A Unit Testing Pattern Language

Peter Gassmann

Unit Testing is a software development practice which has recently gained popularity through a new development methodology called eXtreme Programming (XP). The success of unit tests in a project depends on a well-defined relationship between production and test code. This paper identifies structures, which have been used successfully, and documents them in the form of a pattern language. In addition, important process patterns are identified which help to use unit tests successfully.

Keywords: Unit Testing, eXtreme Programming, JUnit

INTRODUCTION

Unit Testing is a development practice and usually part of an overall testing strategy. Although known for many years, it is not widely used by developers today. This is probably due to the fact that the value of unit testing has been underestimated, and that most developers simply do not know how to apply it. It does not fit well in the traditional development cycle of first designing, then implementing and finally testing big portions of code. In a project with changing and developing requirements, unit tests help to stabilize the system. They also allow for faster and more aggressive changes, because any errors introduced with a change are detected immediately. And they help to create a very short feedback loop for the developer.

Unit testing is not testing done by specialized testers; it is rather part of the daily development routine of a programmer. Unit testing means testing a unit of code. A 'unit of code' in an object-oriented system is usually a class. However, it could also be a whole component or any other piece of related code. A unit test should be an automated test. 'Automated' means that the results are verified automatically. This is necessary to be able to run a large number of tests without human intervention. By running all tests written for the production code, the complete system can be tested automatically. The tests are usually written in the same computer language as the production code. The test verification is therefore code which compares actual results to expected results. If a result is unexpected, the test fails (which is a success because a problem has been identified!). This kind of testing contrasts with the usual traces (in Java `System.out.println(…)`) mixed into the production code, where the programmer looks at the output on the command line and tries to figure out whether the code has behaved as expected.

The success of unit tests in a project depends on a well-defined relationship between production and test code. This paper identifies such structures, which have been used successfully, and documents them in the form of a pattern language. The language consists of structural patterns and process patterns. The patterns are illustrated with solutions implemented in Java. For some patterns, [JUnit] is used to discuss the solution.

The paper intends to help developers starting to use unit tests. It offers help in defining the structures of the unit tests in a project, and it gives information on the development process.

Copyright 2000 Peter Gassmann. All rights reserved.

Organization of the document

The patterns are ordered along a typical development story. The development story is written in blocks that start with the following icon:

Java is used to illustrate the development story. In the development story no differentiation is made between the structural patterns and the process patterns. To see all patterns of one group, the reader may look at the pattern map. It is advisable to read the patterns along with the development story, even if some of them contain forward references to other patterns. The last pattern of the pattern language, UNIT TEST FIRST, deserves special attention because applying this pattern further multiplies the benefits of unit testing beyond the verification of functionality.

The Pattern Form

The pattern form contains the name of the pattern in its title. Some patterns start with a short introduction. Each pattern contains a problem section in the form of a question. The forces section describes the driving forces that should be resolved with the solution. The solution describes the solution in general terms. The discussion section gives additional details, presents sample solutions and relates to other patterns. References to other patterns are shown in SMALLCAPS.

If there are [JUnit] specific hints in the discussion section, they are marked with the icon:

Known Uses

Kent Beck is one of the most important promoters of unit testing. He has written a unit testing framework for Smalltalk and has later ported the framework to Java together with Erich Gamma [JUnit]. There are unit testing frameworks available for many other languages, including C++, Perl and others. See the link under [JUnit].

Some of the patterns are also described in [Binder99]. Particularly in *Incremental Testing Framework* some of the basic structures of a unit testing framework are described.

Many of the structural patterns are implemented in [JUnit]. Therefore it can be concluded that these patterns can be found in most projects using [JUnit]. The author is using [JUnit] on his current project at FJA Feilmeier & Junker AG, where additional structural patterns and the described process patterns are used. A few of the structural patterns are described in [Beck94]. Some of the process patterns are also described online in [Wiki] and in [Beck99], although not in pattern form.

Unit Testing is part of the eXtreme Programming (XP) methodology [Beck99]. But since unit testing is mostly independent of the other XP practices, it may be used in any software development project.

Pattern Map

The following map shows the relationships between the patterns. The arrow points to other patterns that help to resolve the forces introduced or only partially resolved by a pattern. Most of the patterns should not be used alone, only with the help of the other patterns can all forces in the system get in balance.

THE PATTERNS

To explain the patterns in this document, we shall use a software development story. We are going to write an insurance system. An important artifact in insurance software is the contract. Therefore we start with a new class called `Contract`. `Contract` contains various attributes like sum insured, customers rebate and risk category which may be changed using set-methods, and accessed via get-methods. The set- and get-methods are trivial and do not need tests. But `Contract` contains another method, `calculatePremium()`, which is not trivial and therefore should be tested. The premium is calculated using attributes of the contract. As the first thing we need to decide where to put the code to test `calculatePremium`. We introduce the patterns TESTCASECLASS and TESTMETHOD…

TESTMETHOD

Intro

To create a flexible system, the methods of a typical class often accept parameters. Since these parameters may vary during execution, there is a potentially large number of possible inputs and resulting outputs of calling a method. A combination of input parameters which makes sense from a user perspective is often called „test point“ [Binder99, p.47]. There is usually a number of related test points, which vary only in the values of the actual parameters.

Problem

Where should the code be placed which implements the test for a particular test point?

Forces

An effective unit testing system requires that tests be executable independently of other tests. This is important because otherwise the tests would just stop at the first test that fails, and this would make it harder to see where the problem in the system really is.

There are several possibilities of implementing tests. The simplest would be to put the tests for all test points into one big method "test". This, however, would make it difficult to continue execution if one test fails. In addition a method testing, for example, 10 different test points is certainly not easy to understand anymore. Additionally, the method name would not give any hint about the content of the test. It would be hard to find the corresponding test code when the production code gets changed and the test therefore has to be changed, too.

