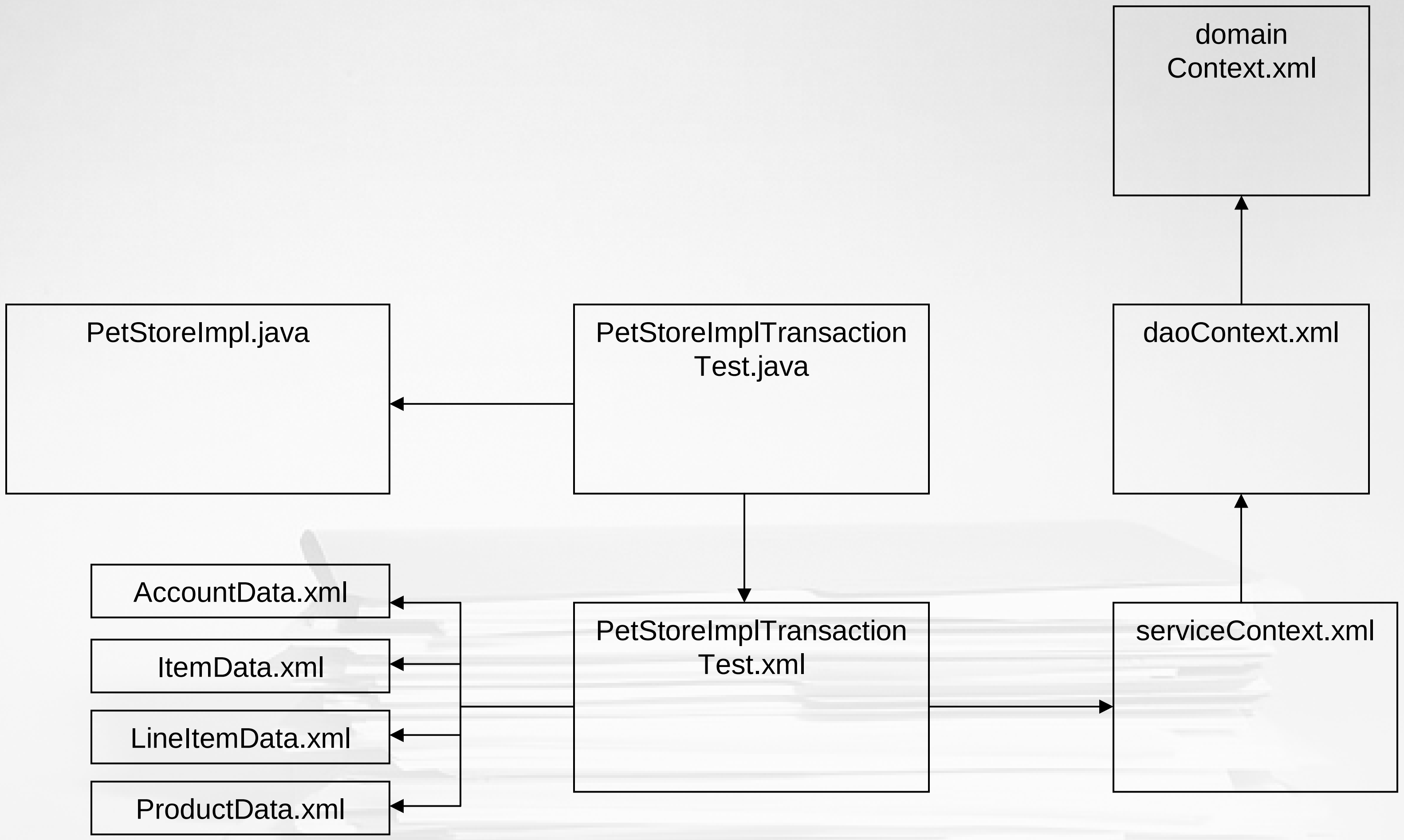
- Perform end-to-end integration of system, including database and transactions
- Establish initial state using database transactions (onSetUp)
- Execute transactional tests of façade methods
- Restore clean state using compensating transactions (onTearDown)

