Software testing in extreme programming

Niklas Svensson, dt06ns1@student.lth.se
Jacob Gradén, d01jg@student.lth.se
Lund Institute of Technology
February 15, 2009

Abstract  This article provides an overview of software testing and how it is related to extreme programming. The focus is on well-known test methods, especially black box and white box methods, and on how testing is performed – and could be performed – in XP. The difficulty of using automated measurements of testing to achieve good code coverage is also discussed.
# Table of contents

1. Introduction .......................................................................................................................... 1  
   1.1. Article outline .................................................................................................................. 1  
   1.2. Context ............................................................................................................................... 1  
   1.3. Extreme programming ....................................................................................................... 2  
   1.4. Basic testing terminology ................................................................................................. 3  
2. Types of testing .................................................................................................................... 4  
   2.1. Unit testing ....................................................................................................................... 4  
   2.2. Integration testing ............................................................................................................ 4  
   2.3. System testing ................................................................................................................ 4  
   2.4. Acceptance testing .......................................................................................................... 4  
   2.5. Regression testing .......................................................................................................... 4  
3. Black box testing .................................................................................................................. 5  
   3.1. Equivalence class partitioning .......................................................................................... 5  
   3.2. Boundary value analysis ................................................................................................ 6  
   3.3. Random testing ............................................................................................................... 6  
   3.4. Error guessing ................................................................................................................ 6  
4. White box testing .................................................................................................................. 7  
   4.1. Coverage ........................................................................................................................ 7  
   4.2. Test adequacy ................................................................................................................ 8  
   4.3. Data flow testing ............................................................................................................ 9  
   4.4. Loop testing .................................................................................................................. 9  
   4.5. Reviews ......................................................................................................................... 9  
5. XP and testing ....................................................................................................................... 10  
   5.1. XP versus testing ............................................................................................................ 10  
   5.2. Actual testing in the PVG course .................................................................................... 11  
   5.3. Possible testing in the PVG course ................................................................................ 13  
6. Conclusions .......................................................................................................................... 14  
7. Further reading .................................................................................................................... 15  
References ............................................................................................................................... 16
1. Introduction
This is an article on software testing. Its intended audience is persons with experience of programming, and perhaps especially the extreme programming method (XP), but no formal education on testing. The goal of the article is to present an overview of the most common testing methods, to provide programmers with a general familiarity with the subject of testing, and to describe how XP and testing are interrelated with specific focus on how testing is used in the PVG course 1.

1.1. Article outline
This section provides a background and describes the surrounding environment. Section two deals with how tests can be applied to a software project on different levels of detail. Sections three and four present specific test methods – methodical ways of looking for problems in software. These three sections (two through four) contain fundamental concepts in testing, with links to extreme programming where suitable. Sections five and six contain an analysis of how extreme programming and testing fit together, with special attention to their use in the PVG course, and the conclusions of this analysis. Finally, section seven provides some pointers for finding more information on specific subjects.

1.2. Context
Software development is a complex process with many different phases. Most projects need to advance through at least the following, all of which may be further subdivided:

- Requirements analysis
- Product specification
- Design
- Implementation
- Testing
- Maintenance

It should be noted that testing may be present in all these phases, depending on which development model is being used. For the waterfall model and its derivatives, testing of results is commonly done after each phase is completed; for iterative models, testing is done as a part of each iteration. One model is commonly encountered and deserves to be mentioned, namely the V model.

---

1 This refers to the course EDA260, Programvaruutveckling i grupp (Program development in teams).
Software testing in extreme programming

The V model is an extension on the waterfall model, but the flow is reversed in the middle. The purpose of visualizing the development process in this way, is that it suits testing very well. Requirements, produced at the very start of the process, are the source of acceptance tests, which are performed at the very end. The system design is likewise mirrored in system tests, and so on. For this reason, the V model is often used when testing is discussed on projects which use the waterfall model. An agile approach to the V-model would be to scale it down and repeat the entire cycle in every iteration.

1.3. Extreme programming

Extreme programming (XP) is an agile software development method. XP offers an alternative to rigid processes of software development, such as the waterfall model, and attempts instead to treat changing requirements, priorities etcetera as natural and manage them. [Chromatic]

It is worth observing that XP is a software development method, and does not explicitly provide for testing software in an independent fashion. There are however two important practices in XP regarding testing, namely test-driven development and pair programming.

