A Unit Testing Pattern Language

Peter Gassmann, FJA Feilmeier & Junker AG, Zurich
peter.gassmann@acm.org
Permission for publication granted for the purposes of EuroPLoP2000.

Unit Testing is a software development practice which has recently gained acceptance through a new development methodology called eXtreme Programming (XP). The success of unit tests in a project depends on well defined structures of the code. This paper identifies such 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 using unit tests successfully.

Keywords : Unit Testing, eXtreme Programming, JUnit

1 Introduction

This document describes a collection of patterns related to unit testing, which together form 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 describe a possible solution.

The paper should be useful for developers starting to use unit tests. It helps defining the structures of the unit tests in a project, and it gives hints on the development process. The paper should also be useful for people planning to develop a unit testing framework or a unit testing tool, because a lot of information is given on the relationships between the different elements of which unit tests consist.

1.1 The Pattern Form

The pattern form contains the name of the pattern in the title. Each pattern contains a problem section in the form of a question. The forces section describes the driving forces, which 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.

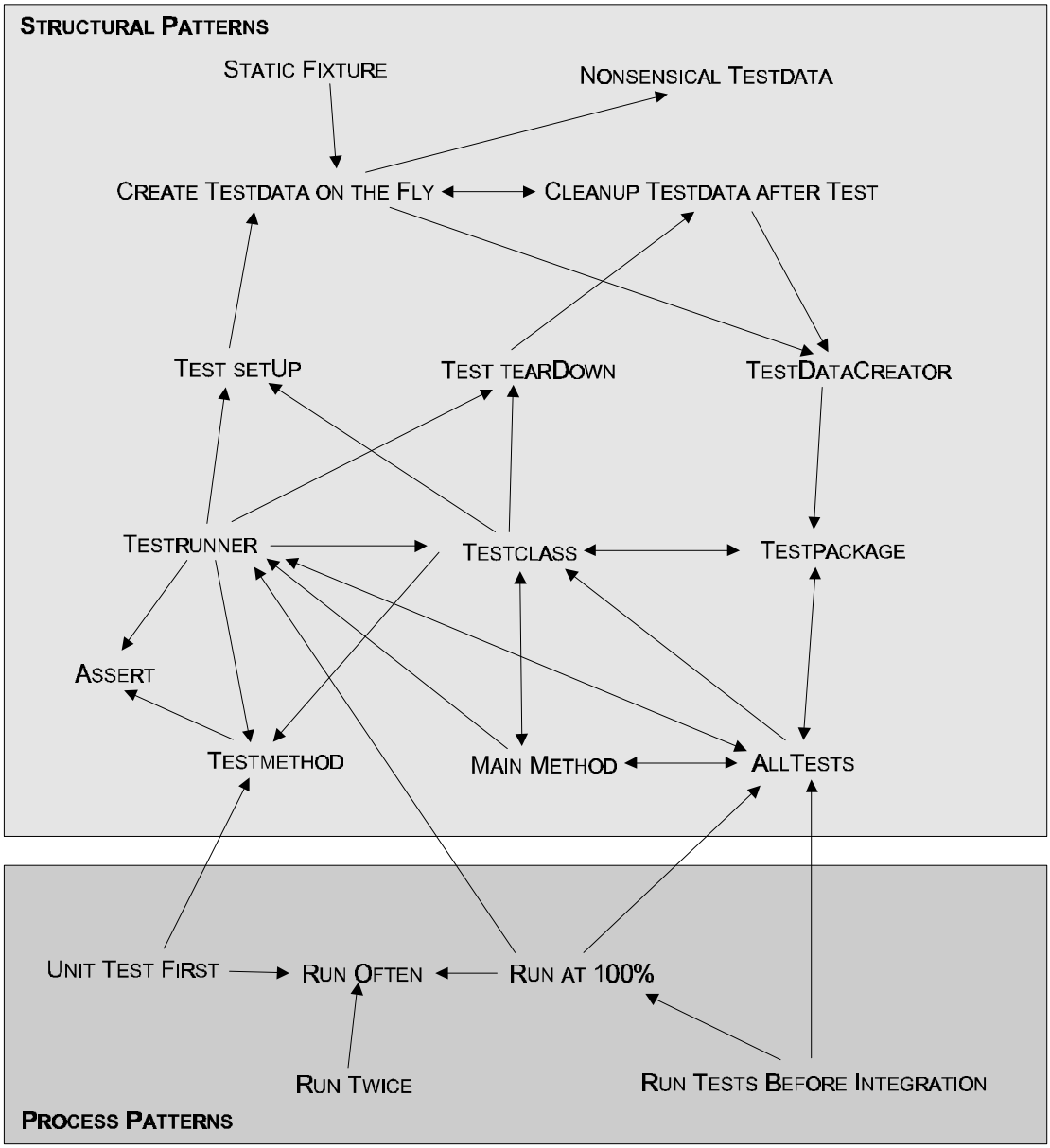
If there are [JUNIT] specific hints in the discussion section, it is marked with the icon: 