System Test Design



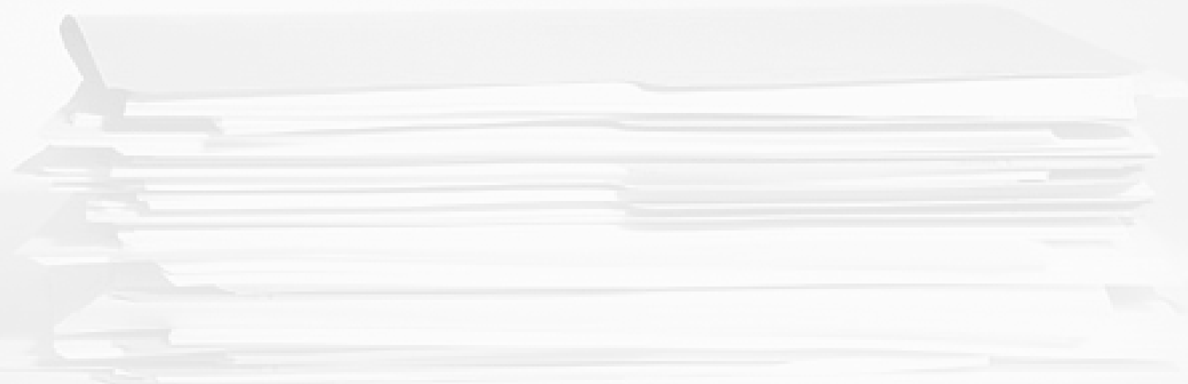
```
protected void onSetUp() {
  List<? extends Category> categories = getObject("categories");
  getAdministrativeService().createCategories(categories);
  List<? extends Product> products = getObject("products");
  getAdministrativeService().createProducts(products);
  /* Similarly for other data types */
}

public void testInsertOrder() {
  Order order = getObject("order");
  getPetStoreService().insertOrder(order);
  /* Asserts for correctness */
}

protected void onTearDown() {
  getAdministrativeService().deleteOrders();
  getAdministrativeService().deleteProducts();
  getAdministrativeService().deleteCategories();
  /* Similarly for other data types */
}
```

Demo #8: System Integration Test



Key Observations From the Case Study

- Data values reused across unit and integration tests
- Persistent data values described in one format (Spring beans) throughout
- All tests are stateless

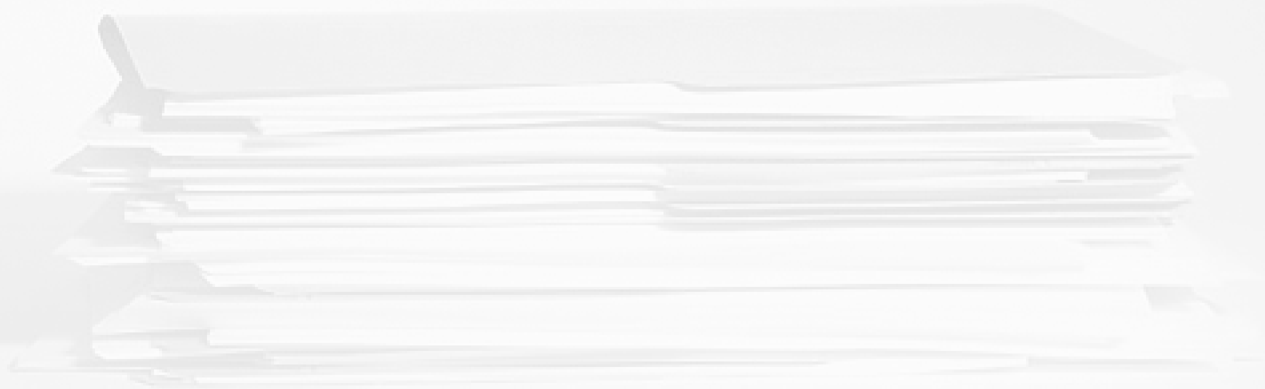


Summary

- SpringUnit marries Spring and JUnit
- SpringUnit facilitates data-driven tests
- SpringUnit enables reuse of data values
- SpringUnit supports unit and integration testing
- SpringUnit jar posted in Maven Repository
- SpringUnit code, documentation, Eclipse plug-in all available at Sourceforge web site

  http://springunit.sourceforge.net

Questions?