Test-driven development (TDD) uses tests to drive development, and is therefore not so much a testing method as a development one. Still, it does produce tests. At its core, TDD boils down to two principles [Beck]:

- Write a failing automated test before you write any code
- Remove duplication
Pair programming is the practice of always having two programmers work together on the same code. This is complementary to TDD, since it means code will always be inspected by two persons, increasing the chances of finding mistakes. However, neither TDD nor pair programming, nor both of them combined, is a replacement for normal testing, and software produced by teams using XP therefore still needs to be tested by a dedicated test group in order to uncover mistakes.

1.4. Basic testing terminology

Just like many other areas of technical complexity, testing puts specific meaning on some common English words. Three basic such words are errors, defects and failures:

- **Error**: A mistake, misconception or misunderstanding on the part of the developer.
- **Defects**: Defects (also faults or bugs) are introduced into the code as the result of an error.
- **Failure**: When software cannot perform its intended function within its performance requirements. Failures occur as the result of faults.

This article deals primarily with defects. In order to find and remove defects, however, failures must generally be observed. This is done by comparing the output or behavior of the tested software with that of a test oracle:

- **Test oracle**: A document or software which specifies or produces the expected output of the program being tested — e.g. a design document or the original version of a program being ported.

Since comparing outputs can be tedious, repetitive work and the risk of missing subtle failures — e.g., a semicolon instead of a colon — is large for human testers, this is almost invariably automated, at least to some extent. Some parts of software, for example the graphical user interface, are notoriously hard to test automatically; others, like text processing, are considerably easier. To actually perform testing, testers use test cases, test procedures, tests and test suites:

- **Test case**: A set of test inputs (e.g. numbers for the program to operate on), execution conditions (e.g. a specific state of a database), and expected outputs (e.g. a result file).
- **Test procedure**: A sequence of steps required to execute a specific test case.
- **Test**: A group of related test cases and possibly also test procedures.
- **Test suite**: A group of related tests.
- **Testing**: Performing a test or executing an entire test suite.

Testing is not the same as debugging. Testing aims to find failures, whereas debugging aims to locate and remove the faults behind them.

Thus armed with the most indispensable terms required to understand the domain of testing, let us turn our attention to actual testing.
2. Types of testing

Testing can be performed in a number of different ways to answer different questions and address concerns at different stages of development. The four most important variants are unit testing, integration testing, system testing and acceptance testing.

2.1. Unit testing

A unit test is the smallest possible testable software component. Usually it performs a single cohesive function. A unit is traditionally a method, function or procedure implemented in a procedural programming language. In an object-oriented programming language, both the method and the class could be considered a unit. Because a unit is small in size and simple in function, it is easier to design, execute, record and analyze test results than for larger chunks of code. Defects revealed by a unit test are easy to locate and relatively easy to repair, since a unit test only tests a single unit. XP relies heavily on unit tests because of the test-driven development.

2.2. Integration testing

Several units are combined into a subsystem. The subsystems are tested by integration tests. This is the first time where it can be tested if the units’ interfaces are working correctly with each other; previously they were probably tested with the help of test drivers or stubs (simulated functionality). Integration is normally done with one unit at a time to limit the number of sources containing possible faults.

2.3. System testing

The whole system is assembled from subsystems. The system is tested for compliance with the specified requirements. Like integration testing, this is the first time the subsystems’ interfaces are tested. System testing tries to detect defects both by looking at the interfaces and by looking at the system as a whole.

2.4. Acceptance testing

Acceptance testing (also known as user acceptance testing) tests that the customer’s expectations of requirements on the functionality are fulfilled. This is a formal testing conducted to determine whether the system satisfies the customer’s criteria, and is most commonly performed on an almost complete product. Because the customer is seldom a testing expert, these tests are often created by the test organization of the producing software from use cases described by the customer. When user acceptance tests have been completed, the software is often considered to be finished and deliverable. XP normally uses acceptance tests for each story, to be sure that the story has been correctly implemented.

2.5. Regression testing

Regression testing is any type of testing done to make sure that recently implemented functionality or modification did not break anything which worked earlier. It is usually done by re-running some of the earlier test cases. One common tool which can be used for regression testing with the object-oriented programming language Java is called JUnit (http://www.junit.org). Regression testing is especially important when following the XP development method, because the code often changes quite drastically.
3. Black box testing

Black box testing is a method where software is tested without knowledge of its internal behavior. When using this method, the program is viewed as a black box and the tester is completely unconcerned with its internal structure. The legal inputs and their matching supposed output is known from the specifications, but no information about how the program reaches that output is available. The testing can be started as soon as the specifications are done and can (and normally should) be done independently of the programmer.

