Frameworks and Testing

Stefan Roock

Apcon Workplace Solutions &
University of Hamburg
Vogt-Kölln-Str. 30
22527 Hamburg, Germany
+49 40 42883 2302
roock@jwam.de

Abstract

Testing is one of the main XP practices (cf. [1]) and becomes more and more understood for application development. But when it comes to frameworks, testing is more manifold. We have experienced a lot of difficulties and some solutions in this area based on the last three years of developing the JWAM framework (cf. [4], [5]). This paper presents four categories of tests relevant for framework development. Design and construction guidelines for the test categories are given.

Keywords

Testing, Framework, Acceptance Test, Test Framework, Test Center

INTRODUCTION

Testing becomes more and more understood for application development. But when it comes to frameworks, testing is more manifold. A number of questions arise:

- How can one do acceptance testing for frameworks? What what aspects differ framework acceptance tests from framework unit tests?
- The interaction between an application and the used framework is often hard to test. What support is neccessary to test this interaction?
- How can one test the application's conformance with the framework?

During the development of the JWAM framework (cf. [4]) with XP we discovered that it is useful to have four test categories. *Unit Tests* cover the implementation of the framework and differ from application unit tests in few aspects.

Acceptance Tests are well known from application development. Transferring the concept of acceptance criteria from applications to frameworks isn't trivial. If we assume a framework product manager who specifies the acceptance criteria for the framework, isn't he specifying the whole framework design already?

Often frameworks hinder testing of applications based on the framework since necessary framework APIs aren't available or difficult to use. Beneath the required framework API the framework should provide a *Test Framework* to support application testing. The test framework provides a set of utility classes and operations to ease testing application components highly integrated with the framework.

Application classes based on the framework have to follow certain protocols. A *Test Center* bundled with the framework supports testing the protocol of application classes derived from the framework. The test categories are shown in the following table.

Test Category	**Main Purpose**
Unit Test	Framework developers check their code
Acceptance Test	Framework Product Managers specify acceptance criteria for the framework
Test Framework	Support for application developers when writing unit and acceptance tests for applications.
Test Center	Test conformance of application classes with the framework

Since unit tests for frameworks differ from unit tests for applications in only a few minor aspects I will not discuss them in this paper.

ACCEPTANCE TEST

Framework acceptance tests define the acceptance criteria for framework functionality. At a first glance it seems impossible and perhaps unnecessary to separate acceptance from unit tests. But if the two test categories aren't separated from each other, important design decision are very difficult to make. In this case the framework developers have only soft factors to decide which functionality of the framework may be changed or even deleted without affecting the acceptance criteria. They have to have detailed knowledge about the applications based on the framework. Only then they may guess which functionality may be refactored or deleted without violating the acceptance criteria for the framework.

Let's think about a very simple framework: A desktop based tool works on a set of business objects (cf. Fig. 1). Complex tools may be composed from simpler ones.

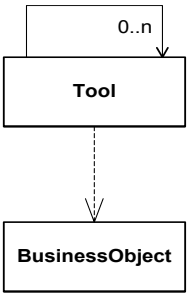

Fig. 1: Sample Framework

The acceptance tests for this tiny framework should not define the exact classes and methods of the framework. The definition of the concrete classes and methods is the task of the framework developers. Acceptance tests should only define the functionality of the framework. Therefore the acceptance tests have to be formulated in an abstract way. Hence the acceptance tests are programmed with imaginary framework classes. The user story for the framework may be formulated in few sentences:

"*Every business object has a name which may change. A tool may be equipped with different business objects during its lifetime. A complex tool may be composed from simple tools.*"