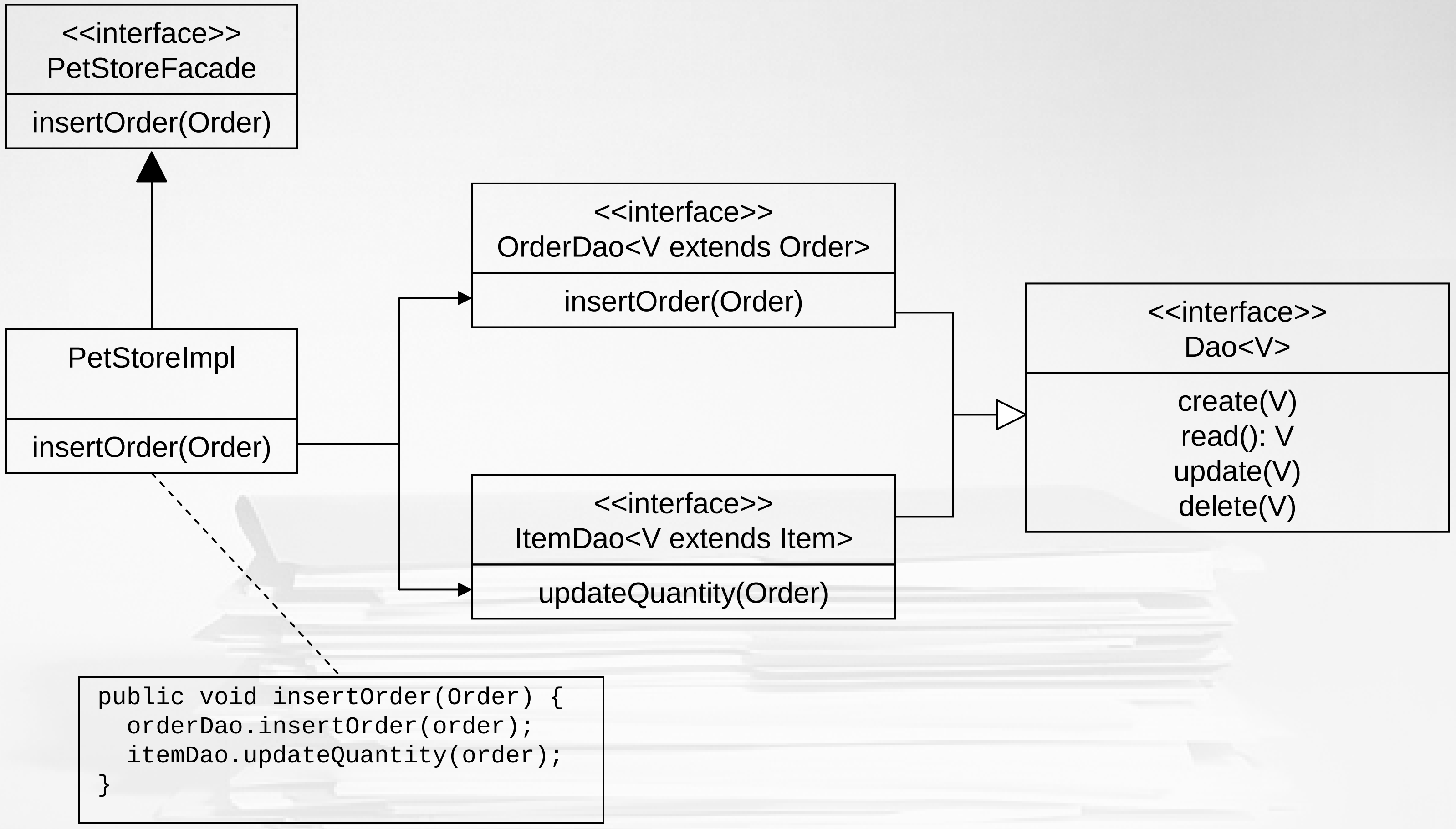
Thank You For Attending

Ted Velkoff
Senior Software Engineer
Integrity One Partners, Inc.
1900 Campus Commons Drive
Suite 150
Reston, VA 20191
email: ted.velkoff@ionep.com