Solution

Write one method for each test for a test point. The method name should reflect the content of the test. This is important to find related tests.

Discussion

It is advisable to place all TESTMETHODs which test the functionality of one production-class in a TESTCASECLASS. This makes it easy to find the implementation of related tests.

An example of related tests and their respective method names, where each test deals with calculating the premium: `testCalculatePremiumMediumRisk`, `testCalculatePremiumLowRisk`, `testCalculatePremiumHighRisk`.

It is advisable to use a naming convention for the names of TESTMETHODs. For example [JUnit] expects a TESTMETHOD to start with „test“. The TESTRUNNER will then be responsible for the execution of a TESTMETHOD.

See also *Test Case/Suite Method* and *Test Case/Suite Class* [Binder99].

Also known as *Test Case* [Beck94].

TESTCASECLASS

Intro A production class has usually more than one method which is non-trivial and needs testing. There are usually several TESTMETHODs related to one production class.

Problem Where should the code for various tests related to one production class be placed (e.g. TESTMETHODs plus additional code)?

Forces The code for the TESTMETHODs could be written in several places. The most obvious place would be directly in the production class. However, this would make it very hard or even impossible to distinguish between production code and testcode. And it would make the production code bigger, which is a problem if the code needs to be loaded over a slow network connection or has to run on a small system. The latter can be avoided by conditional compilation such as provided by C. One advantage of testcode in production code is that the developer is forced to keep the test code consistent when changing the production code.

Another solution would be to put all testcode into one single testclass. A typical software system is made of many classes, sometimes hundreds. If all TESTMETHODs for all production classes were put into one testclass, the result would be the mother of all classes containing hundreds of methods. This would make it easy to find testcode, because it must be in that class. On the other hand, it would be hard to find related tests without extremely well chosen method names.

When developing a class, one would usually like to execute just the tests for that particular class, until they are successful. If all tests are in one big class, how could just the required tests be executed?

Solution All TESTMETHODs for exactly one production class plus additional helper-methods should be placed in a separate TESTCASECLASS. A TESTCASECLASS should contain all tests directly related to one production class. The name of the TESTCASECLASS should be connected to the name of the production class.

Discussion It is advisable to use a naming convention for the names of testclasses. A common pattern is to use “Test“ as a prefix or postfix to the name of the class under test, e.g. `Contract -> ContractTest`. A TESTCASECLASS should be placed in a TESTPACKAGE.

With [Junit], a TESTCASECLASS is usually implemented as a subclass of `junit.framework.TestCase`.

See also *Test Case/Suite Method* and *Test Case/Suite Class* [Binder99].

After applying TESTCASECLASS and TESTMETHOD we have a class `ContractTest` with an empty method `testCalculatePremium()`. Now we want to implement the test methods.

Tests should have a common structure to make them easily understandable. The basic structure of a test is always the same:

1. Prepare IUT (implementation under test [Binder99]).
2. Execute or call IUT.
3. Compare actual results to expected results.
4. Cleanup.

Preparation usually means to instantiate the IUT and then call several set-methods of the IUT to configure a particular test point. Execution of the IUT means to call at least one method of the IUT, which produces an observable result. The observable result is then compared to the expected result. An observable result is usually the return value of the invoked method, it could also be a thrown exception or a value which can be accessed via a get-method.

Cleanup means to free resources used in the test, and undo changes made to the environment. This is important because unit tests must be repeatable. This means it must be possible to execute the tests, develop some functionality, and execute the tests again to see whether the new functionality has introduced a failure. If a test would make persistent changes, for example deleting a required file, or changing a database entry, the test might fail during the next execution. Such a failure would not be the result of wrong code, but an external dependency, which was not met at runtime.

Preparation is often similar for several TESTMETHODs in one TESTCASECLASS. This leads us to the pattern TEST SETUP…

TEST SETUP

Problem Where should code be placed which needs to be executed before each TESTMETHOD in a TESTCASECLASS?

Forces To be able to test a class, a certain amount of test preparation is always required. The simplest case is to create a new instance of the class to be tested, but quite often preparation is more complicated. Usually all TESTMETHODs in a TESTCASECLASS test the same production class. Therefore, test preparation is often similar for all TESTMETHODs in a TESTCASECLASS.

The preparation code could be written at the beginning of each TESTMETHOD. But the resulting code duplication is certainly not a good solution.

As a better solution, the common preparation code could be placed into a separate method, which could then be called at the beginning of each TESTMETHOD. This solves the duplication problem, however, the developer still needs to remember to call the preparation method when he adds a new test to the class later.

Solution

A `setUp`-method should be written in the TESTCASECLASS, which is executed before each TESTMETHOD. The TESTCASECLASS should call the `setUp`-method before invoking the TESTMETHOD.

The basic execution scheme for a TESTMETHOD should look therefore as follows :

```
for all testmethods XXX in testclass {
   testclass.setUp();
   testclass.testXXX();
   testclass.tearDown();
}
```

The `setUp`-method should contain test preparation code that is common to all TESTMETHODs in the TESTCASECLASS. TESTMETHOD specific preparation code would still be placed at the beginning of the TESTMETHOD.

Discussion

The class `junit.framework.TestCase` contains a template-method called `setUp`. The `setUp`-method could be used to CREATE TESTDATA ON THE FLY. The opposite of TEST SETUP is TEST TEARDOWN. The basic execution scheme as described above should be implemented in the TESTCASECLASS (see `junit.framework.TestCase.runBare()`).

Typical logic placed in `setUp` is initializing a database (for example using a TESTDATACREATOR), loading configuration data, and preparing objects for the test.

TEST SETUP could also be applied to a TEST SUITE, to add test preparation code to a number of TESTCASECLASSes. See also TEST DECORATOR.

Also known as *Fixture* [Beck94].