Fig. 2 shows an according code fragment for an acceptance test (a JUnit type of test support is assumed, cf. [3]).

```
public class FrameworkAcceptanceTest {

  public void testBusinessObject() {
    DummyBO bo = new DummyBO("MyBO");
    assertEquals("MyBO", bo.getName();
    bo.rename("NewName");
    assertEquals("NewName", bo.getName();
  }

 public void testSimpleTool() {
  DummyBO bo1 = new DummyBO("BO1");
  DummyBO bo2 = new DummyBO("BO2");
  DummyTool tool = new DummyTool();
  assertFalse(tool.hasBusinessObject());
  tool.equip(bo1);
  assertEquals(bo1,
   tool.getBusinessObject());
  tool.equip(bo2);
  assertEquals(bo2,
   tool.getBusinessObject());
 }

  public void testComplexTool() {
    DummyBO bo1 = new DummyBO("BO1");
    DummyBO bo2 = new DummyBO("BO2");
    DummyComplexTool tool =
      new DummyComplexTool();
    assertFalse(tool.hasBusinessObject());
    assertEquals(1, tool.getSubTools().
      length);
    assertEquals(tool.getSubTools()[0]
      instanceof DummyTool);
    tool.equip(bo1);
    assertEquals(bo1,
      tool.getBusinessObject());
    assertEquals(bo1,
tool.getSubTools()[0].
      getBusinessObject());
  }
}
```

Fig. 2: Acceptance Test

The *DummyBO, DummyTool* and *ComplexDummyTool* classes are programmed by the acceptance test author with dummy operations. Acceptance tests have to be compilable but needn't succeed with the dummy classes.

After the acceptance tests are defined the framework developers have the task to make the acceptance test succeed without modifying the test itself. For the given example the following code makes the tests succeed (cf. Fig. 3).

```
public class BusinessObject {

  public void rename(String name) {
    _name = name;
  }

  public String getName() {
    return _name;
  }

  public boolean equals(Object o) {
    boolean eq =
      o instanceof BusinessObject;

    if (eq) {
      BusinessObject bo =
        (BusinessObject) o;
      eq = bo.getName().equals( getName()
);
    }

    return eq;
  }

  private String _name;
}


public class Tool {

  public void equip(BusinessObject bo) {
    _bo = bo;
    Iterator iter = _subTools.iterator();

    while (iter.hasNext()) {
      Tool t = (Tool) iter.next();
      t.equip(bo);
    }
  }

  public boolean hasBusinessObject() {
    return _bo != null;
  }

  public BusinessObject getBusinessOb-
ject(){
    return _bo;
  }

  protected void setParent(Tool t) {
    _parent = t;
  }

  protected void addSubTool(Tool t) {
```

```
    _subTools.add(t);
    t.setParent(this);
  }

  private BusinessObject _bo;

  private Tool _parent;

  private Collection _subTools =
    new ArrayList();
}


public class DummyBO extends BusinessOb-
ject
{}

public class DummyTool extends Tool
{}

public class DummyComplexTool extends Tool
{
  public DummyComplexTool() {
    addSubTool(new DummyTool());
  }
}
```

Fig. 3: Framework Implementation

In this case the dummy classes are mainly empty since the imagined method names match the method names choosen by the framework developers. This needn‘t be the case. Then the adapter pattern (cf. [2]) may be used. In the example the framework developers have choosen to add the method *addSubTool* for tool composition.

TEST FRAMEWORK

Often frameworks hinder testing of applications based on them. To support application testing frameworks should provide a test framework. Often this test framework can be realized as subclasses of TestCase (in the case of JUnit). These subclasses often provide additional convenience methods specialized for the framework. In the case of our example framework testing the combination of tools can be supported. Let’s assume that subtools use the observer pattern (cf. [2]) to notify their parent tool about state changes. Then a special *TestCase* superclass may look like Fig. 4.

```
public class ToolTestCase extends TestCase
  implements Observer {

  public void notify() {
    _notifications++;
  }

  protected void resetNotifications() {
    _notifications = 0;
  }

  protected void getNotificationCount() {
    return _notifications;
  }

  protected void setUp() {
    super.setUp();
    resetNotifications();
  }

  private int _notifications;
}
```

```
public interface Observer {
  public void notify();
}
```

Fig. 4: ToolTestCase

Concrete tool test cases now can be developed in a more convenient way. Let’s assume a simple management tool for visiting cards. The *VisitingCardManager* has the subtool *VisitingCardEditor*. While the *VisitingCardManager* works on a *VisitingCardBox* the *VisitingCardEditor* uses a *VisitingCard* (see Fig. 5).