How To Get SpringUnit

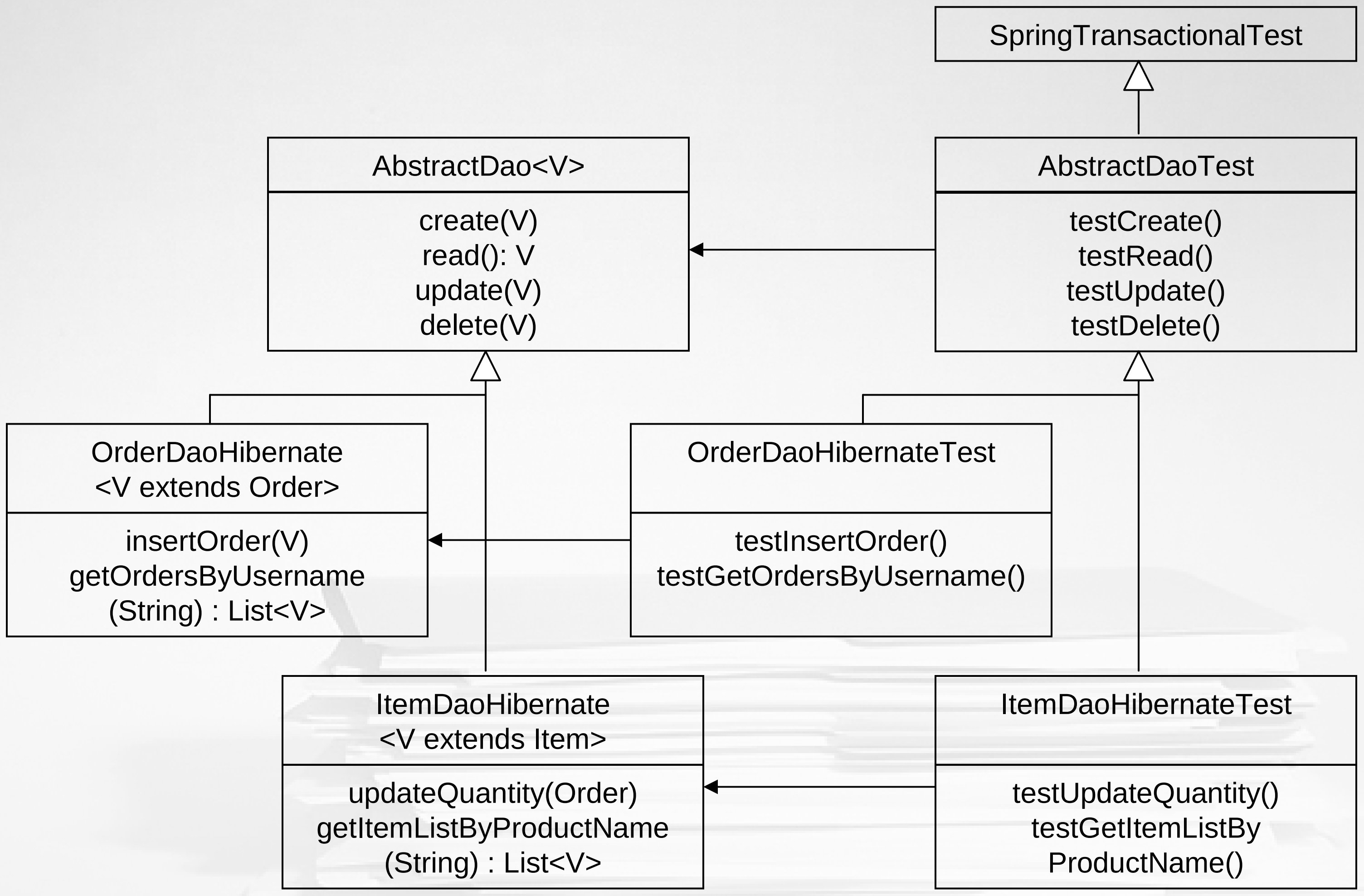
http://sourceforge.net/projects/springunit