As the result of applying TEST SETUP, which is the first step of the general test structure, our testclass `ContractTest` now looks as follows:

```
public class ContractTest{
   private Contract contract_; // initialized in setUp and used in the tests

   public void setUp(){
      contract_ = new Contract();
      // we initialize our contract
      contract_.setContractType(Contract.LIFE_INSURANCE);
      contract_.setRiskClass(RiskClasses.MEDIUM_RISK);
      contract_.setSumInsured(5000000);
      contract_.setCustomerRebate(0);
   }

   public void testCalculatePremium(){
   }
}
```

In the testmethod (`testCalculatePremium`), we want to compare the result of invoking `calculatePremium` to the expected result. The pattern ASSERT will help us here…

ASSERT

Problem How should testresults be analysed and reported?

Forces To be able to run a large number of tests in reasonable time without human intervention, tests need to verify their success automatically. This means that they have to check whether an actual result is correct or not. One way to implement checking would be by using an if-statement. If the if-statement is false, the test is aborted.

A failure needs to be recorded to provide a summary of all failures and successes. This means that if the if-statement is false, the test must be aborted in a way that the failure is recorded. However, this requires that the developer writes the if-statement and does not forget to register the failure. In this way the comparison and bookkeeping code would be quite complicated.

Solution There should be various methods available when implementing tests, for example in a testframework, to perform basic comparisons of test results. These methods should also support registration of the test result. The comparison methods should be named "assert…", because this is a well known term for evaluating an expression in many languages.

Using a common name has the additional advantage that the places of comparison in the testmethod can be easily spotted.

Discussion The class `junit.framework.TestCase` defines various methods to perform basic comparisons. They all start with "assert". If an assertion fails, e.g. the comparison returns false, the assert-method throws a runtime-exception, which aborts the test. The TESTRUNNER registers a runtime-exception as a failure. Failures or errors are registered in `junit.framework.TestResult` instances, which capture all the necessary information about the failure or error.

[Binder99] contains a chapter on assertions, with example implementations for many programming languages.

ASSERT is implemented as an assertion method [Riehle2000].

Also known as *Check* [Beck94].

An implementation of an assert-method and our testmethod now looks as follows:

```
public void assertEquals(String message, int expected, int actual){
   if ( expected != actual ){
      // the caller of the test needs to catch AssertExeptions to record
      // failures. AssertionFailedError is a simple subclass of
      // java.lang.Error and part of the unit testing framework.
      throw new AssertionFailedError(message + ": expected:"+expected+
                                     ", actual:"+actual);
   }
}
public void testCalculatePremium(){
   // we know that the combination of RiskClass, SumInsured, ContractType
   // and CustomerRebate initialized in setUp should result in the
   // following premium :
   int expectedPremium = 5000;
   int actualPremium = contract_.calculatePremium();
   // we invoke the assertEquals-method.
   // the first parameter will be displayed as an explanation message if the
   // comparison fails.
   assertEquals("premium calculation failed", expectedPremium, actualPremium);
}
```

We have now implemented step two and three of the test structure as explained before the pattern TEST SETUP. This leaves step 4, cleanup. Let us have a look at TEST TEARDOWN…

TEST TEARDOWN

Problem Where should code be placed which needs to be executed after each TESTMETHOD in a TESTCASECLASS?

Forces Since usually all TESTMETHODs in a TESTCASECLASS test the same production class, they require the same kind of testdata. But testdata should be deleted again after a test, even if the test fails. Other cleanup might be required, such as freeing resources. Therefore, test cleanup is usually similar for all TESTMETHODs in a TESTCASECLASS.

The cleanup code could be written at the end of each TESTMETHOD. But the resulting code duplication is certainly not desirable.

As a better solution, the common cleanup code could be placed into a separate method which could then be called at the end of each TESTMETHOD. This solves the duplication problem, however, the developer still needs to remember to call the cleanup method when he later adds a new test to the class. An even bigger problem is that the cleanup code needs to be executed even if the test fails! But if a test fails, e.g. an assertion evaluates to false, execution of the TESTMETHOD is immediately aborted, the end of the testmethod is never reached and therefore the cleanup code not executed. Not executing the cleanup code could lead to failures when the tests are run the next time.

Solution A `tearDown`-method should be written in the TESTCASECLASS, which is executed after each TESTMETHOD. The TESTCASECLASS should call the `tearDown`-method after executing the TESTMETHOD, even if the test failed.

The basic execution scheme for a TESTMETHOD should therefore be as follows :

```
For all testmethods XXX in testclass {
   testclass.setUp();
   testclass.testXXX();
   testclass.tearDown();
}
```

Discussion The class `junit.framework.TestCase` contains a template-method called `tearDown`. The tearDown-method could be used to CLEANUP TESTDATA AFTER TEST. The opposite of TEST TEARDOWN is TEST SETUP. The basic execution scheme as described above should be implemented in a TESTCASECLASS. It is important to note that `tearDown` should be executed by the TESTCASECLASS also when the test failed.

Since we have not made persistent changes or used resources in our tests so far, we can just write an empty `tearDown`-method to complete the common test structure:

```
public void tearDown(){
}
```

Revising the basic structure of a test, we see now that we have a pattern for each step:

1. Prepare IUT (implementation under test [Binder99]) -> TEST SETUP
2. Execute or call IUT -> TESTMETHOD
3. Compare actual results to expected results -> ASSERT
4. Cleanup -> TEST TEARDOWN