It should be noted that we can only be sure that the program we are testing appears to be working according to the specifications. If we were to make certain that the program worked according to the specifications, we would have to try every possible input and that is not doable in all cases; for example when the current input is dependent on previously given inputs. The result of this is that a program cannot be tested to guarantee that it is error free. Instead, the goal is to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases. [Myers, Beizer]

Some of the advantages of using black box testing are:

- The developer and tester are independent of each other, so the testing is not affected by any biased programmer.
- The tester functions as a user who can give feedback about the experience with the program.
- The tester does not have to know how the system works internally.
- Test cases can be implemented as soon as the specifications are done.

The four most important black box techniques are equivalence class partitioning, boundary value analysis, random testing and error guessing.

3.1. Equivalence class partitioning

As previously noted, it is in practice impossible to test every possible input. The aim is therefore to try to maximize the number of errors found with a limited number of test cases. The equivalence class partitioning technique has two considerations. The first is that each test case should cover as many different requirements on the software as possible, in order to minimize the total number of test cases. The second is that the input domain should be partitioned into as few equivalence classes as possible.

An equivalence class is a set of inputs where a test of a value from an equivalence class is equivalent to a test of any other value in the same class. If one test case in an equivalence class detects a defect, all other test cases in the equivalence class would be expected to find the same defect. If a test case did not detect a defect, it would be expected that no other test case in the equivalence class would detect it.

Because the inputs within an equivalence class are considered equal, each equivalence class needs to be tested only once. A single test case can cover several equivalence classes. Introducing equivalence classes can drastically reduce the number of test cases needed. It should be noted that equivalence class partitions can overlap. There is always a valid and an invalid partition, but there can exist more. It is im-
important to remember to test the invalid partitions. If 1-10 is the valid input, then <= 0 is an invalid partition, 1-10 is a valid partition, and >= 11 is another invalid partition.

One issue with equivalence class partitioning is how to test multiple variables simultaneously, especially when they may possible influence each other. The naïve approach is to build the Cartesian product of all possible values for all possible variables, and [Whittaker] notes that all such combinations must at least be considered. [Beizer] argues, however, that this works only when variables have well-defined upper and lower values, and that ordinary analysis of the input domain normally suffices.

### 3.2. Boundary value analysis

Boundary value analysis is a technique that focuses on the boundary areas of a program’s input domain. Test cases that explore boundary conditions have a higher payoff than test cases that do not. Boundary conditions are the situations directly on, above and beneath the edges of equivalence classes. Boundary value analysis differs from equivalence class partitioning in two respects. First, instead of selecting any element in an equivalence class, boundary value analysis requires that the boundary conditions are selected. Second, instead of just focusing on the input equivalence classes, boundary value analysis also focuses on output equivalence classes.

If the valid input equivalence class is 1-100, the tested values would be 0, 1, 2, 99, 100, and 101. If there exist any output equivalence classes, their boundaries should be tested in the same way. It should be noted that the boundaries of the input equivalence class do not always represent the boundaries of the output equivalence class (think of the trigonometric function sine). When dealing with a certain maximum number, e.g. only one object is allowed somewhere, try to add one more than the maximum allowed value. “**Boundary-value analysis, if practiced correctly, is one of the most useful test-case-design methods. However, it often is used ineffectively because the technique, on the surface, sounds simple.**” [Myers p. 65]

### 3.3. Random testing

Random testing (also known as fuzz testing) is a black box testing technique where the input to a program is selected at random. Normally the tester enters a range of inputs to the random value generator, which saves the generated data to know what input caused the failure. Random testing requires less effort and time compared to using other testing techniques. However, it should be noted that many testing experts consider random testing to have a very little chance to produce effective test data [Myers p. 43]. There are cases where random testing detects faults which human testers would fail to find, but in many cases random testing only proves that the software can handle strange input without crashing.

### 3.4. Error guessing

“It has often been noted that some people seem to be naturally adept at program testing. Without using any particular methodology [...] these people seem to have a knack for sniffing out errors.” [Myers p. 88] The people who seem to be naturally adept at program testing are most often using a technique called
method of error guessing, often without even knowing it. Error guessing is based on intuition, past experiences and knowledge. With these assets they guess certain probable types of errors and then write test cases to expose those errors. The tester normally writes down possible errors and error-prone situations on a list, which he later writes test cases for. If the value zero is a valid input in a program, for example, the tester recognizes it as an error-prone situation and therefore writes test cases specifically for the input value zero.