1.2 Known Uses

Many of the structural patterns are implemented in [JUNIT]. Therefore it can be concluded that these patterns can be found in any project using [JUNIT]. The author is using [JUNIT] on his current project, where in addition to the structural patterns the described process patterns emerged. Some of the process patterns are also described on [WIKI] and in [XP], although not in pattern form.

1.3 Pattern Map

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



2 Structural Patterns

2.1 Introduction to Structural Patterns

The structural patterns describe how to organize and structure the code to implement unit tests. The patterns are ordered small-scale to large-scale (TESTMETHOD to ALLTESTS), and basic testing to complex testing (TESTMETHOD to TESTDATACREATOR and STATIC FIXTURE).

2.2 TESTMETHOD

Problem Where should the code be placed which implements a testcase?

Forces 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 „testcase“. There is usally a number of related testcases, which vary only in the values of the actual parameters. Related testcases should be kept together.

Too complicated unit tests will be an additional source of errors. Therefore, the implementation of the tests should be as simple as possible and as understandable as possible.

Solution Write one method for each testcase. The method name should reflect the content of the testcase.

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

An example of related testcases and their respective methodnames, where each testcase deals with deleting a varying number of adresses: `testDeleteWithOneAdress`, `testDeleteWithTwoAdresses`, `testDeleteWithNoAdress`.

It is advisable to use a naming convention on how to name a TESTMETHOD. For example [JUNIT] expects a TESTMETHOD to start with „test“. The TESTRUNNER is responsible to execute a TESTMETHOD.

2.3 ASSERT

Problem How should testresults be analysed ?

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

1. Prepare test
2. Execute test
3. Compare actual results to expected results.

Preparation and execution is different for each test. However, the logic for the comparisons is basically the same for all tests.

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. Since result-checking is needed in each TESTMETHOD, it should be structured in a common way and should be as easy as possible.

Solution There should be various methods available when implementing tests, for example in a testframework, to perform basic comparisons of test results.

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 runtime-exception is caught by the TESTRUNNER, which records the failure.

2.4 TESTCLASS

Problem Where should the code for various testcases related to one production-class be placed?

Forces Testcode should not be mixed with production code, to keep a clean separation and to avoid unnecessary dependencies. A production class has usually more than one method which needs testing. Related tests should be easy to find. The relationship between production code and testcode should be visible somehow.

Solution All TESTMETHODs plus additional helper-methods should be placed in one TESTCLASS. A testclass should contain all testcases directly related to one production-class.

Discussion It is advisable to use a naming convention on how to name testclasses. A common pattern is to use “Test“ as a prefix or postfix to the name of the class under test, e.g. PartnerModel -> PartnerModelTest. A TESTCLASS should be placed in a TESTPACKAGE.

2.5 MAIN METHOD

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

Forces Unit tests support the developer best if they are both easy to write and easy to execute. Ideally, a click on one button or a simple keyboard shortcut should be enough to start a test. This way, executing a test is not complicated and takes almost no time. The easier and faster, the more often will a test be executed.

Solution In Java, a main-method should be written, which calls the TESTRUNNER, which in turn executes the tests.

Discussion In a development environment like VisualAge for Java, which allows to execute 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 class. 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 TESTCLASS. This is clearly not the most effective way to execute tests.

2.6 TESTPACKAGE

Problem Into which package should the testclasses be placed ?

Forces In an average system there will be many testclasses. There is no obvious way to distinguish testclasses from production classes if they are placed in the same package. The system is easier to understand if production code and test code is clearly separated. If the testcode 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 visible features, they should not be used in the tests as well.

