Unit Testing

Fritz Solms

January 29, 2016

Contents

1 Introduction

Figure 1: Keep going – the pay-off will come.

In many ways unit testing is like exercising. You know its good for you but you hate the effort. When done constructively, unit testing makes your code considerably more flexible and reusable and it can result in improvements to the development process efficiency and productivity.

On the other hand, if unit tests are purely written because they are mandated by management, the positive effect may be lost and the effort can result in increased development and maintenance costs without much benefit to software quality. It is a bit like your sports teacher mandating activities you do not believe in – you will find all sorts of ways to sabotage system and make the effort useless.

1.1 What is unit testing?

A unit test tests a unit in isolation against its requirements. Often the requirements are formalized within a component contract. A unit can be a class, a service or a function.

Unit testing is a form of *functional testing*. I.e. non-functional requirements like supported access and integration channels as well as quality requirements (e.g. scalability, performance, security, auditability, modifiability, ...) are generally not tested within unit tests. The non-functional requirements are addressed by software architecture – some quality requirements are commonly applied to components via aspects.

Unit tests are either written by developers or generated from contract specifications. In either case, they are generally bound into the build cycle.

1.2 White-box versus black-box testing

Different practitioners use either white-box or black-box testing for unit testing. In the case of white-box testing one tests the internal working of the component. White box testing requires knowledge of implementation details (e.g. algorithm used). One has thus a separate test per concrete class, even for classes realizing the same contract (implementing the same interface). I would recommend to tie implementation-specific testing to the implementation where it can access aspects of the internal workings of the class. This can be done using assertions. programming languages which support assertions allow for the switching assertions on or off across different areas of the code base.

Black-box testing, on the other hand, assumes no knowledge of the implementation. It is used to test whether the component fulfills the requirements, i.e. whether the component realizes its contract. Black-box tests can thus be developed against a contract specification. Every class realizing the contract would then be tested against a single test - the test developed for the contract. I would recommend that unit tests should be developed as black-box test, testing whether any component implementing interface realizes component contract.

Note: When a component is changed it must still pass the unit test for the contract. This approach enforces the *open for extension, closed for modification* principle.

1.3 Benefits of unit testing

Unit testing may provide a range of benefits including:

1. Reduced system *failure risk*.

2. *Rapid feedback* on developed components,
3. Reduced *cost* due to
    - the development off well understood requirements being more stream-lined,
    - having to spend less time on bug fixes,
    - reduced integration problems, and
    - lower manual testing costs.
4. Improved *Maintainability* due to unit testing
    - making changes to system less risky,
    - removing the need for code ownership, and
    - simplifying continuous refactoring,
    - and unit testing with dependency injection enabling one to use unit tests as integration & regression tests.
5. Improved **Reusability** leading to less code being developed and maintained due to
    - components realizing specified contracts, and
    - the unit tests providing usage documentation.

1.4 Challenges for unit testing

There are a range of challenges around writing good tests. A good test needs to cover all aspects of the contract and needs to cover a wide range of scenarios through good selection of test data.

Another challenge is that of isolating the unit being tested. The unit may have all sorts of dependencies, interacting with its environment in various ways. One needs to fake/mock the environment in a controlled way.

From a corporate perspective the cost of unit testing is often a concern. Writing a good test may be very complex and at times may be no simpler than writing the component itself. The cost of testing can, however, be even negative (i.e. that testing reduces cost) due to the lowering in the costs incurred by bug fixing and maintenance. The cost of unit testing can be contained by (i) Reusing mock objects across tests (ii) Reusing tests across implementations of same contract (iii) Reusing unit tests for integration and regression tests

A further challenge is the evolution of unit testing in the context of an evolving system. In particular, different components may be plugged into a pluggable system and these component may be realized in different technologies. One possibility is to write the test against a technology abstraction layer (e.g. a web-services abstraction layer). The evolution of the system is also driven by continuous refactoring.

Finally, it is non trivial to change the culture of an organization toward a contract-driven approach with tests being written prior to the concrete components. In this context it is important for developers to understand the benefits they get from unit testing for their own development productivity and quality.

1.5 General approach

In contracts-based unit testing one would for each component perform the following steps:

1. *Understand the component requirements*
2. *Formalize in service/component contract*
3. *Develop one or more implementations*
    - e.g. class, service, function
4. *Test implementation against unit test for contract*

Note that *developing the unit test* prior to the component implementation deepens the understanding of the component requirements and component usage and any such understanding may be used to refine the contract. Furthermore, developing the test first simplifies component development and encourages modularization. The latter is particularly true when the difficulty in writing a unit test leads to further factorization of the system to make testing more feasible. The improved modularization leads to lower system complexity and increased reusability.