We have implemented our first test, which checks the resulting premium for a medium risk contract. We now want to write a test to verify the result for a high-risk contract. We write another test-method for that purpose. We recognize that we need some setUp code common for both test-methods, e.g. creating the `Contract` instance, but we also need some test-method specific preparation code like setting the risk type of the contract. We also change the test-method names to better reflect their content. The result looks as follows:

```
public void testCalculatePremiumMediumRisk(){
   contract_.setRiskClass(RiskClasses.MEDIUM_RISK);
   // we know that the combination of RiskClass, SumInsured, ContractType
   // and CustomerRebate should result in the following premium :
   int expectedPremium = 5000;
   int actualPremium = contract_.calculatePremium();
   // we invoke the assertEquals-method.
   // the first parameter will be displayed as an explanation message if the
   // comparison fails.
   assertEquals("premium calculation failed", expectedPremium, actualPremium);
}
public void testCalculatePremiumHighRisk(){
   contract_.setRiskClass(RiskClasses.HIGH_RISK);
   // we know that the combination of RiskClass, SumInsured, ContractType
   // and CustomerRebate should result in the following premium :
   int expectedPremium = 8000;
   int actualPremium = contract_.calculatePremium();
   // we invoke the assertEquals-method.
   // the first parameter will be displayed as an explanation message if the
   // comparison fails.
   assertEquals("premium calculation failed", expectedPremium, actualPremium);
}
```

Of course we want to execute the tests now. Let us have a look at TESTRUNNER…

TESTRUNNER

Problem How can tests be executed and their results collected?

Forces If the tests are organized in TESTCASECLASSes, each TESTCASECLASS could implement a main-method (in Java, or something similar in other languages). In this main-method, the developer would write code, which calls each TESTMETHOD in the TESTCASECLASS, and records failures and successes. At the end, the failures and successes could be displayed in some way. However, this would require the developer to implement test execution logic for each TESTCASECLASS separately, again and again. Additionally, with this concept it would be difficult or impossible to execute many TESTCASECLASSes in a series, and collect the results, which is required when testing the whole system to see whether all tests work or not.

Solution There should be a helper class, which automatically executes any class which contains TESTMETHODs. The class should register failures and errors during execution of the tests. A failing test must be handled in such a way that the following TESTMETHODs are still executed.

A class to be executed by the TESTRUNNER should contain a method which executes all TESTMETHODs contained in that class (e.g. `runTests()`).

Discussion [Junit] requires a TESTCASECLASS to implement the interface `junit.framework.Test` to be executable by the TESTRUNNER.

We have implemented a simple version of TESTRUNNER for our purposes. To be able to execute different TESTCASECLASSes, we shall now write an interface `Test` which defines the methods to be used by TESTRUNNER:

```
public interface Test{
   /** Execute all tests and return the number of failures /
   public int runTests();
}

public class Testrunner{
   /** The first argument is expected to be the name of a class that
       implements the Test interface */
   public static void main(String[] args){
      Test testClass = (Test)Class.forName(args[0]);
      executeAllTests(testClass);
   }
   private static void executeAllTests(Test aTest){
      int numberOfFailures = aTest.runTests();
      if ( numberOfFailures > 0 ){
         System.out.println("Failed: Number of failures for Test ["+aTest+"]:"+
                           numberOfFailures);
      }
      else{
         System.out.println("Successfully run ["+aTest+"].");
      }
   }
}
```

We shall add the `Test` interface to our testclass and implement it as follows:

```
public class ContractTest implements Test{
   public int runTests(){
      int numberOfFailures = 0;
      try{
         setUp();
         testCalculatePremiumMediumRisk();
      }
      catch(Exception e){
         numberOfFailures++;
      }
      finally{
         tearDown();
      }
      try{
         setUp();
         testCalculatePremiumHighRisk();
      }
      catch(Exception e){
         numberOfFailures++;
      }
      finally{
         tearDown();
      }
   }
}
```

We now know how to write tests and how to execute them. We can now add more production code and more tests. To make sure that we do not introduce undetected errors, we have to execute the tests as often as possible or necessary…

RUN OFTEN

Problem How often should tests be executed?

Forces Unit tests may be run frequently or only at special occasions in the development process. The overall development process with unit tests is as follows :

1. Run the tests. They must execute completely successful.
2. Develop some code.
3. Run the tests again. If there are failures, step 2 must have introduced the problem. Fix the problem. If test execution in step 1 had not been completely successful, it would require additional effort to find out whether step 2 introduced the error, or if the error was already there before step 2.

Tests could be executed just before functionality is delivered to the customer (traditional testing at the end of the project). Since this happens usually only after a few months or even years of development, the developer would learn very late that his functionality does not work as expected. And he certainly could not remember every change he has made to the system after the last test execution.

Tests could also be executed once a week. The feedback cycle would be shorter this way. But in one week so much functionality is typically developed that could be responsible for a test failure, that it still would be rather difficult to remember or find out which part of the new functionality made a test fail.

On the other hand, making a small change and then running the whole test suite might be quite time-consuming. In an average system, executing all unit tests of the system can easily take a few minutes.

Solution The tests should be run often, to get feedback from the tests as soon as possible. If only a few minutes have passed since the last successful execution, the developer will easily remember what he has changed. And one of these changes must be responsible for the failure. The immediate feedback is one of the main benefits of having unit tests [Beck99], [Gassmann].

How often to run the tests depends on the magnitude of the change (e.g. the risk that something might break because of the change), and the time required to run the test(s). It is not always necessary to run all tests of the system, normally the tests for the class one is working on are sufficient.

Discussion In a typical development scenario, tests are executed a few times per hour. First just the tests for the production class, which is being developed. This may happen many times per hour. When these tests are all successful, the tests for the package and then the whole system are executed as well, to see if there are any side effects on other parts of the system.

If it is too complicated or if it takes too much time to execute a test, developers will not invoke the tests often enough. MAIN-METHOD may be considered to make it easy to start a test. TEST DECORATOR may be considered to optimize execution time if all TESTMETHODs are using the same set of testdata.

We have started to develop another class `Customer`. We are also writing tests for this class, so we have another testclass called `CustomerTest`. It is already foreseeable that in the end we will have many testclasses. Should we keep the testclasses in the same package as the production code? Let us have a look at TESTPACKAGE…

TESTPACKAGE

Problem Into which package or namespace should test code be placed?