4. White box testing

Unlike black box testing, white box testing (also known as glass box testing) allows the tester access to the internal mechanisms of the unit to be tested. The tester looks at the code and uses the extra information thus gained to make sure that the software works as intended. The goal is generally to find defects in the specific implementation, which may catch obscure errors not detected by black box testing.

This allows for much more detailed testing, but also means that testing is dictated by the design of the code. Because of the increase in details, white box testing is primarily suitable for smaller components and components where verification is particularly important. Also, the code must of course be available, which means that white box testing is impossible for programs where only the binary is distributed.

A very important aspect of white box testing is code coverage, and the related notion of test adequacy. It should however be noted that all methods which directly utilize the source code can be considered white box testing – for example 4.5. Reviews, even though reviews are sometimes placed in a category of their own.

4.1. Coverage

Coverage is the amount of source code which is exercised by a test or a test set. It is usually expressed as a percentage and qualified by a coverage criterion. A coverage criterion identifies how the coverage is measured – for example, “100% line coverage” simply means that each line of code has been executed at least once. There are several criteria which can be applied to measure coverage, and they are commonly used together to provide a complete picture. Some are more extensive than others and therefore, by definition, require that the less extensive criteria are also fulfilled.

Line coverage, also known as statement coverage, indicates that each line has been exercised at least once. This is a very weak criterion, since it may mean only a single pass in a loop, or entering a simple branch but never skipping it (for example an if-statement without an accompanying else).

Branch coverage, also known as decision coverage or complete coverage, which is a horrible misnomer, requires that not only has each line been exercised, but also that all outcomes of all branches have been tested. This eliminates the problem mentioned above, but does not necessarily mean that all branch logic has been exercised adequately. This, and all other criteria below, requires that all possible entry and exit points of the tested software (for example main methods and exit() calls) have been exercised.


*Condition coverage* comes in several different flavors. In its most basic form, it requires that each condition in each branch statement has taken on all its possible outcomes. In the statement "if(a or b)", this would mean that a has been both true and false, and the same for b. There is however no requirement that a and b must both have been false at the same time or similar, which means that it is possible to have complete condition coverage without achieving branch coverage – since not all outcomes of the branch need to have been tested.

When each condition takes on all possible values, and each branch has resulted in all its different outcomes, this is called *condition/decision coverage*, and is stronger than condition coverage. This is related to *multiple condition coverage*, which requires all conditions in a statement to take on all of their possible values – essentially testing every possible combination of conditions.

Condition/decision coverage and multiple condition coverage are sometimes used interchangeably, and sometimes they are simply called condition coverage, even though they are very different in strength.

Along a somewhat different line, *function coverage* (also called *method coverage* or *procedure coverage* depending on the notation used for different programming languages) requires that each function in a program has been exercised. Function coverage is often used together with one of the first three criteria, to say for example that “program X has been tested with 100% function coverage, and 95% branch coverage within the functions”.

### 4.2. Test adequacy

Test adequacy is a stopping criterion, which decides when sufficient testing has been done. Adequacy criteria also help the tester to decide which properties of a program to focus on, which data set to choose for testing, and how to quantify the progress of testing. Adequacy is typically defined in terms of coverage; for example: A *test set is adequate for program X if it causes all branches to be executed*, or A *test set is adequate for program Y if it exercises all possible code paths from start to end*. See section 4.1. Coverage above for more information.

Adequacy is a measure of how well the software has been tested. This is however complicated by the fact that there are a number of different ways in which failures can occur – apart from the fact that finding and correcting defects can be very hard. Whittaker describes four different categories of circumstances which may allow defects to slip through the testing process: [Whittaker, p. 1]

- The user executed untested code
- The order in which statements were executed in actual use differed from that during testing
- The user applied a combination of untested input values
- The user’s operating environment was never tested

Conventional adequacy criteria, as presented in the sections above, can typically mitigate the risks of the first problem, and to some extent also the second. The third problem is best handled by black box techniques such as Equivalence class partitioning, and test adequacy is not relevant for black box tech-
niques. Handling adequacy for the fourth problem requires that coverage is extended to define different environments for which testing must be performed. This is generally not done, except in situations where the requirements clearly define such environments, since it is costly to install and maintain additional environments (called test beds) throughout the software development cycle. [Whittaker]

4.3. Data flow testing

Data flow testing focuses on variables and their three different uses: def for definition (x = 2), p-use for predicate use (if(x == 2)), and c-use for computation use (return(x + 2)). Coverage can then be measured in terms of testing all defs of all variables, or all p-uses, and so on. The strongest criteria is called all def-use paths, and requires both that all variables which are defined are used, and that all variables which are used have been defined. For some languages, these tests are performed by the compiler; for others, they can show up as subtle failures during execution.