1.6 Alternatives to unit testing

One alternative to unit testing is to perform extensive integration as well as manual user testing. These typically incur high cost, are time consuming and error prone. Furthermore, reusing lower level components is more difficult and risky without unit tests.

Another alternative is to use formal methods to either verify the correctness of programs or to mathematically derive programs. This approach is, however, only viable for isolated critical code.

2 Simple parabola example

This simple example of an excerpt of a JUnit test for a particular class. This is not an example of testing a contract. Nevertheless the test on whether the roots are calculated correctly is independent of the algorithm used to calculate the roots.

It tests whether the roots are calculated correctly. For this it verifies (i) that the function value at a found root is indeed zero, and (ii) that the roots for specific test data is indeed equal to some hard-coded root values.

Test data is chosen as samples from different equivalence partitions and includes boundary values (e.g. 2 positive roots, positive and negative root, single root, no real roots).

2.1 The class under test

This example uses as test subject a simple parabola class with (i) a constructor which raises exception when trying to create non-parabola, (ii) a `getValue` method returning function value, (iii) a `getTurningPoint` method, and (iv) a `getRoots` method returning 2, 1 or zero real roots.

```
package za.co.solms.math;

public class Parabola
{
  public Parabola(double c_0, double c_1, double c_2) throws NotParabolaException
  {
```

```
    super();
    if (c_2 == 0)
      throw new NotParabolaException();
    this.c_0 = c_0;
    this.c_1 = c_1;
    this.c_2 = c_2;
  }

  public double getValue(double x)
  {
    return c_0 + c_1*x + c_2*x*x;
  }

  /**
   *
   * @return set of roots (either 2, 1 or zero length array)
   */
  public double[] getRoots()
  {
    double d = c_1*c_1 - 4*c_2*c_0;
    double[] roots;
    if (d < 0)
      roots = new double[0];
    else if (d == 0)
    {
      double tp = getTurningPoint();
      roots = new double[1];
      roots[0] = tp;
    }
    else
    {
      roots = new double[2];
      roots[0] = (-c_1 - Math.sqrt(d))/(2*c_2);
      roots[1] = (-c_1 + Math.sqrt(d))/(2*c_2);
    }
    return roots;
  }

  public double getTurningPoint()
  {
    return -c_1/(2*c_2);
  }

  private double c_0, c_1, c_2;
}
```

2.2 The unit test

The Unit Test includes

- a `@Before` method setting up test data covering points from different equivalence partitions as well as some boundary values,
- a `@Test(expected=NotAParabolaException.class)` used to verify that the constructor raises the appropriate exception if trying to create non-parabola,
- a `@Test validateThatFunctionValueAtRootsZero()` method verifying that the function value at roots is indeed zero, and
- a `@Test` testRootValues() method which checks whether the roots are correct for some hard-coded root values.

```
package za.co.solms.math;
```

```
import java.util.LinkedList;
import java.util.List;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import za.co.solms.math.NotParabolaException;
import za.co.solms.math.Parabola;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;


public class ParabolaTest
{
  @Before
  public void preTestInitialization()
  {
    try
    {
      testData = new LinkedList<ParabolaTest.TestParabola>();
      testData.add(new TestParabola("normal2PositiveRoots", new Parabola(1, -3, 2), new double[] {1,2}));
      testData.add(new TestParabola("normalPositiveNegativeRoot", new Parabola(1, 1, -2), new double[]
          {1,-2}));
      testData.add(new TestParabola("invertedNegativeRoots", new Parabola(-1, -1, 2), new double[] {1,2}));
      testData.add(new TestParabola("singleRealRoot", new Parabola(1, -2, 1), new double[] {1}));
      testData.add(new TestParabola("noRealRoots", new Parabola(2, 1, 2), new double[] {}));
    }
    catch (NotParabolaException e)
    {
      fail("Constructor claimed not parabola, but is");
    }
  }

  @After
  public void PostTestCleanup()
  {
    testData.clear();
  }


  @Test(expected = NotParabolaException.class)
  public void notParabolaTest() throws NotParabolaException
  {
    new Parabola(2,1,0);
  }

  @Test
  public void validateThatFunctionValueAtRootsZero()
  {
    for (TestParabola testParabola: testData)
    {
      Parabola p = testParabola.getParabola();
      double[] roots = p.getRoots();
      for (double root:roots)
        assertEquals("function value at roots for " + testParabola.getIdentifier()
          + " not zero.", 0, p.getValue(root), eps);
    }
  }

  @Test
  public void testRootValues()
  {
    for (TestParabola testParabola: testData)
    {
      Parabola p = testParabola.getParabola();
      double[] roots = testParabola.getRoots();
      assertEquals (testParabola.getRoots().length, roots.length);
      for (double root:roots)
        assertEquals("Roots for " + testParabola.getIdentifier(),
            true, contains(testParabola.getRoots(), root, eps));
    }
```

```
    }

    private static boolean contains(double[] array, double value, double eps)
    {
      for (double v : array)
        if (Math.abs(v-value) < eps)
          return true;

      return false;
    }

    private static class TestParabola
    {

      public TestParabola(String identifier, Parabola parabola, double[] roots) {
      super();
      this.identifier = identifier;
      this.parabola = parabola;
      this.roots = roots;
    }

    public String getIdentifier() {
      return identifier;
    }
    public void setIdentifier(String identifier) {
      this.identifier = identifier;
    }
    public Parabola getParabola() {
      return parabola;
    }
    public void setParabola(Parabola parabola) {
      this.parabola = parabola;
    }
    public double[] getRoots() {
      return roots;
    }
    public void setRoots(double[] roots) {
      this.roots = roots;
    }

    private String identifier;
      private Parabola parabola;
      private double[] roots;
    }


    private double eps = 1e-7;
    private List<TestParabola> testData;
  }
```