Façade / Data Access Design



```
public void insertOrder(Order) {
  orderDao.insertOrder(order);
  itemDao.updateQuantity(order);
}
```

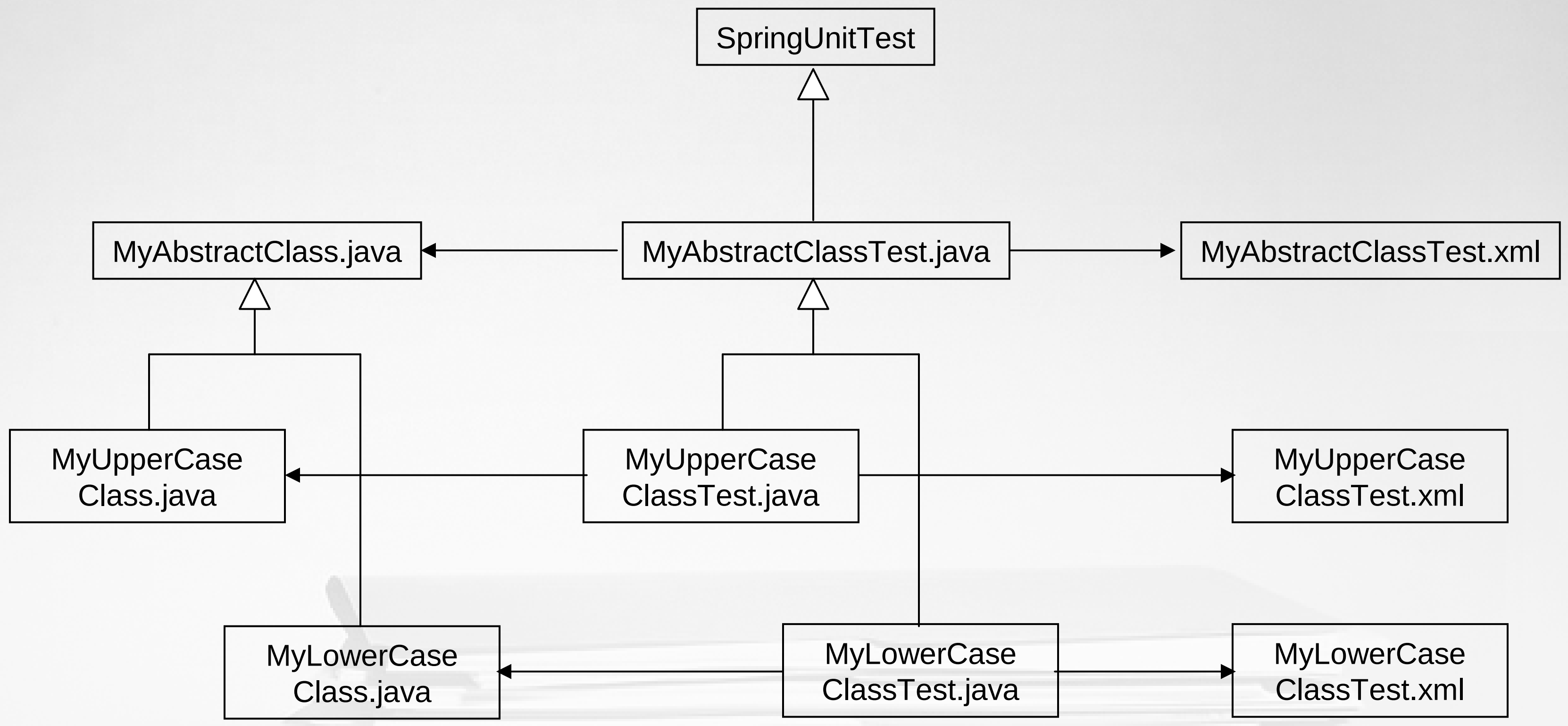
Data Access Implementation and Test



Expecting Exceptions

```
public void testInsertOrder() throws Exception {
  Order order = getObject(“order”);
  OrderDao<Order> subject = getObject(“subject”);
  Exception exception = getObject(“exception”);
  try {
    subject.insertOrder(order);
    if (exception != null) {
      fail(“Exception not thrown”);
    }
  }
  catch (Exception ex) {
    if (exception != null &&
        !exception.getClass().isAssignableFrom(ex.getClass())) {
      throw ex;
    }
  }
}
```