Forces In an average system there will be many testclasses (see TESTCASECLASS). In a well-tested system there might be as many as one TESTCASECLASS for each production class.

The testcode could be placed into the same package as the production code. But there is no obvious way to distinguish test code from production code this makes the system less understandable for the developers.

Additionally, when preparing the system for the end-users, the test code is usually not put into the final executable to save space, particularly if the code needs to be loaded via network. Therefore the person who does the preparation needs to know which code is production code and which is test code as well.

If the test code is placed in the same package as the production code, the package visibility feature of the Java language could be (mis-)used in the tests. Since another user of a class outside of the package cannot use package visibility features, they should neither be used in the tests.

Solution At least one separate package for the TESTCASECLASSES should be created. This makes it easy to identify test code, and it provides a more realistic scenario on how another class may use a production class because the package visibility feature of the Java language cannot be used in the tests.

Discussion The test package contains all testclasses, plus additional helper classes like TESTDATACREATOR. In a large system, it is advisable to create a TESTPACKAGE for each production package. To execute all testclasses in a test package, ALLTESTS may be considered.

If the package visibility feature needs to be used, separate production code and test code source trees may be considered. This way test code could be placed logically in the same package as the production code, but for the developer it would be in a different location. Note however that, for example, VisualAge for Java does not support such a setup.

We have written a number of TESTCASECLASSes for all our `Contract` related classes. But we still have to start the TESTRUNNER for each TESTCASECLASS separately. Now we would like to execute all tests related to `Contract` in one go…

TEST SUITE

Problem How can several tests be combined to run together?

Forces In a typical development scenario, the tests are executed a few times per hour. First just the tests for the production class (for example `ContractTest`) which is beeing developed. When these tests are all successful, the tests for the package and then the whole system are executed as well, to see if there are any side effects on other parts of the system. This means that many Testmethods defined in various TestCaseClasses must be executed together.

One solution would be to define for each TESTMETHOD the following TESTMETHOD to execute. But in the above scenario, the TESTMETHODs defined in `ContractTest` must be executed first alone, then together with the other tests in the same package and finally with all tests of the system.

Solution A class should be created which implements the required interface to be executable by the TESTRUNNER (e.g. `runTests()`). In `runTests`, any number of other classes that contain tests may be called. Schematically, this could be as follows:

```
Testrunner.executeAllTests{
   -> mainTestSuite.runTests(){
      -> test1.runTests(){
         // TestSuite
         -> test11.runTests(){
            -> testMethod111()
            -> testMethod112()
         }
         -> test12.runTests(){
            -> testMethod121()
         }
      }
      -> test2.runTests(){
         // TestSuite
         -> test21.runTests(){
            // TestSuite
            -> test211.runTests(){
               // TestCaseClass
               -> testMethod2111()
            }
         }
      }
      -> test3.runTests(){
         -> testMethod31()
         -> testMethod32()
      }
   }
}
```

In this way, test suites may be composed from other tests and test suites.

Discussion The class `junit.framework.TestSuite` is used to group tests to be executed together. The framework uses `TestSuite` to extract the TESTMETHODs from a TESTCASECLASS.

See also *Test Case/Suite Method* and *Test Case/Suite Class* [Binder99].

See also *Test Suite* [Beck94].

We have already quite a few tests that we execute often to test our system. Is there no easier way to execute the tests than to start the TESTRUNNER and type in the name of the TESTCASECLASS (that is how [JUnit] requires it)?

MAIN METHOD

Intro In a typical development scenario, tests are executed many times per hour. If starting a test is too complicated, developers will start the tests less frequently, which lowers the benefits of having unit tests.

Problem Which is the easiest way to start a particular test?

Forces There are several ways how a unit test may be started using a TESTRUNNER, as for example implemented in [JUnit].

The first way is to start the TESTRUNNER and type the name of the TESTCASECLASS into a field on the TESTRUNNER-GUI. This requires that the developer types the correct name of the test every time he wants to execute the test.

A newer version of [JUnit] stores a history of executed tests. But with over 100 testclasses in a project (as for example in the project described in [Gassmann]), this again becomes difficult to use.

Solution In Java, a main-method should be written which calls the TESTRUNNER, which in turn executes the tests. Note that this solution only makes sense if the development environment makes it easy to start any class with a main-method.

Discussion In a development environment like VisualAge for Java, which allows execution of a class with just one mouse-click or a keyboard-shortcut, writing a main-method is certainly the most elegant solution. Since the main-method for the test-classes always looks the same, it might help to automate writing the main-method by providing a tool, which generates the main-method for a particular testclass. Here is a sample main method for an AllTests-class which uses [JUnit] :

```
Public static void main(){
   String[] myargs = new String[]{AllTests.class.getName()};
   junit.ui.TestRunner.main(myargs);
}
```

The default way to execute a test with [JUnit] is to start the TESTRUNNER and type in the name of the TESTCASECLASS. This is clearly not the most effective way to execute tests.

See also *Command Line Test Bundle* [Binder99].

We have just applied MAIN METHOD to our testclasses. After having written a few more TESTCASECLASSes, it becomes apparent that our TESTSETUP methods contain a lot of duplicated test preparation code. Where should we put common test preparation code?

TESTDATACREATOR

Problem Where should common code to prepare testdata be placed?

Forces When testing a system with different entities, which have relations to each other, there is often a common testdata-structure. For example, in an insurance system there is usually a partner with addresses, who has one or more contracts. When testing the contract-related classes, a complete set of data is needed. But the same set of data is also needed while testing the partner-related classes, e.g. there is more than one testclass that needs to create the same set of testdata.

The test preparation code could be written in every testclass that requires it. This would lead to undesirable code duplication. It would also make it difficult to find all the places to change, if, for example, the code to create an address-object changes in a basic way.

Solution A helper-class should be written to create different sets of testdata. This concentrates the required logic in one place and avoids duplication.