3 JUnit

JUnit is a very widely used open-source unit testing framework. It introduces annotations to declare test methods as well as initialization and clean-up methods which are executed either before and after each test method or on test class instantiation and garbage collection.

In addition JUnit introduces a range of assertion methods which are used to verify the state of objects and variables.

3.1 JUnit setup annotations

one of the principles of unit testing is that tests should be independent (run in any order). For this it is required that clean test data is provided for each test, i.e. test data which was not modified by the previous test. To make this easier, JUnit introduces initialization methods which are any methods annotated with `@Before` and clean-up methods which are any methods annotated as

`@After`. Before executing any test method, all methods annotated with `@Before` are executed (in a non-determined order). Then the test method is executed and then all methods annotated with `@After`.

Sometimes one needs to perform a once-off setup of the test environment (typically external dependencies like that of bringing up an embedded application server or an embedded database) which needs to be torn down (cleaned up) after all tests for that class have been executed. This is done by annotating setup and tear-down methods annotated respectively with `@BeforeClass` and `@AfterClass` annotations.

3.2 JUnit @Test annotations

Any `@Test`method will be executed by `JUnitRunner` as a separate test of the subject class. One can specify that the method body must result in a particular exception by annotating a test method with

```
@Test(expected=SomeException.class)
public void someTestMethod()
{
...
}
```

Such a test will fail if `SomeException` is not raised. The full set of unit tests for a system is typically executed on system builds.

To protect the build from being held up by infinite loops, deadlocks and so on, one can used

```
\texttt{@Test(timeout=500)}
```

The test will fail if it is not completed within the timeout period. Note that it is not recommended to use the `timeout` parameter for performance testing. Unit tests are typically executed on a variety of platforms including individual developer machines which may vary considerably and which may fail unnecessarily. Furthermore, non-functional testing should be kept separate from unit (functional) testing.

Finally, in development one may, at times, want to temporarily disable a test. This could be so because certain functionality is not yet developed or because there is some dispute around the requirements. One can disable a unit via an `@Ignore` annotation:

```
@Ignore{``Temporarily disabled because dispute around requirements''}
@Test
public void testSomeDisputedRequirement()
{
...
}
```

4 Mocking

There are many mocking frameworks available for Java and also for other programming languages. Generally they require a mock object per unit test and not a single mock object per contract. The result is that one typically develops or generate many mock classes for the same contract. There is no guarantee that any of these realize the contract for the lower level component they are mocking. Also, should a contract change, one needs to find across unit tests all mock classes of all classes implementing that contract.

Here we introduce a contracts-based approach within which one writes a single mock class for each contract. These mock classes are then reused across all unit tests which require a service provider realizing that contract. Furthermore, one can test the mock object against its contract covering the test scenarios supported by mock class. These are specified either by different mock states or by inputs satisfying specific constraints. These contract-specific mock classes are then maintained with the contract.

4.1 A contracts-based mocking framework

The contracts-based framework for mocking

- introduces the concept of a mock class as a `@Mocking annotation`,
- defines a contract for mock objects in the form of a `Mock` interface,
- provides a convenience base class for mock classes which applies the `@Mocking` annotation and provides the infrastructure for state management and call logging, and
- a call logging aspect which is woven into all methods of all classes annotated as `@Mocking`.