4.4. Loop testing

Loop testing aims to exercise all loops in the software, with the rationale that they are common sources of error. Since loops are ultimately represented by some kind of branching, this is related to branch coverage (see section 4.1. Coverage), and loop testing is sometimes said to be a subcategory of branch testing, which would be more general and include alternate execution paths through a software — for example, if-then-else statements.

Depending on the language, loops can be more or less complicated — in low-level languages which allow spaghetti code, they can be downright horrible. For more structured languages, like Java, loops come in two main versions: simple and nested, where nested loops are loops inside other loops. For both types, the loops should be exercised a number of times to make sure that they work as intended for different number of iterations. Typically, the number of iterations tested should be zero, one, two, the maximum number of iterations minus one, maximum, maximum plus one, and possibly some “typical” value roughly in the middle of the interval.

Nested loops need to be tested for all combinations of these numbers of iteration, for each loop — that is, while the outer loop does only one iteration, the inner loop should be tested with everything from zero to maximum plus one iterations. The outer loop should then do two iterations, and the inner loop again everything from zero to maximum plus one; and so on.

4.5. Reviews

Reviews are a group of static test methods, which means that no code is executed. Instead, the code is inspected by human eyes. There are several different kinds of reviews, and they can be roughly divided into either formal reviews (also called Fagan reviews) or informal reviews. The former are strictly specified processes which are often used together with traditional software development models, whereas

---

2 Adequacy analysis can be applied to inputs, but requires other definitions of coverage than those presented here.

3 There are other uses, such as undefined and dead, but these are not relevant here.
the latter are intended to require less resources and planning. Pair programming is an example of an informal review technique.

There are other ways of categorizing reviews, such as whether the purpose is managerial (finding the status of the project) or technical (finding defects). Pair programming would then be a technical review – to be precise, an informal technical review. Other review methods are code inspections and walkthroughs, both of which are formal technical reviews.

5. XP and testing

For testing purposes, the two most important practices in XP are test-driven development and pair programming.

Test-driven development produces tests, primarily unit tests which can be used as regression tests (see sections 2.1. Unit testing and 2.5. Regression testing). In terms of testing, TDD’s contribution is that it obliges the programmer to write test cases and structures to support these test cases (such a structure is called a test harness) and that it provides regression testing. These test cases are almost always on the level of unit testing, where a unit is of the size of a method or class.

TDD can be viewed as either black box or white box testing. Test cases are written before any implementation is done and are actually used to specify the interface of the code to be written; this amounts to black box testing. The programmer does however have access to the entire code, and when new tests are introduced in modules where code already exists, white box testing is possible. When used strictly as intended, however, TDD is more like black box than white box testing, since tests are written to specifications and are written before new code is added.

Pair programming contributes to testing in that it amounts to a review (see section 4.5. Reviews) of the code or document produced. This is clearly a white box technique, and is certainly one of XP’s strengths over traditional programming in terms of testing, but a drawback is that the review is performed by developers – optimally, a different set of eyes should be involved to maximize the probability that faults will be detected.

XP also recommends automating testing and using a good framework [Chromatic]. Some professional frameworks include the functionality to analyze coverage, which is laudable and does increase the chance that defects will be found; but if coverage is reported in terms of line coverage, it is easy to be lulled into a false sense of security. Remember that line coverage is very weak, and that optimally, some kind of condition coverage should be employed to achieve proper code coverage.

5.1. XP versus testing

There seems to be a discrepancy between how XP describes testing, and how test researchers and authors of books on testing describe it. XP appears to take the view that test-first, automated testing and pair programming is enough to be confident in the software produced, whereas authors from the testing domain maintain that testing is more complex and complicated than this; that testing must be per-
formed by a group separate from the developers producing the software; and that testing is a discipline of its own.

Apart from the politics of how to subdivide the process of producing stable software, is there any real opposition between the two views? The desired end result is certainly the same – feature-complete software which performs as expected without bugs or other nasty surprises. Perhaps the testing area needs to be brought up to speed with agile methods and how testing for them differs from testing for traditional methods; or perhaps agile practitioners need to accept that formal testing is still required and should be performed after each iteration.