Discussion The testdata-creator class should be placed in a TESTPACKAGE. The testdata-creator class may be used to CLEANUP TESTDATA AFTER TEST. The testdata-creator class CREATES TESTDATA ON THE FLY.

A typical usage scenario for TESTDATACREATOR could be as follows :

```
public void testGetPartner{
   // create test fixture / test setup
   TestDataCreator.initialize();
   Partner partner = TestDataCreator.createPartner(„John“, „Doe“);
   Address address = TestDataCreator.createAddress(„Street“,
                                   „Samplecity“, partner);
   // perform test and check result
   assertEquals(„wrong partner“, partner, address.getPartner());
}
// see pattern „Test tearDown“ for more on this method
public void tearDown(){
   TestDataCreator.deleteData();  // delete data created earlier
}
```

We finally have tested most of our current production code. We would now like to integrate our code with what the other programmers in the team developed today. But we want to make sure that our tests will also work tomorrow…

RUN AT 100%

Problem How many tests should be successful when integrating?

Forces Unit tests provide their greatest benefit when they are simple to understand. This is particularly true when the results of the unit tests have to be interpreted. It is easiest to interpret a thumb-up, thumb-down indicator. Either the test was successful, or it failed. Now, if there are many tests, potentially hundreds, are the tests successful if 10% failed? Or are they successful if only unimportant tests failed? How should a developer decide if the system works, given that not all tests were successful?

Solution All tests (=100%) must be successful when integrating code. If it is allowed to integrate failing tests, the tests will loose their value – namely, simple to understand feedback - very soon. They will not be trusted anymore. If a developer starts writing code with tests that are not working at 100%, he can no longer decide whether a test has failed because of his changes, or whether the failure was already there before he started.

Discussion This point is actually important at various levels. The thumb-up indicator serves also as a motivator. Therefore, the more often the thumb is up, the better the motivation effect. This is true on the TESTCASECLASS level (all TESTMETHODS are successful) and on the ALLTESTS level (all TESTMETHODs in all invoked TESTCASECLASSes are successful).

We have done a checkout from the central code repository to get all changes done by the other developers. We have executed all our tests but a few of them did not work anymore, so we had to fix them. Now we are ready to commit our changes…

RUN BEFORE INTEGRATION

Problem When should the tests be executed in the integration process?

Forces As described above in RUN AT 100%, only successful tests should be integrated.

So the typical scenario would be to develop code, then run all tests on the local machine, and then commit the changes to the central repository.

But what happens if another developer in the meantime integrated code that does not work (anymore) together with the code to be integrated? The result would be that the tests in the central repository do not run 100% successfully.

Solution All tests, which means all tests from the central repository plus all new or changed tests on the local machine, should be run immediately before integration, in general directly before committing the changes to the central repository. They have to run at 100%. This requires downloading all changes made in the central repository since the last integration, before running the tests.

Discussion For a detailed example of an integration process, see [Gassmann2].

It might be necessary to explicitly serialize integration for different developers, so that no two developers can integrate at the same time. This could be done by setting up a separate integration machine, or by using an integration token (for example a hat). Only the person sitting on the integration machine or having the integration token may integrate code. If two developers are allowed to integrate at the same time, this could produce a situation where all tests of each developer were successful, but the resulting combination still leads to failures.

See also *High-frequency Integration* [Binder99].

We have now so many tests that we start forgetting to execute some of them to test the complete system. But as stated in RUN BEFORE INTEGRATION, we have to execute all our tests successfully before we are allowed to commit. We need a way to run all of our tests automatically…

ALLTESTS

Problem How can all tests be executed for a set of classes, a subsystem or even the complete system?

Forces In an average system there will be more than one testclass (in the project described in [Gassmann] there are well over 100). When testing a component or the complete system, all testmethods in all testclasses should be executed, one after the other.

The developer could execute each testclass manually. But in this way he is likely to forget one, it will take longer than necessary, and it will not be done very frequently. In addition, there will be no overall thumbs-up / down sign.

Solution For each logical set of TESTCASECLASSes, an ALLTESTS-class should be created. The ALLTESTS-class executes all TESTMETHODs in all testclasses belonging to that set, one after another. The result of executing ALLTESTS should be a summary of the successes and failures of all tests executed.

Discussion ALLTESTS is best implemented using a TEST SUITE.

Typical sets of classes are all TESTCASECLASSes in a TESTPACKAGE. There should be at least one ALLTESTS in each TESTPACKAGE, which executes all tests in the respective package. A MAIN-METHOD may be used to start execution. To test the complete system, an AllAllTests-class should be created, which in turn calls all ALLTESTS-classes of each TESTPACKAGE. Here is an example of an ALLTESTS-class for a TEST PACKAGE:

```
import junit.framework.*;
/** AllTests for this package to call all unit-tests.*/
public class AllTests{

   /** start the JUnit GUI with this class.*/
   public static void main(String[] args){
      String[] myargs = {AllTests.getClass().getName()};
      junit.ui.TestRunner.main(myargs);
   }

   /** suite method.*/
   public static Test suite(){
      TestSuite suite = new TestSuite();
      suite.addTest(new TestSuite(AddressModelTest.class));
      suite.addTest(new TestSuite(ContractModelTest.class));
      suite.addTest(SearchModelTest.suite());
      return suite;
   }
}
```

See also *Retest All* in [Binder99].

We have started to write tests for our database layer. Testing code that generates or accesses persistent data adds another dimension to our tests. The testcode itself looks still the same, but our tests suddenly fail because the data we exepected to be in the database for the test was not there anymore after we had to regenerate the database…

CREATE TESTDATA ON THE FLY

Problem When should testdata be created which is used in a test?

Forces In tests which access persistent data storage, the data used in the test may be generated in several ways. One way is to enter the data manually or with a script into the database before the tests are run. However, if the developer forgets to enter the data before the test, the tests will fail not because the code does not work, but because an external dependency was not met. In addition, if the data is needed in several tests, and changed during some of them, it would not be possible to run ALLTESTS, because after the first test changed the data, the following tests would fail.