4.1.1 The Mocking annotation

```
package za.co.magnabc.services.mocking;

import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.interceptor.InterceptorBinding;

import java.lang.annotation.ElementType;

/**
 * An annotation which is assigned to any mock object to designate that it is mocking another object.
 * This annotation is, for example, used to designate that the object must be intercepted by a call logging
 * interceptor.
 *
 * @author fritz@solms.co.za
 *
 */
@Inherited
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface Mocking {}
```

4.1.2 The mock interface

```
package za.co.magnabc.services.mocking;

import org.springframework.stereotype.Service;

/**
 * Specifies that any Mock object must be able to provide the call logger
 * which is used to log the calls made to it in order to query the
 * call history to assert that pre-conditions and post-conditions of a
 * contract have been addressed.
 *
 * @author fritz@solms.co.za
 *
 */
```

```
@Mocking
@Service
public interface Mock
{
  /**
   * This method returns the call logger (if any) which was assigned to the
   * mock object. It is used by the {@link CallLoggingAspect} to log call descriptions.
   *
   * @return the call logger used by the mock object.
   */
  public CallLogger getCallLogger();

  /**
   * Method to set the state of a mock object to some specific state in which it
   * behaves in some particular way.
   *
   * @param state the state identifier
   * @throws InvalidStateException if the mock object does not support the specified state
   */
  public void setState(State state) throws InvalidStateException;

  /**
   *
   * @return the current state of the mock object
   */
  public State getState();

  public interface State{};

  public class InvalidStateException extends Exception {};
}
```

4.1.3 A convenience base class for mock classes

```
package za.co.magnabc.services.mocking;

/**
 * A minimal base class for mocking objects which ensures that the call logging aspect is applied to the logging
      object
 * and which can provide the call logger which was assigned to the mock it.
 *
 * @author fritz@solms.co.za
 */
@Mocking
public class BaseMock implements Mock
{

  public BaseMock()
  {
    callLogger = new SimpleCallLogger(this);
  }

  @Override
  public CallLogger getCallLogger()
  {
    return callLogger;
  }

  public State getState() {
    return state;
  }

  public void setState(State state) {
    this.state = state;
  }

  private CallLogger callLogger;
  private State state;

}
```

4.1.4 A call logging aspect

```
package za.co.magnabc.services.mocking;

import java.lang.reflect.Method;
import java.time.LocalDateTime;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.reflect.MethodSignature;

import za.co.magnabc.services.messaging.Request;

/**
 * Logging aspect for Mock objects which intercepts all mock object calls and
 * adds the call events to the call history of the mock object.
 *
 * @author fritz@solms.co.za
 *
 */
@Aspect
public class CallLoggingAspect
{
  /**
   * Logs all service calls, but not other method calls.
   * @param joinPoint
   * @return
   * @throws Throwable
   */
  @Around(value = "@within(za.co.magnabc.services.mocking.Mocking)"
      + "|| @annotation(za.co.magnabc.services.mocking.Mocking)")
  public Object logCall(ProceedingJoinPoint joinPoint) throws Throwable
  {
    LocalDateTime requestTime = LocalDateTime.now();

    Method method = ((MethodSignature)joinPoint.getStaticPart().getSignature()).getMethod();
    Request request = null;
    try
    {
      request = (Request)joinPoint.getArgs()[0];
    }
    catch (Exception e){ /* Ignore all non-service requests, i.e. methods which do not receive a single service
          request */ }

    Object response = null;
    LocalDateTime responseTime;

    try
    {
      response = joinPoint.proceed();
      responseTime = LocalDateTime.now();

      if (request != null)
        ((Mock)joinPoint.getTarget()).getCallLogger().logCall(
          new CallDescriptor(method, requestTime, request, response, responseTime));
      return response;
    }
    catch (Exception e)
    {
      responseTime = LocalDateTime.now();
      response = e;
      ((Mock)joinPoint.getTarget()).getCallLogger().logCall(
          new CallDescriptor(method, requestTime, request, response, responseTime));
      throw e;
    }
    }
}
```

5 A simple OrderPricer example with mocking

In this example we first specify service-oriented, contracts based requirements which are decomposed across levels of granularity. the contracts are mapped onto Java and for each contract a mock object is developed which can be used across unit tests of services which require this lower level component.

A unit test is then developed against the contract where we mock out lower level service providers using mock objects developed against contracts, and mock states to change behaviour determined by mock's environment.

5.1 Requirements

Here we follow the URDAD approach of specifying the requirements in the form of services contracts and decomposing the requirements across levels of granularity. A requirements unit is represented by an interface encapsulating methods whose requirements are specified using pre- and post-conditions as well as the request and result data structure.

The requirements are decomposed across levels of granularity, allocating parts of the responsibilities to lower level services. The detailed requirements around these responsibility components are factored out into the requirements around these lower level services. The decomposition is documented through a functional (service) requirements diagram and the process specification specifies the orchestration of the higher level service from these lower level services.