Hierarchical Tests: Pattern



Hierarchical Tests: Classes Under Test

MyAbstractClass.java

```
public abstract class MyAbstractClass {
  public String getSentence(String name) {
    return transform(getGreeting() + " " + name +
      getPunctuation());
  }
  public String getGreeting() {
    return this.greeting;
  }
  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
  public String getPunctuation() {
    return this.punctuation;
  }
  public void setPunctuation(String punctuation){
    this.punctuation = punctuation;
  }
  protected abstract String transform();
  private String greeting;
  private String punctuation;
}
```

MyUpperCaseClass.java

```
public class MyUpperCaseClass
      extends MyAbstractClass {
  protected String transform(String sentence) {
    return sentence.toUpperCase();
  }
}
```

MyLowerCaseClass.java

```
public class MyLowerCaseClass
      extends MyAbstractClass {
  protected String transform(String sentence) {
    return sentence.toLowerCase();
  }
}
```

Hierarchical Tests: Test Code

MyAbstractClassTest.java

```
public abstract class MyAbstractClassTest
      extends SpringUnitTest {
  /* Getters/setters not shown */
  private SpringUnitContext myAbstractClassTest;

  public void testOne() throws Exception {
    runSentence();
  }
  public void testTwo() throws Exception {
    runSentence();
  }
  protected void runSentence() throws Exception {
    String greeting = getObject("greeting");
    String name = getObject("name");
    String punc = getObject("punctuation");
    String expected = getObject("expected");
    MyAbstractClass subj = getObject("subject");
    subj.setGreeting(greeting);
    subj.setPunctuation(punc);
    String actual = subj.getSentence(name);
    assertTrue(expected.equals(actual));
  }
}
```

MyUpperCaseClassTest.java

```
public class MyUpperCaseClassTest
      extends MyAbstractClassTest {
  public void testThree() throws Exception {
    /* Unique to this class */
  }
  /* Getters/setters not shown */
  private SpringUnitContext myUpperCaseClassTest;
}
```

MyLowerCaseClassTest.java

```
public class MyLowerCaseClassTest
      extends MyAbstractClassTest {
  public void testThree() throws Exception {
    /* Unique to this class */
  }
  /* Getters/setters not shown */
  private SpringUnitContext myLowerCaseClassTest;
}
```

Hierarchical Tests: Test Data

MyAbstractClassTest.xml

```
<beans>
  <bean id=“myAbstractClassTest”
     class=“SpringUnitContext”>
    <property name=“data”>
      <map>
        <entry key=“greeting”>
          <value>Hello</value>
        </entry>
        <entry key=“testOne”>
          <map>
            <entry key=“name”>
              <value>World</value>
            </entry>
          </map>
        </entry>
        <entry key=“testTwo”>
          <map>
            <entry key=“name”>
              <value>Dolly</value>
            </entry>
          </map>
        </entry>
      </map>
    </property>
  </bean>
</beans>
```

Hierarchical Tests: Test Data

MyUpperCaseClassTest.xml

```
<!-- Inside data property of context bean -->
<map>
  <entry key=“punctuation”>
    <value>!</value>
  </entry>
  <entry key=“testOne”>
    <map>
      <entry key=“subject”>
        <bean class=“MyUpperCaseClass”/>
      </entry>
      <entry key=“expected”>
        <value>HELLO WORLD!</value>
      </entry>
    </map>
  </entry>
  <entry key=“testTwo”>
    <map>
      <entry key=“subject”>
        <bean class=“MyUpperCaseClass”/>
      </entry>
      <entry key=“expected”>
        <value>HELLO DOLLY!</value>
      </entry>
    </map>
  </entry>
</map>
```