If the tests have unstable external dependencies, they will fail from time to time because of these external dependencies, and not because the code does not work! If this happens frequently, the developers will loose their trust in the unit tests.

Solution Create testdata on the fly, during the test. This way the test has fewer dependencies on the test environment.

Discussion It might be complicated or even impossible to create testdata during the test. In these cases, there should be a mechanism that verifies if the expected data is present, before running the test. Otherwise it would not be immediately clear whether the tests failed or just a prerequisite for running the tests was not met.

The logic to create testdata may be placed in TESTDATACREATOR. This logic is often invoked from TEST SETUP. When creating testdata on the fly, and storing the data in the datastorage, it should not be forgotten to CLEANUP DATA AFTER TEST.

Now that we have fixed the problem with persistent data used in the test, we note that we can execute our tests only once. What has happened? The data generated during the test remains in the database and makes the test fail when executed the next time…

CLEANUP TESTDATA AFTER TEST

Problem How can the tests be made repeatable?

Forces If a test makes persistent changes to the test environment, for example inserting data into a database, the changes could be just left there. However, there is the chance that the test will fail when run the next time, because the test preparation code cannot insert the testdata a second time. This could happen if the test requires specific data, which cannot be inserted twice into the database because of an index that enforces uniqueness.

This would make the test not repeatable, which is one of the most important properties of unit tests [Beck99][Gassmann].

Solution After the test, the test environment should be restored to the state before the test e.g. any changes must be undone.

Discussion The logic to delete testdata after execution of the test could be placed in the TESTDATACREATOR. The call to delete the testdata is best placed in TEST TEARDOWN, because the TESTRUNNER calls `tearDown` even if the test fails.

When testing database access logic, it is often not possible to use the rollback-mechanism of the database, because a commit is usually required during the test.

We have now changed our tests which insert data into the database, for example, as follows:

```
public void testCreateAddress(){
   Partner partner = new Partner();
   partner.setName("Doe");
   // now we insert the data into the database,
   // creating testdata on the fly . . .
   try{
      partner.save();
   }catch(SaveException e){
      // fail aborts the test.
      fail("unexpected exception in save of partner : "+e);
   }
   Address address = new Address();
   address.setCity("Samplecity");
   address.setPartner(partner);
   try{
      address.save();
   }catch(SaveException e){
      // fail aborts the test.
      fail("unexpected exception in save of address : "+e);
   }
   // we do some testing
   assertEquals("address has wrong partner", partner, address.getPartner());
   // we do the clean-up after the test . . .
   try{
      address.delete();
      partner.delete();
   }catch(DeleteException e){
      fail("unexpected exception in delete:"+e);
   }
}
```

The above code applies CLEANUP TESTDATA AFTER TEST at the end of the TESTMETHOD. But what happens if the insert of the address fails? The partner is already in the database, but the test is aborted before the cleanup code is executed. To avoid this problem, cleanup code should be placed in TEST TEARDOWN.

The problem with testdata that remains undeleted during the test was recognized only a few hours later, and we are a bit worried about that. How can we make sure that we recognize such problems earlier?

Run Twice

Problem How can be ensured that the tests are repeatable?

Forces The system under test might be changed during test execution. Data might be stored persistently, without beeing cleaned up after the test, caches might be filled etc. Since these changes in the system might influence the next execution of the tests, such problems should be detected as soon as possible.

Solution The tests should be run at least twice in a row.

Discussion Running the tests twice in a row helps to detect most problems related to caches and undeleted data immediately. And executing the tests twice is usually still fast enough to have the developer not wait too long. The TESTRUNNER window should not be closed between the two runs.

We have now started to test our search logic. But even with CREATE TESTDATA ON THE FLY we get false hits in some tests. It seems that there is data in the database which disturbs the tests, data that has nothing to do with the tests…

NONSENSICAL TESTDATA

Problem How can conflicts with unexpected data in the datastorage be avoided?

Forces In a system which stores data persistently, e.g. in a database or files, the datastorage is usually accessed in different scenarios:

- when executing the system (e.g. with a GUI),
- when running the unit tests,
- when running the functional tests.

When executing the system with the GUI, the developer quite often enters data to see if and how the system works. This data usually stays in the datastorage. *But some unit tests might expect particular data to be present or not.* For example, when testing search logic, data with specific valid-from and valid-to dates might be used during the test. But if there is unexpected data in the database, the tests might fail even if the code works as it should.

Solution For the unit tests, nonsensical test data may be used, e.g. data where there is a very low probability that it is present in the database. For example, valid-from and valid-to dates from the last century could be used.

Discussion It is not always possible to use non-sensical data. In these cases, the unit tests need to take into account the possibility of existing data in the datastorage. This pattern is particularly useful when testing search-logic.

We have now consequently applied CREATE TESTDATA ON THE FLY and TEST SETUP for our search logic tests. The search logic tests use read-only access to the database. There are quite a few test methods, and for each test method a lot of data is inserted in TEST SETUP and again deleted in TEST TEARDOWN. The result is that the tests are very slow…

TEST DECORATOR

Problem Often there is code that needs to be executed only once for all TESTMETHODS in a TESTCASECLASS. Where should this code be placed?

Forces There are some tests which do not need to make changes to persistent data, but still require persistent data to be present, for example tests for search-logic. Usually all TESTMETHODs in a TESTCASECLASS for this kind of logic require read-only access to the database.

When using the TEST SETUP approach together with CREATE TESTDATA ON THE FLY, one would insert the testdata into the database before each TESTMETHOD. However, this makes the tests take longer than necessary. If it takes too long, the developers will hesitate to start the tests and the main benefit – immediate feedback – will be lost.