5.1.1 Functional (service) requirements

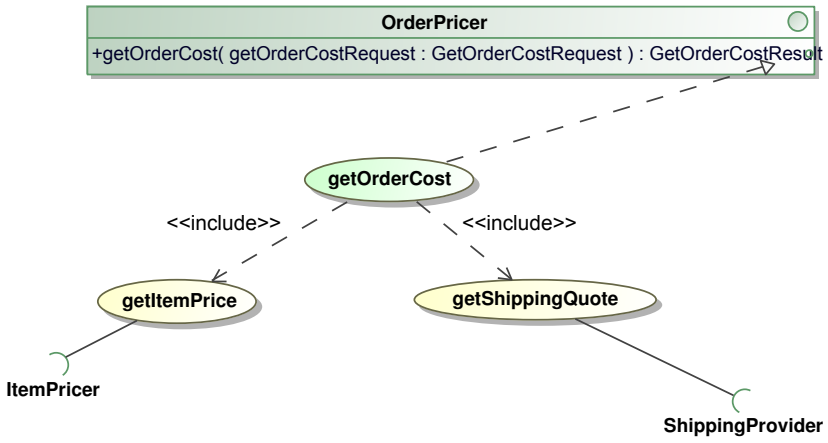

Figure 2:

The functional or service requirements specify the decomposition of the required functionality into lower level services. These lower level services are then allocated to responsibility domains with a contract (interface) per responsibility domain.

5.1.2 Service contract

The service contract specifies requirements at a particular level of granularity. It includes

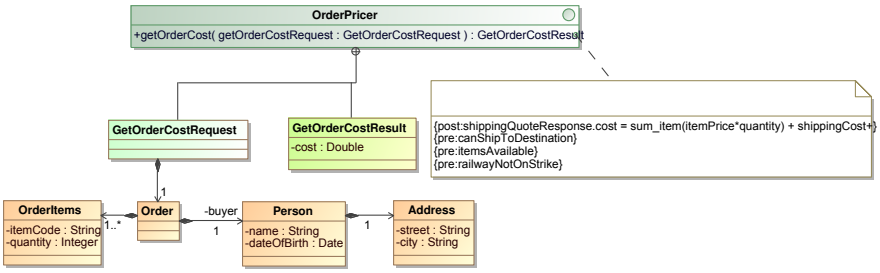

Figure 3: A UML representation of a services contract.

1. The service to be provided.
2. The *pre-conditions* of the service, i.e. the conditions under which the service may be refused without the refusal being a breach of contract/error.
3. the *post-conditions* for the service which must hold true once the service has been provided
4. the *data structure requirements* for the request and result objects potentially with OCL invariance constraints which would be mapped onto Java bean-validation constraints.

5.1.3 Process specification

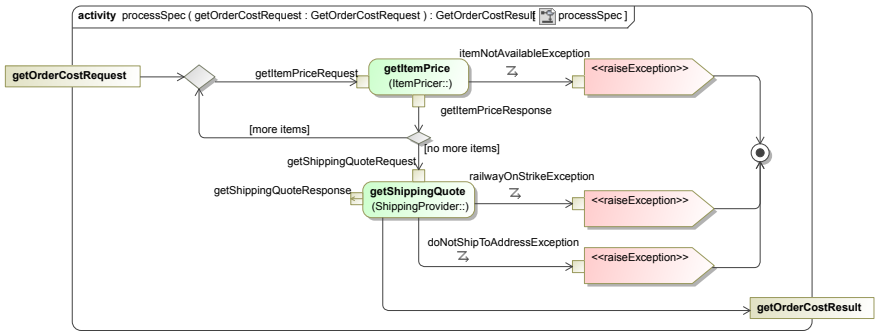

Figure 4:

The process specification specifies how the higher level service is to be orchestrated across lower level services. Note that if a precondition is not met, the associated exception is raised and the service is not provided.

5.2 Java implementation mapping

One maps each contract onto a separate interface and develops a mock object for the contract/interface. The unit test is developed such that the component under test is isolated by mocking out any dependencies.

One can then develop classes realizinging the contract (implementing the interface and meeting the requirements of the contract). Any implementation class is tested against all unit tests developed for all the contracts it realizes, i.e. each interface it implements.