[Talby] et al make a distinction between adopting agile development methods and adopting agile quality assurance practices, and note that the latter is slower and harder. Their suggestion is to include test experts in the XP team – to ensure that the test cases are of high quality – and to encourage them to talk to developers. They also argue against the traditional view held by testers that tests must be independent of programmers, saying that “The counterargument isn’t that these claims are false but simply that the alternative is better overall” [Talby, p. 33]. [Hendrickson] agrees, pointing specifically to the fact that even seasoned XP programmers are bested at finding defects by a good tester and that agile teams therefore benefit greatly from the inclusion of test specialists.

[Talby] et al also note that testing is a common bottleneck in software development and that this problem is especially large for XP projects, since tests are accumulated with each iteration. This means that if programming proceeds at a constant pace, the time required for testing will increase linearly with each new iteration. To handle this problem, they suggest asking developers to spend more time writing tests and performing actual testing, which will both lower the pace at which software is produced and increase the available resources devoted to testing. Development and testing can thus be kept in balance at a reasonable level.

5.2. Actual testing in the PVG course

In practice, pair programming works very well on the PVG course, so reviewing can probably be trusted to reach an acceptable level. This is however a static technique – no code is executed – and the situation is a bit different for dynamic techniques.

If TDD as a discipline is enough to provide a high degree of confidence in the software, which is really what testing is all about too, then the next question becomes whether it is in practice used in a suitable manner by the teams. This is an issue on which there is some focus in the course, although the intensity of that focus varies between teams. It is also an issue for which it is hard to find concrete facts – the number of tests, or the code coverage, might be indicators of how well the software is tested, but it is very easy to write many tests, or to achieve a high degree of coverage, without actually writing good tests. A few statistics are presented in Table 1 below.

The number of tests is reported by JUnit. All forms of coverage are measured in percent – a branch coverage of 50%, for example, means that 50% of all branch outcomes have been tested (see section 4.1. Coverage). The three different forms of line coverage are defined as follows:
1. Total line coverage, as reported by EclEmma (http://www.eclemma.org/). This, unfortunately, includes the code in the JUnit test cases. Since most of the code in test cases is of course executed when tests are executed, this means that the coverage is reported as being higher than it actually is. For next year, EclEmma should probably be replaced by some other supporting tool for coverage.

2. Recalculated line coverage, as calculated manually from EclEmma’s report of how many lines of code were executed in the different packages. By adding up the number of tested lines for production code only, not for test cases, and dividing by the appropriate sum, a more reasonable coverage is derived. As seen in the table, these values are consistently lower than (1).

3. Total line coverage, as reported by CodeCover (http://codecover.org/). CodeCover requires the user to select which classes or packages to perform coverage analysis for, which means that even when JUnit tests are run, coverage is reported only for the actual software to be tested, not the unit tests. This is even lower than the recalculated coverage, because CodeCover does not consider conditional statements (such as if) or looping constructs (such as while) for line coverage; these are presented under branch coverage and/or loop coverage. CodeCover would be a good alternative to EclEmma.

<table>
<thead>
<tr>
<th>Team number</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of tests</td>
<td>93</td>
<td>103</td>
<td>75</td>
<td>131</td>
<td>128</td>
<td>67</td>
<td>101</td>
<td>148</td>
<td>81</td>
<td>103.0</td>
</tr>
<tr>
<td>Line coverage (1)</td>
<td>88.0</td>
<td>74.5</td>
<td>81.7</td>
<td>69.9</td>
<td>95.4</td>
<td>81.4</td>
<td>88.5</td>
<td>64.5</td>
<td>87.2</td>
<td>81.2</td>
</tr>
<tr>
<td>Line coverage (2)</td>
<td>83.8</td>
<td>68.6</td>
<td>68.9</td>
<td>46.5</td>
<td>93.9</td>
<td>73.1</td>
<td>79.5</td>
<td>44.7</td>
<td>82.3</td>
<td>71.3</td>
</tr>
<tr>
<td>Line coverage (3)</td>
<td>77.2</td>
<td>60.3</td>
<td>62.5</td>
<td>38.0</td>
<td>93.0</td>
<td>66.9</td>
<td>74.7</td>
<td>39.2</td>
<td>81.0</td>
<td>65.9</td>
</tr>
<tr>
<td>Branch coverage</td>
<td>79.4</td>
<td>53.1</td>
<td>58.9</td>
<td>35.8</td>
<td>89.7</td>
<td>74.2</td>
<td>65.1</td>
<td>42.9</td>
<td>69.7</td>
<td>63.2</td>
</tr>
<tr>
<td>Loop coverage</td>
<td>49.3</td>
<td>38.8</td>
<td>47.4</td>
<td>38.8</td>
<td>63.7</td>
<td>46.4</td>
<td>48.1</td>
<td>42.1</td>
<td>44.8</td>
<td>46.6</td>
</tr>
<tr>
<td>Condition coverage</td>
<td>75.0</td>
<td>54.4</td>
<td>54.1</td>
<td>43.4</td>
<td>85.2</td>
<td>70.6</td>
<td>59.4</td>
<td>43.7</td>
<td>66.4</td>
<td>61.4</td>
</tr>
<tr>
<td>Errors</td>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Failures</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Acceptance</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Branch and condition coverage are reported by CodeCover and are the same as presented in section 4.1. Coverage; the only point to note is that CodeCover performs multiple condition coverage, the strongest coverage version. Loop coverage means that each loop must have been executed zero, one, and more than one times (CodeCover does not know what the maximum number of loops is, so that is not tested).