Solution A test-class should be written to wrap (decorate) another testclass. The class contains `setUp` and `tearDown`. After TESTCASECLASS has executed `setUp`, the class executes the TESTMETHODs in the wrapped class using the same execution scheme as the TESTCASECLASS.

```
TestCaseClass.runTests(){
   decorator.setUp();
   decorator.runTests(){
      for all testmethods XXX in testclass{
         testclass.setUp();
         testclass.testXXX();
         testclass.tearDown();
      }
   }
   decorator.tearDown();
}
```

In this way, one can still benefit from applying TEST SETUP and CREATE TESTDATA ON THE FLY while making the execution of the tests faster.

Discussion [JUnit] contains the class `junit.extensions.TestSetup`, which is a subclass of `junit.extensions.TestDecorator`, and which can be used to implement a static fixture. In the following example, the class `ContractTestSetup` extends `TestSetup`. The methods `setUp` and `tearDown` have been overwritten to implement test preparation and test cleanup. The setup-class is used in `ContractTest` as follows:

```
public class ContractTestSetup extends junit.extensions.TestSetup{
   // setUp and tearDown as in other test-classes
}
public class ContractTest extends junit.framework.TestCase{
   // in the testclass the setup-class is used as a wrapper
   public static TestSuite suite(){
      return new ContractTestSetup(new TestSuite(ContractTest.class));
   }
   // test-methods omitted
}
```

A typical scenario for a static fixture is preparing and inserting testdata when testing search-logic, since these tests require no write-access to the database.

We have now written many tests for our system. We always started by implementing the production code, and then we wrote the test code. Sometimes we forgot to write a test at all, and quite a few times we were not very motivated to write the test after we were sure that the production code was already working. Even worse, we had a few cases where we had to change the production code to make it testable! Is something wrong with the order of development? The next and last pattern is probably the most important of all patterns…

UNIT TEST FIRST

Problem When should the tests be written during development?

Forces Unit tests may be written before, during or after development of the production code. Writing the tests after development has the disadvantage that the design of the code to be tested is already fixed. This sometimes leads to problems because the code might be structured in a way which makes testing difficult or impossible. But even more important is the motivation factor. A developer will not be very motivated to write a test if he believes the code is already working. Writing it beforehand is hard for developers, because there is nothing to test yet.

Writing the tests at the same time as the production code is logically impossible.

Solution The unit tests should be written, if possible, before the production code. If the tests are written before the production code is written, the motivation will be to make the tests run! And if all tests run, the developer knows he has completed the task.

Discussion Writing the test before the production code has another huge advantage: When writing a test, the developer is creating the "micro"-design of the system. The components in a system developed with unit tests before the production code are less dependent on each other, and the methods communicate much more error information, for example via exceptions. This is a result of the „caller“-perspective while writing the tests, instead of the „implementor“-perspective usually taken by developers.

This style of development leads to the following development cycle:

1. Write a test.
2. Implement the production code necessary to make the test run.
3. Run the test.

This cycle is usually executed many times during a day, often even many times during an hour of development.

More information can be found in [Beck99], [Wiki] and [Gassmann].

CONCLUSIONS

The patterns in this pattern language describe how to structure unit tests. They further give advice on how to structure the development process. Applying the structural patterns makes the tests better understandable and help to navigate in testcode written by another developer. Applying the process patterns reduces some of the problems associated with unit tests, and in the case of UNIT TEST FIRST further improve the benefits of unit testing.

Unit testing frameworks like [JUnit] directly support most of these patterns. The framework should make it as easy as possible to write and execute unit tests to maximize the benefits while minimizing the effort necessary.

The patterns are applied with success in the project the author is working on.

ACKNOWLEDGMENTS

The author would like to thank Christa Schwaninger for many valuable suggestions during the EuroPLoP shepherding process. He would also like to thank Robert H. Gassmann for helping to improve his english skills. Thanks go to Claudia Chini and Christian Ulmann for their comments and suggestions. Thanks to Dirk Riehle for his feedback, suggestions and known uses. Thanks to Frank Westphal for his comments. And finally thanks to Kent Beck for infecting me with the unit testing virus.

REFERENCES AND FURTHER READING

[Beck99] K. Beck. *Extreme Programming explained, embrace change.* Reading, MA: Addison-Wesley, 1999

[Beck94] K. Beck. "Simple Smalltalk Testing: With Patterns". In *The Smalltalk Report* 4(2), October 1994. Available online: http://www.xprogramming.com/software/…

[Binder99] R. Binder. *Testing Object-Oriented Systems. Models, Patterns and Tools.* Reading, MA: Addison-Wesley, 1999

[Fowler99] M. Fowler, et al. *Refactoring, improving the design of existing code.* Reading, MA: Addison-Wesley, 1999

[Gassmann] P. Gassmann. *Unit Testing in a Java Project.* March 2000, paper written for the conference XP2000.

[Gassmann2] P. Gassmann. *Development process with VisualAge and vaj2cvs.* February 2000. see http://…

[JUnit] K. Beck and E. Gamma. *JUnit, Unit testing framework for Java*
The testing framework for Java (and other programming languages) can be found under *http://www.xprogramming.com/software.htm*

[JR1] K. Beck and E. Gamma. "JUnit: A Cook's Tour." In *Java Report*. May 1999

[JR2] K. Beck and E. Gamma. "Test Infected: Programmers Love Writing Tests." In *Java Report*. July 1998

[Riehle2000] D. Riehle. "Method Types in Java". In *Java Report*. February 2000.

[Wiki] *http://c2.com/cgi/wiki?UnitTests* A webpage about unit testing.

Peter Gassmann can be reached at peter.gassmann@acm.org **or at** FJA Feilmeier & Junker AG, Zollikerstrasse 183, 8008 Zurich, Switzerland

Copyright 2000 Peter Gassmann. All rights reserved.
Permission for publication granted for the purposes of EuroPLoP 2000.