Solution At least one separate package for the TESTCLASSES should be created.

Discussion The testpackage 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 testpackage, ALLTESTS may be considered.

2.7 ALLTESTS

Problem How can all tests for a set of classes / package or the complete system be executed ?

Forces In an average system there will be more than one testclass. When testing a component or the complete system, all testmethods in all testclasses should be executed, one after the other. But if the developer needs to execute each testclass manually, he is likely to forget one, it will take longer than necessary, and it won‘t be done very frequently.

Solution For each logical set of TESTCLASSes a ALLTESTS-class should be created. The ALLTESTS-class executes all TESTMETHODs in all testclasses belonging to that set.

Discussion A typical set of classes are all TESTCLASSes 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, a AllAllTests-class should be created, which in turn calls all ALLTESTS-classes of each TESTPACKAGE.

2.8 TESTRUNNER

Problem How should the tests be executed ?

Forces In a system with a potentially large number of TESTCLASSes, it should not be necessary to write the logic to execute the tests again and again. It should also be possible to execute a series of tests. To keep the tests independent of each other, a failing test should not affect other tests, even if an exception was thrown.

Solution There should be a class which automatically executes all TESTMETHODs in a TESTCLASS. A failing TESTMETHOD must be handled in a way that the following TESTMETHODs may still be executed.

Discussion [JUNIT] contains various classes to execute tests. One of them is `junit.ui.TestRunner`. The TESTRUNNER implements the basic execution scheme for tests :

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

Note that execution of the testmethod must be guarded against failures and exceptions, to avoid affecting other tests. It should also be guaranteed that tearDown is executed in any case.

2.9 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 adresses, 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 when testing the partner-related classes, e.g. there is more than one testclass which needs to create the same set of testdata.

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 DELETE TESTDATA AFTER TEST. The testdata-creator class CREATES TESTDATA ON THE FLY.

A typical usage scenario for TESTDATACREATOR could look as follows :

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

2.10 CREATE TESTDATA ON THE FLY

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

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

Additionally, after datamodel-changes the database needs to be deleted and then recreated. Therefore, tests cannot expect specific data to be present in the datastorage, unless it can be guaranteed that the required data will be present whenever the tests are executed. 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 less dependencies on the testenvironment.

Discussion It might be complicated or even impossible to create testdata during the test. In these cases, there should be a mechanism which 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 and STATIC FIXTURE. When creating testdata on the fly, and the data is stored in the datastorage, it should not be forgotten to DELETE DATA AFTER TEST.

2.11 NONSENSICAL TESTDATA

Problem What testdata should be used to avoid conflicts with possibly existing data in the datastorage ?

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 possibility 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.

2.12 DELETE TESTDATA AFTER TEST

Problem How can the tests be made repeatable ?

Forces One of the main benefits of unit tests is that they are repeatable [Beck][Gassmann]. But if, for example, data created during a test is left in the datastorage after the test, there is the chance that the test will fail when running the next time. To be repeatable, a test must not make persistent changes to the test environment.

Solution The test environment should be restored after the test to the state before the test, e.g. any changes must be rolled back.

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 tearDown is called by the TESTRUNNER 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.

2.13 TEST SETUP

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

Forces To be able to test a class, there is always some test preparation 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 TESTCLASS test the same production class. Therefore, test preparation is sometimes similar for all TESTMETHODs in a TESTCLASS. To avoid duplicating logic, it should be possible to put common preparation code (e.g. for all TESTMETHODs in a TESTCLASS) in one place.

Solution A setUp-method should be written in the TESTCLASS, which is executed before each 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();
}
```

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 TESTRUNNER.

Typical logic placed in setUp is initializing a database, loading configuration data, and preparing objects for the test.

2.14 TEST TEARDOWN

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

Forces Since usually all TESTMETHODs in a TESTCLASS 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 like freeing resources. Therefore, test cleanup is usually similar for all TESTMETHODs in a TESTCLASS. To avoid duplicating logic, it should be possible to put common cleanup code (e.g. for all TESTMETHODs in a TESTCLASS) in one place.

Solution A tearDown-method should be written in the TESTCLASS, which is executed after each 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();
}
```

Discussion 

The class `junit.framework.TestCase` contains a template-method called `tearDown`. The tearDown-method could be used to DELETE TESTDATA AFTER TEST. The opposite of TEST TEARDOWN is TEST SETUP. The basic execution scheme as described above should be implemented in the TESTRUNNER. It is important to notice that tearDown should be executed by TESTRUNNER even if the test failed.