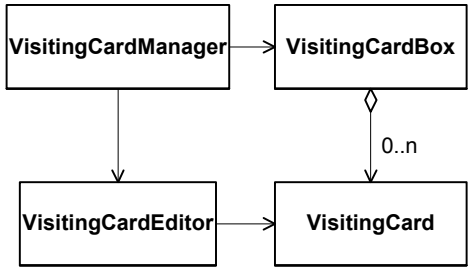

Fig. 5: Example tool

When the *VisitingCardEditor* stores a changed *VisitingCard* in the *VisitingCardBox* the *VisitingCardEditor* notifies the *VisitingCardManager*. Then the *VisitingCardManager* redraws the list with the visiting cards in the *VisitingCardBox*. Therefore the *VisitingCardEditor* only works correcly if it notifies its parent tool when its state changes (cf. Fig. 6).

```
public class VisitingCardEditorTest
 extends ToolTestCase {

  public void testTool() {
    VisitingCardEditor editor =
      new VisitingCardEditor();
    editor.setParent(this);
    editor.storeCard();
    assertEquals(1, getNotification-
Count());
  }
}
```

Fig. 6: Concrete tool test case

TEST CENTER

The test center is delivered together with the framework. It helps application developers to check the consistency of framework based application classes with the framework. In the case of the above example the consistency of the *VisitingCardManager* is determined by the *ToolTestCenter* (see Fig. 7).

```
public class ToolTestCenter extends Assert
{

  public void test(ToolTestContext con-
text){
    Tool t = context.getNewTool();

t.equip(context.getNewBusinessObject());
    assertTrue(t.hasBusinessObject());
  }
}
```

```
public interface ToolTestContext {
  public Tool getNewTool();
  public BusinessObject
    getNewBusinessObject();
}
```

Fig. 7: Test center

The ToolTestContext provides the objects used by the test center. Often it is tedious or even impossible to supply all needed objects as parameters up front.

The test case of the *VisitingCardManager* is then extended by a call to the test center (cf. Fig. 8).

```
public class VisitingCardManagerTest
  extends ToolTestCase
  implements ToolTestContext {
  public void testManager {
    VisitingCardManager manager =
      new VisitingCardManager();
    new ToolTestCenter().test(manager);
  }
  public Tool getNewTool() {
    return new VisitingCardManager();
  }
  public BusinessObject
    getNewBusinessObject() {
    return new VisitingCardBox();
  }
}
```

Fig. 8: Usage of the test center

TOOL SUPPORT

All described test categories can be easily developed with JUnit or a similar test framework and test tool. It is convenient to extend JUnit with the concept of test categories to separate at least unit from acceptance tests. This is relatively easy by extending the class *TestCase* (see Fig. 9)

```
public class XTestCase extends TestCase {
 protected void setCategory (String cat) {
  _category = cat;
 }
 protected String getCategory () {
  return _category;
 }
 public String toString() {
  return getCategory() + ": " +
    super.toString();
 private String _category = UNIT_TEST;
 protected final static String
  UNIT_TEST = "Unit Test",
  ACCEPTANCE_TEST = "Acceptance Test";
}
```

Fig. 9: JUnit extension

The usage of these extensions is straightforward.

```
public class FrameworkAcceptanceTest
  extends XTestCase {
 public void testBusinessObject() {
  setCategory(ACCEPTANCE_TEST);
  DummyBO bo = new DummyBO("MyBO");
  ...
 }
 ...
}
```

Fig. 10: Usage of JUnit extensions

It is obvious that these extensions provide nothing more than documentation, but that is exactly what is needed. Application and framework developers must be able to separate the different test categories from each other since these are used in different ways.

ACKNOWLEDGEMENTS

I like to thank all my collegues at Apcon Workplace Solutions for their support in applying testing to frameworks.

REFERENCES

1. K. Beck: *eXtreme Programming explained: Embrace Change*. Reading, Massachusetts, Addison-Wesley. 2000.
2. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
3. Junit Web Site: http://www.junit.org
4. The JWAM framework: http://www.jwam.org
5. M. Lippert, S. Roock, H. Wolf, H. Züllighoven: *JWAM and XP - Using XP for framework development*. In: [6]. S. 103-117.
6. G. Succi, M. Marchesi (Eds.): *Extreme Programming Examined*. Reading, Massachusetts, Addison-Wesley, 2001.