5.2.1 Contracts as interfaces

Each contract is mapped onto a separate Java interface. Furthermore, we choose to package all contract components within the contract (interface). This includes the request and result classes as well as the exception classes associated with the different pre-conditions of a contract.

```
package za.co.magnabc.services.mocking.example.shipping;

import javax.validation.constraints.NotNull;
import za.co.magnabc.services.messaging.PreconditionViolation;
import za.co.magnabc.services.messaging.Request;
import za.co.magnabc.services.messaging.Response;
import za.co.magnabc.services.mocking.example.Address;
import za.co.magnabc.services.validation.RequestNotValidException;

public interface ShippingQuoteProvider
{

  GetQuoteResponse getQuote(GetQuoteRequest request) throws RequestNotValidException,
       DoNotShipToAddressException,
    RailwayOnStrikeException;

  class GetQuoteRequest implements Request
  {

    public GetQuoteRequest(Address address) {
      super();
      this.address = address;
    }

    public Address getAddress() {
      return address;
    }

    public void setAddress(Address address) {
      this.address = address;
    }

    @NotNull
    private Address address;
  }

  class GetQuoteResponse implements Response
  {

    public GetQuoteResponse(double price)
    {
      super();
      this.price = price;
    }

    public double getPrice() {
      return price;
    }

    public void setPrice(double price) {
      this.price = price;
    }

    private double price;

  }

  class RailwayOnStrikeException extends PreconditionViolation
```

```
  {

    public RailwayOnStrikeException(){}

    public RailwayOnStrikeException(String msg) {super(msg);}

    public RailwayOnStrikeException(Throwable cause) {super(cause);}

    public RailwayOnStrikeException(String msg, Throwable cause) {super(msg,cause);}

  }

  class DoNotShipToAddressException extends PreconditionViolation
  {

    public DoNotShipToAddressException() {super();}

    public DoNotShipToAddressException(String message, Throwable cause)
    {
      super(message, cause);
    }

    public DoNotShipToAddressException(String message) {super(message);}

    public DoNotShipToAddressException(Throwable cause) {super(cause);}

  }
}
```

5.2.2 Mock objects

When developing a mock object for a contract/interface, one either sub-classes a `BaseMock`, or implements the `Mock` interface and annotates the mock class as `@Mocking`.

In either case one defines different states for different behaviour due to different states of environment and one provides documented behaviour (input/output behaviour) for different equivalence partitions and different boundary/extreme values.

```
package za.co.magnabc.services.mocking.example.shipping;

import javax.ejb.Stateless;
import javax.inject.Inject;
import org.springframework.stereotype.Service;
import za.co.magnabc.services.mocking.BaseMock;
import za.co.magnabc.services.mocking.Mock;
import za.co.magnabc.services.validation.RequestNotValidException;
import za.co.magnabc.services.validation.beanvalidation.ServiceValidationUtilities;

/**
 * A mock ShippingQuoteProvider which
 * <ul>
 * <li>throws {@link RailwayOnStrikeException} if set in a railwayOnStrike state, </li>
 * <li>throws {@link DoNotShipToAddressException} if the shipping city is Timbuktu,</li>
 * <li>returns a zero quote if the city is Pofadder, and</li>
 * <li>otherwise returns a fixed cost of 888.88.</li>
 * </ul>
 *
 * @author fritz@solms.co.za
 *
 */
@Stateless
@Service
public class ShippingQuoteProviderMock extends BaseMock implements ShippingQuoteProvider
{
  public ShippingQuoteProviderMock()
    {setState(State.externalRequirementsMet);}

  public GetQuoteResponse getQuote(GetQuoteRequest request)
      throws RequestNotValidException, DoNotShipToAddressException, RailwayOnStrikeException
```

```
    {
        // Check pre-condition: Request must be valid.
        serviceValidationUtilities.validateRequest(GetQuoteRequest.class, request);

      if (getState() == State.railwayOnStrike)
        throw new RailwayOnStrikeException();

      if (request.getAddress().getCity().trim().toLowerCase().equals("timbuktu"))
        throw new DoNotShipToAddressException();

      if (request.getAddress().getCity().trim().toLowerCase().equals("pofadder"))
        return new GetQuoteResponse(0);
      else
        return new GetQuoteResponse(888.88);

    }

    public enum State implements Mock.State{externalRequirementsMet, railwayOnStrike}

      @Inject
      private ServiceValidationUtilities serviceValidationUtilities;

}
```

5.2.3 Spring unit test configuration

The Spring unit test configuration is done by a configuration class which is annotated as a Spring configuration and also as requiring the Spring AspectJ AutoProxy so that call the logging aspect is applied to all methods of all classes annotated with `@Mocking`.