Errors and failures are the numbers reported by JUnit. An error occurs when the tested code throws an uncaught exception, and a failure occurs when the software exits normally but produces the wrong results. Acceptance, finally, is the number of errors or failures caused by the acceptance tests.
Figure 2: Number of test cases vs. line coverage (found in the Appendix) illustrates that as the number of test cases rises (bolded series), the coverage remains unaffected at best (upper three series). In fact, the regression line for line coverage (3) divided by number of test cases is negative (lower two series). Although a full statistical analysis would be necessary to be able to draw certain conclusions, the implication is that quantity of test cases adversely affects their quality and that more test cases may in fact be a bad thing.

Figure 3: Coverage relationships indicate that the different measures of coverage are strongly related to each other, which is to be expected. It also indicates that line coverage is generally more complete than branch coverage, which in turn is more complete than condition coverage. This is to be expected, since that is the order in which the measures are found, if sorted by strictness. Somewhat surprisingly, good loop coverage turns out to be much harder to attain than the other three, which is unfortunate since loops are common sources of off-by-one-errors.

It is worth noting that the old adage “what gets measured gets done” holds true. Measuring the number of tests is very easy, and with the help of tools such as EclEmma, so is total line coverage. Clearly, however, neither is a be-all, end-all measurement for test quality – which, in turn, is an indicator of code quality. If any measure should be used, it would be prudent to select condition coverage, possibly in conjunction with loop coverage, since these are the most stringent. Code with good ratings for those measurements would by necessity also have good ratings for the other, weaker, measurements.

Table 1 above also indicates that acceptance tests are an important source of failing tests. It is worth noting that in addition to the numbers above, a preliminary overview performed during the coding day of iteration 5 showed more teams with errors or failures, the majority of which were caused by acceptance tests. Some ideas for remedying this are presented at the end of section 5.3 below.

5.3. Possible testing in the PVG course

If TDD should turn out not to be enough, how could software testing be added to the current structure of the course? Testing is not really a prioritized area in the XP course, until the last iterations where stories regarding stability show up. There is however, as noted, a great deal of focus on test-driven development, which could possibly be extended to use tests not only for design purposes but also to introduce actual testing of the software after it has been built.

One possibility could be a recurring spike for each team from the coding day to the planning game, which aimed at testing system robustness using black-box techniques, and another, similar, spike with the same aim but using white-box techniques. Ideally, though, testing should be performed by a different group than the programmers who wrote the code. Testing could then possibly be performed by members of another group; or even by students on a testing course.

As with testing, there is little focus on requirements in the course, which means that test methods which rely on requirements are hard to use. The planning game results in a loose specification of what needs to be done, namely a specification of the program’s features, but there are no formal requirements stating which inputs the program should accept, performance requirements, etcetera. The closest approxi-
Software testing in extreme programming

Information is the acceptance tests for different stories, but in order for them to be useful they would have to be improved and extended.

There are currently small imperfections in the acceptance tests (e.g. typos and incorrect calculations), and after the initial iterations they are only available for batch testing of several completed stories. It would mean a lot of work, but if the defects were corrected and a complete combinatorial set of the tests were produced – that is, one acceptance test for each combination of stories: only S1; S1 and S2; S1 and S3; S1 and S2 and S3; and so on) – these tests could very well be used both as functional requirements and, of course, as tests of the functional requirements.