MyLowerCaseClassTest.xml

```
<!-- Inside data property of context bean -->
<map>
  <entry key=“punctuation”>
    <value>?</value>
  </entry>
  <entry key=“testOne”>
    <map>
      <entry key=“subject”>
        <bean class=“MyLowerCaseClass”/>
      </entry>
      <entry key=“expected”>
        <value>hello world?</value>
      </entry>
    </map>
  </entry>
  <entry key=“testTwo”>
    <map>
      <entry key=“subject”>
        <bean class=“MyLowerCaseClass”/>
      </entry>
      <entry key=“expected”>
        <value>hello dolly?</value>
      </entry>
    </map>
  </entry>
</map>
```

Singleton data value

AccountData.xml

```
<beans>
  <bean id="account1" class="Account">
    <property name="username">
      <value>ted</value>
    </property>
  </bean>
  <bean id="account2" class="Account"
      singleton="false">
    <property name="username">
      <value>ted</value>
    </property>
  </bean>
</beans>
```

FirstAccountTest.xml

```
<import resource="classpath:AccountData.xml"/>

<!-- enclosing context elements -->
<entry key="testOne">
  <map>
    <entry key="account">
      <ref bean="account1"/>
    </entry>
  </map>
</entry>
<entry key="testTwo">
  <map>
    <entry key="account">
      <ref bean="account1"/>
    </entry>
  </map>
</entry>
<!-- enclosing context elements -->
```

testOne and testTwo share the same instance of Account

Non-singleton data value

AccountData.xml

```
<beans>
  <bean id="account1" class="Account">
    <property name="username">
      <value>ted</value>
    </property>
  </bean>
  <bean id="account2" class="Account"
      singleton="false">
    <property name="username">
      <value>ted</value>
    </property>
  </bean>
</beans>
```

SecondAccountTest.xml

```
<import resource="classpath:AccountData.xml"/>

<!-- enclosing context elements -->
<entry key="testOne">
  <map>
    <entry key="account">
      <ref bean="account2"/>
    </entry>
  </map>
</entry>
<entry key="testTwo">
  <map>
    <entry key="account">
      <ref bean="account2"/>
    </entry>
  </map>
</entry>
<!-- enclosing context elements -->
```

testOne and testTwo refer to different instances of Account

Another variation on singletons

FirstAccountTest.xml

```
<!-- enclosing context elements -->
<property name="data">
  <map>
    <entry key="subject">
      <bean class="Account"/>
    </entry>
    <entry key="testOne">
      <map><!-- various data values --></map>
    </entry>
    <entry key="testTwo">
      <map><!-- various data values --></map>
    </entry>
  </map>
</property>
<!-- enclosing context elements -->
```

SecondAccountTest.xml

```
<!-- enclosing context elements -->
<property name="data">
  <map>
    <entry key="testOne">
      <map>
        <entry key="subject">
          <bean class="Account"/>
        </entry>
      </map>
    </entry>
    <entry key="testTwo">
      <map>
        <entry key="subject">
          <bean class="Account"/>
        </entry>
      </map>
    </entry>
<!-- enclosing context elements -->
```

FirstAccountTest shares the same instance of Account across tests.
SecondAccountTest is usually better practice.

Test Context With EasyMock

PetStoreImplTest.xml

```
<beans>
  <import resource="classpath:OrderData.xml"/>
  <bean id="orderDao" class="org.easymock.EasyMock" factory-method="createMock" singleton="false">
    <constructor-arg>
      <bean class="java.lang.Class" factory-method="forName">
        <constructor-arg>
          <value>org.springunit.framework.samples.jpetstore.dao.OrderDao</value>
        <!-- etc. -->
  <bean id="itemDao" ... similarly ... />
  <bean id="subject" class="org.springunit.framework.samples.jpetstore.domain.logic.PetStoreImpl"
      singleton="false">
    <property name="orderDao"><ref local="orderDao"/></property>
    <property name="itemDao"><ref local="itemDao"/></property> <!-- etc. -->
  </bean>
  <bean id="petStoreImplTest" class="org.springunit.framework.SpringUnitContext">
    <property name="data">
      <map>
        <entry key="testInsertOrder">
          <map>
            <entry key="subject"><ref local="subject"></entry>
            <entry key="order"><ref bean="order1"/></entry> <!-- Reuse values from OrderData.xml -->
          <!-- etc. -->
```