The configuration configures the dependency injection for mocking beans, the call logging aspect, validation beans and service utility beans.

```
package za.co.magnabc.services.mocking.example.retail;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.validation.beanvalidation.LocalValidatorFactoryBean;
import za.co.magnabc.services.ServiceUtilities;
import za.co.magnabc.services.ServiceUtilitiesBean;
import za.co.magnabc.services.mocking.CallLoggingAspect;
import za.co.magnabc.services.mocking.example.shipping.ShippingQuoteProvider;
import za.co.magnabc.services.mocking.example.shipping.ShippingQuoteProviderMock;
import za.co.magnabc.services.validation.beanvalidation.ServiceValidationUtilities;
import za.co.magnabc.validation.beanvalidation.BeanValidation;
import za.co.magnabc.validation.beanvalidation.BeanValidationBean;

@Configuration
@EnableAspectJAutoProxy(proxyTargetClass=true)
public class OrderPricerTestConfiguration
{

    @Bean
    public CallLoggingAspect callLoggingAspect()
    {
        return new CallLoggingAspect();
    }

    @Bean
    public OrderPricer orderPricer()
    {
        logger.trace("Configuring order pricer bean.");
        return new OrderPricerBean();
    }

    @Bean
```

```
    public ItemPricer itemPricerMock()
    {
        logger.trace("Configuring mock item pricer.");
        return new ItemPricerMock();
    }

    @Bean
    public ShippingQuoteProvider shippingQuoteProviderMock()
    {
        logger.trace("Configuring mock shipping quote provider.");
        return new ShippingQuoteProviderMock();
    }

    @Bean
    public javax.validation.Validator validator() {
        return new LocalValidatorFactoryBean();
    }

    @Bean
    public BeanValidation.BeanValidationLocal beanValidation() {
        return new BeanValidationBean();
    }

    @Bean
    public ServiceValidationUtilities serviceValidationUtilities() {
        return new ServiceValidationUtilities();
    }

    @Bean
    public ServiceUtilities serviceUtilities() {
        return new ServiceUtilitiesBean();
    }

    private static final Logger logger = (Logger)LoggerFactory.getLogger(OrderPricerTestConfiguration.class);

}
```

5.2.4 Spring unit test

Use annotations to specify that `JUnit4Runner` is to be used to execute the unit test and to specify which the test configuration class to use.

Different test methods annotated as `@Test`s. Have test methods which set the state of different mock objects to different states in order to test different environmental scenarios. Use input from different equivalence partitions and boundary values. Use assertions to check results and specify when and what exceptions expected.

We also need to specify which lower level services (currently mocked) must have been called to either check that certain pre-conditions are met or to ensure that certain post-conditions are addressed (or both).

```
package za.co.magnabc.services.mocking.example.retail;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import java.time.LocalDate;
import java.time.Month;
import java.util.Map;
import java.util.TreeMap;
import javax.inject.Inject;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import za.co.magnabc.services.ServiceUtilities.NotAServiceException;
import za.co.magnabc.services.mocking.example.Address;
import za.co.magnabc.services.mocking.example.Person;
```

```
import za.co.magnabc.services.mocking.example.shipping.ShippingQuoteProvider;
import za.co.magnabc.services.mocking.example.shipping.ShippingQuoteProviderMock;
import za.co.magnabc.services.validation.RequestNotValidException;

/**
 * A unit test which is used
 * <ol>
 * <li> as a test for the services-oriented mocking framework </li>
 * <li> to provide a reference implementation for using unit testing with
 *  services-oriented mocking </li>
 * </ol>
 *
 * @author fritz@solms.co.za
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = {OrderPricerTestConfiguration.class})
public class OrderPricerTest
{

  /**
   * Unit test demonstrating normal execution with normal assertions as well as
   * using the mock-object's call log to assert that the required calls were made to address the
   * pre- and post-conditions for the service

   */
  @Test
  public void everythingKosherTest()
  {
    Order order = new Order(createNormalPerson(), createDefaultOrderItems());

    // Set mock objects into desired states:
    shippingQuoteProviderMock.setState(ShippingQuoteProviderMock.State.externalRequirementsMet);
    shippingQuoteProviderMock.getCallLogger().clearLog();
    itemPricerMock.getCallLogger().clearLog();

    try
    {
      double cost = orderPricer.getOrderCost(new OrderPricer.GetOrderCostRequest(order)).getCost();

      // Check return value state
      assertEquals(1488.82, cost, 1e-6);

      // Use the mock object's call logger populated by the call logging aspect to
      // check calls used to address post-conditions made.
      assertEquals(1,shippingQuoteProviderMock.getCallLogger().getInvocationCount("getQuote"));
      assertEquals(2,itemPricerMock.getCallLogger().getInvocationCount("getItemPrice"));
    }
    catch (Exception e)
    {
      e.printStackTrace();
      fail("Exception " + e.getClass().getName() + " thrown, but should not have been thrown.");
    }

    try
    {
      shippingQuoteProviderMock.getCallLogger().getInvocationCount("nonExistingService");
      fail("Found non-existing service");
    }
    catch (NotAServiceException e)
    { /* expected behaviour */ }
  }

  /**
   * This test demonstrates the state management for mock objects. We are setting the
   * mock object in a state which is independent of the input. In this case this is a state
   * where the service provider refuses/cannot provide the service;
   *
   * @throws Exception
   */
  @Test(expected = ShippingQuoteProvider.RailwayOnStrikeException.class)
  public void railwayOnStrikeTest() throws Exception
  {
```

```
    shippingQuoteProviderMock.setState(ShippingQuoteProviderMock.State.railwayOnStrike);

    orderPricer.getOrderCost(new OrderPricer.GetOrderCostRequest
        (new Order(createNormalPerson(), createDefaultOrderItems())));
  }

  /**
   * Sets mock object into normal state and checks that exception associated with
   * input-related pre-condition check raised.
   *
   * @throws Exception
   */
  @Test(expected = ShippingQuoteProvider.DoNotShipToAddressException.class)
  public void testUnshippableAddress() throws Exception
  {
    shippingQuoteProviderMock.setState(ShippingQuoteProviderMock.State.externalRequirementsMet);

    orderPricer.getOrderCost(new OrderPricer.GetOrderCostRequest
        (new Order(createTimbuktuen(), createDefaultOrderItems())));
  }

  /**
   * Test a boundary value/extreme point for the input data -- that of an empty order.
   *
   * @throws Exception
   */
  @Test
  public void emptyOrderTest() throws Exception
  {
    shippingQuoteProviderMock.setState(ShippingQuoteProviderMock.State.externalRequirementsMet);

    assertEquals(888.88, orderPricer.getOrderCost(new OrderPricer.GetOrderCostRequest
        (new Order(createNormalPerson(), new TreeMap<String, Integer> ()))).getCost(),1e-7);
  }

  @Test(expected=RequestNotValidException.class)
  public void nullOrderTest() throws Exception
  {
    shippingQuoteProviderMock.setState(ShippingQuoteProviderMock.State.externalRequirementsMet);

    orderPricer.getOrderCost(new OrderPricer.GetOrderCostRequest(null));
  }

  private static Person createNormalPerson()
  {
    Address address = new Address("Road to Nowhere", "Lusikisiki");
    return new Person("Abraham", LocalDate.of(1985, Month.APRIL, 22), address);
  }

  private static Person createTimbuktuen()
  {
    Address address = new Address("111 Library Road", "Timbuktu");
    return new Person("Fazil", LocalDate.of(1908, Month.APRIL, 1), address);
  }


  private static Map<String, Integer> createDefaultOrderItems()
  {
    Map<String, Integer> orderItems = new TreeMap<String, Integer>();
    orderItems.put("jam", 5);
    orderItems.put("rooibosTea", 1);
    return orderItems;
  }

  @Inject
  private ShippingQuoteProviderMock shippingQuoteProviderMock;
  @Inject
  private ItemPricerMock itemPricerMock;
  @Inject
  private OrderPricer orderPricer;
}
```