This would probably have to be performed outside the XP course for reasons of time – perhaps as a study on the coaching course, by students on some other course, or by staff. If done as a study, statistics could be gathered for all teams over time, and towards the end, a suitable graph could be created, indicating which acceptance tests make each other obsolete. Even if the test data is not rewritten or combinatorial sets produced, this would make them easier to use and reduce the number of failures.

6. Conclusions

This section aims to very briefly summarize the key conclusions from section 5. XP and testing.

From a testing standpoint, the two most important practices of XP are test-driven development, which amounts to black-box testing, and pair programming, which amounts to reviews. Using a testing framework may also help in analyzing coverage, but beware: metrics may be misleading.

There appears to be an ongoing discourse regarding whether test specialists should be integrated into agile teams at all, and if so, how it should be done. Proponents of XP argue that the twelve practices are enough to ensure high quality; proponents of traditional testing argue that testing needs to be performed as a separate activity by persons who are not engaged in writing code. A possible middle ground is to enlist test specialists into the team, which enables finding defects earlier and avoids the problem of testing becoming a bottleneck.

Measuring code quality is hard and using metrics which are easy to come by – such as number of test cases or percent code coverage – may be tempting. Good results for these are however by no means a guarantee for high code quality. Furthermore, the number of test cases shows signs of being negatively correlated with code coverage; and measuring line coverage is a poor indicator of how well the code is actually tested, since it is a very weak coverage criterion.

If any sort of measurement should be used, number of test cases appears to be an inadequate choice. Condition coverage, ideally in conjunction with loop coverage or other strong coverage criteria would be a better choice. Strong coverage criteria are especially useful because fulfilling them by definition means good ratings for weaker criteria, such as line coverage.

Possibly limited in scope to allow for only “sensible” combinations of stories, in order to lessen the combinatorial explosion of cases which would otherwise ensue.
EclEmma, specifically, which has been popular among teams during this year’s course, incorrectly includes the number of lines in the test cases in its calculations of line coverage – only lines in the tested software should be counted. For this reason, many projects appear to have higher coverage than is actually the case. CodeCover appears to be a better candidate for tool support.

Many of the problems different teams experience with testing appear to be related to the acceptance tests. This may simply be because acceptance tests are, by design, large and test much of the software’s functionality; or it may be related to the fact that there are many small errors in the files and that it is hard to know when to run which acceptance tests. It would be helpful if these errors were rectified, and it would also be very helpful if some kind of overview could be presented, depicting how the acceptance tests relate to the different stories and each other.

7. Further reading

This section aims to provide the reader with details regarding where to find more information on different subjects. On a general note, [Burnstein], [Kaner] and [Myers] are large and informative enough to provide both a great deal of background to testing and details on many specific topics, and any of them is a good choice for a more thorough presentation of the area than has been accomplished here.

For more information regarding the Context within which testing is performed, both [Burnstein] and [Kaner] are good choices. Apart from its focus on testing as a process and the environment which surrounds testing, [Burnstein] also presents the related Testing Maturity Model which attempts to formalize different levels of organizational awareness regarding testing.

The different Types of testing are presented especially clearly in [Burnstein], and a few others are also mentioned. Black box testing is explained in great detail in [Beizer], a book which is considered a very important work regarding testing and general and black box techniques in particular. [Myers] complements the picture, and also identifies some of the weak points of these techniques. White box testing is covered very well in both [Burnstein] and [Kaner], and [Myers] has some interesting discussions on this topic too.

Regarding Extreme programming, [Chromatic] is a beginner’s guide to extreme programming and is a very good source for more information about XP itself and its practices, including test-driven development, pair programming etcetera. [Beck] gives a great deal of information on test-driven development, but rather as a tool for software development than as a testing method.

The different measurements performed by CodeCover, finally, are explained on that product’s homepage: [http://codecover.org/documentation/references/javaMeasurement.html](http://codecover.org/documentation/references/javaMeasurement.html).
References


Appendix 2: Test statistics in the PVG course

This appendix presents graphical interpretations of Table 1: Coverage statistics for Junit tests after iteration 5 on Page 12.

Figure 2: Number of test cases vs. line coverage

Regression equation:

\[ y = -8.9265x + 114.28 \]

\[ R^2 = 0.8152 \]
Figure 3: Coverage Relationships

- Line coverage
- Branch coverage
- Condition coverage
- Loop coverage

Team number: 1, 2, 3, 4, 5, 6, 7, 8, 9

Percent coverage: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100