2.15 STATIC FIXTURE

Problem Where should test preparation and cleanup code be placed, which needs to be executed once for all TESTMETHODS in a TESTCLASS ?

Forces It is important that running the tests does not take too long. If it takes too long, the developers will hesitate to start the tests and the main benefit – immediate feedback – will be lost. If the test fixture is not changed (e.g. read-only access) during the test, it is possible to prepare many testmethods at once. This helps speeding up the test execution.

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

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

Discussion JUnit contains the class `junit.extensions.TestSetup`, 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 abstract class ContractTestSetup extends
junit.extensions.TestSetup{
   // setUp and tearDown as in other test-classes
}

public class ContractTest extend 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.

3 Process Patterns

3.1 Introduction to Process Patterns

The next patterns describe process issues around unit testing.

3.2 RUN OFTEN

Problem How can the benefits of the unit tests be optimized ?

Forces One of the main benefits of having unit tests is immediate feedback [Beck], [Gassmann]. As soon as a change has been made in the system, the tests should be executed. This way it will be immediately clear if a change in the system introduced an unexpected problem.

Solution The tests should be run often. Typically a few times per hour.

Discussion 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. STATIC FIXTURE may be considered to optimize execution time if the same set of testdata may be used by all TESTMETHODs.

3.3 RUN TWICE

Problem How often should the tests be executed in a row ?

Forces Tests should be independent of each other. And it must be possible to execute them many times. The system under test might be changed during test-execution. Data might be stored, 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 Run the tests 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 quick enough that the developer does not have to wait too long.

3.4 RUN AT 100%

Problem How many tests should be successful when integrating ?

Forces Unit tests provide their greatest benefit if 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 thumbs up, thumbs 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, if not all tests are successful ? If failing tests are allowed to be integrated, the tests will loose their value – simple to understand feedback - very soon. They won‘t be trusted anymore. If a developer starts writing code with tests that are not working at 100%, he cannot see anymore whether a test fails because of his changes or if the failure was already there before he started.

Solution All tests (=100%) must be successful when integrating code.

Discussion This is actually important at various levels. The thumbs up indicator serves also as a motivator. Therefore, the more often the thumb is up, the bigger the motivation. This is true on the TESTCLASS level (all TESTMETHODS are successful) and on the ALLTESTS level (all TESTMETHODs in all invoked TESTCLASSes are successful).

3.5 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. But what happens if another developer integrated in the meantime code which does not work together with the code to be integrated ?

Solution The tests should be run immediately before integration, e.g. before committing the changes. They have to run at 100%.

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

3.6 UNIT TEST FIRST

Problem When should the tests be written during development ?

Forces Unit tests may be written either 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 leads sometimes 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. But if the test is written before the production code is written, the motivation will be to make the test run ! And if all tests run, the developer knows he is finished with the task.

Solution The unit tests should be written, if possible, before the production-code.

Discussion More information can be found in [BECK], [WIKI] and [Gassmann].

4 References

- [XP] „Extreme Programming explained, embrace change“ by Kent Beck, ISBN 0-201-61641-6, a book about the concept and philosophy behind XP.
- [FOWLER] „Refactoring, improving the design of existing code“ by Martin Fowler et al., ISBN 0-201-48567-2, the book contains further material on unit testing.
- [JUNIT] „JUnit, Unit testing framework for Java“, by Kent Beck and Erich Gamma, the testing framework for Java and other programming languages can be found under http://www.xprogramming.com/software.htm
- Two articles on JUnit by Kent Beck and Erich Gamma appeared in JavaReport. Both are contained in the distribution of JUnit 2.1 ("JUnit: A Cook's Tour" in Java Report May 1999, "Test Infected: Programmers Love Writing Tests" in Java Report July 1998).
- [WIKI] http://c2.com/cgi/wiki?UnitTests
  This is a page which contains an ongoing discussion on unit testing.
- [Gassmann] "Unit Testing in a Java Project" by Peter Gassmann, January 2000, a paper written for the conference XP2000.
- [Gassmann2] "Development process with VisualAge and vaj2cvs" by Peter Gassmann, February 2000. see http://…