5.2.5 An implementation class

The implementation class is, of course, simply a class realizing its contract(s). It is tested against all unit tests for all the contracts it claims to realize.

```
package za.co.magnabc.services.mocking.example.retail;

import java.util.Map;
import javax.ejb.Stateless;
import javax.inject.Inject;
import org.springframework.stereotype.Service;

import za.co.magnabc.services.mocking.example.Address;
import za.co.magnabc.services.mocking.example.shipping.ShippingQuoteProvider;
import za.co.magnabc.services.validation.RequestNotValidException;
import za.co.magnabc.services.validation.beanvalidation.ServiceValidationUtilities;

@Stateless
@Service
public class OrderPricerBean implements OrderPricer
{

  @Override
  public GetOrderCostResponse getOrderCost(GetOrderCostRequest request) throws RequestNotValidException,
        ShippingQuoteProvider.DoNotShipToAddressException, ItemPricer.ItemNotAvailableException,
             ShippingQuoteProvider.RailwayOnStrikeException
  {
        // Check pre-condition: Request must be valid.
        serviceValidationUtilities.validateRequest(GetOrderCostRequest.class, request);

        Map<String,Integer> orderItems = request.getOrder().getOrderItems();
        double total = 0;

        for (String itemCode : orderItems.keySet())
        {
          ItemPricer.GetItemPriceRequest req = new ItemPricer.GetItemPriceRequest(request.getOrder().getBuyer
                ().
                itemCode);
          total += itemPricer.getItemPrice(req).getPrice() * orderItems.get(itemCode);
        }

        Address shippingAddress = request.getOrder().getBuyer().getAddress();
      total += shippingQuoteProvider.getQuote(new ShippingQuoteProvider.GetQuoteRequest(shippingAddress)).
            getPrice();

    return new GetOrderCostResponse(total);
  }

    @Inject
    private ServiceValidationUtilities serviceValidationUtilities;
    @Inject
    private ItemPricer itemPricer;
    @Inject
    private ShippingQuoteProvider shippingQuoteProvider;

}
```