8

On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension

Leon Moonen[1], Arie van Deursen[1,2], Andy Zaidman[1], and Magiel Bruntink[2,1]

[1] Delft University of Technology, The Netherlands
[2] CWI, The Netherlands

Summary. We know software evolution to be inevitable if the system is to survive in the long-term. Equally well-understood is the necessity of having a good test suite available in order to (1) ensure the quality of the current state of the software system and (2) to ease future change. In that light, this chapter explores the interplay that exists between software testing and software evolution, because as tests ease software evolution by offering a safety net against unwanted change, they can equally be experienced as a burden because they are subject to the very same forces of software evolution themselves.

In particular, in this chapter, we describe how typical refactorings of production code can invalidate tests, how test code can (structurally) be improved by applying specialized test refactorings. Building upon these concepts, we introduce "test-driven refactoring", or refactorings of production code that are induced by the (re)structuring of the tests. We also report on typical source code design metrics that can serve as indicators for testability. To conclude, we present a research agenda that contains pointers to—as yet—unexplored research topics in the domain of testing.

8.1 Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less satisfactory [317, 321]. We also know that due to ever changing surroundings, new business needs, new regulations and also due to the people working with the system, the software is in a semi-permanent state of flux [319]. Combined with the increasing life-span of most software systems [56], this leads to a situation where an ever higher fraction of the total budget of a software system is spent during the maintenance or evolution phase of a software system, considerably outweighing the initial development costs of a system [329].

For many people, evolving a software system has become a synonym for adapting the source code as this concept stands central when thinking of software. Software, however, is multidimensional, and so is the development process behind it. This multidimensionality lies in the fact that to develop high-quality source code, other artifacts are needed. Examples of these are: specifications, which are needed

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

to know what should be developed, constraints, which are defined so that the software has to adhere to them, documentation, which needs to be written to ease future evolution, and tests, which need to be set up and exercised to ensure quality [436]. The central question then is how evolution should happen: in a unidimensional way, where only the source code is changed, or in a multidimensional way, where (all) the other artifacts are also evolved?

Within this chapter we will explore two dimensions of the multidimensional software evolution space, as we will focus on how the production software evolves with regard to the accompanying tests of the software system. To characterize why tests are so important during evolution, we first discuss some general focal points of tests:

Quality assurance Tests are typically engineered and run to ensure the quality of a software system [131]. Other facets that are frequently tested are the robustness and stress-resistance of a software system.

Documentation In *agile software development* methods such as *extreme programming* (XP), tests are explicitly used as a form of documentation, and as such, the tests serve as a means of communication between developers [516, 149].

Confidence At a more psychological level, test code can help the software (re-) engineer become more confident, because of the safety net that is provided by the tests. Furthermore, the confidence within the development team can be improved when they see that the system they are trying to deliver, is working correctly [119, 149].

An aspect of testing that cannot be neglected is the impact on the software development process: testing is known to be very time-intensive, thus driving up the total costs of the software system. Estimates by Brooks put the total time devoted to testing at 50% of the total allocated time [85, 447], while Kung et al. suggest that 40 to 80% of the development costs of building software is spent in the testing phase [301].

Several types of testing activities can be distinguished. The focus of this chapter is on *developer testing* (often also called *unit testing*), i.e., testing as conducted by the development team in order to assess that the system that is being built is working properly. In some cases, such tests will be set up with knowledge of the inner workings of the system (*white box* testing)—in others the test case will be based on component requirements, (design) models or public interfaces (black box testing) [66, 346].

One of the alternatives to developer testing is *acceptance testing*, i.e., testing as conducted by end user representatives in order to determine whether the system meets the stated criteria. Although acceptance testing is not the primary focus of this chapter, it has many techniques in common with developer testing (as observed by Binder [66]), which is why we believe that the results that we discuss will to a large extent be valid for acceptance testing as well.

Having discussed the necessity of a software system's evolution and also the importance of having a test suite available for a system, we can turn our attention to the interactions that occur between tests and the system under evolution. To this end, we define a number of research questions that we will investigate in the remainder of this chapter:

1. How does a system's test suite influence the program comprehension process of a software engineer trying to understand a given system? What are the possible side effects with regard to evolving the software system?
2. Are there typical code characteristics that indicate which test code resists evolution? And if so, how can we help alleviate these, so called, *test smells*?
3. Given that production code evolves through e.g. refactorings—behavior preserving changes—, what is the influence of these refactorings on the associated test code? Does that test code need to be refactored as well or can it remain in place unadapted? And what will happen to its role as safety net against errors?
4. Can we use metrics to understand the relation between test code and production code? In particular, can object-oriented metrics on the production code be used to predict key properties of the test code?

In order to find answers to the above questions, we have studied how the test suites of a number of applications evolve through time. We have specifically looked at software developed using agile software development methods since these methods explicitly include a number of evolutionary steps in their development process. Furthermore, such projects typically make use of testing frameworks, such as JUnit [49, 262]. To sketch this context, we give a short introduction to agile methods in Section 8.2.

The four research questions introduced above, are discussed in Sections 8.3 through 8.6: we investigate the effects of test suites on comprehension in Section 8.3. We present a catalogue of test smells and test refactorings in Section 8.4. In Section 8.5 we make a classification of classical refactorings [183] into categories, so that one can easily see which refactorings (possibly) break a test. Finally, we discuss a study that shows how certain object-oriented metrics correlate to testing effort in Section 8.6.

In our concluding remarks (Section 8.7) we present a retrospective and touch upon a number of unexplored research tracks.

8.2 Agile Software Development Methods

Agile software development methods (or Agile methods in short) refer to a collection of "lightweight" software development methodologies that adhere to the ideas in the Agile Manifesto [233]. Agile methods aim at minimizing risk and achieving customer satisfaction through a short (development) feedback loop.

Agile methods recognize that continuous change of software systems is natural, inevitable and actually a desirable aspect of successful software systems. Agile software development is typically done in short iterations, lasting only a few weeks. Each iteration includes all software engineering activities, such as planning, design, coding, and testing, that are needed to add a (small) piece of functionality to the system. Agile methods aim at having a working product (albeit not functionally complete) deliverable to the customer after each iteration.

Agile software development builds upon various existing and common sense practices and principles, such as code reviewing, testing, designing and refactoring. However, these practices are done *continuously* rather than at dedicated phases of the software process only. On the other hand, the need for extensive documentation on an agile project is reduced by several of its practices: test-driven development and a focus on acceptance testing ensures that there is always a test suite that shows that your system works and fulfills the requirements implemented to that point. For the developers, these tests act as significant documentation because it shows how the code actually works [9], and how it should be invoked.

A particular agile method that is studied in more detail in this chapter is *Extreme Programming* (XP). XP is one of the initial and most prominent of the agile methods and applies many of the agile practices to "extreme levels". It is a lightweight methodology for small teams of approximately 10 people developing software in the face of vague or rapidly changing requirements [50]. XP is performed in short *iterations*, which are grouped into larger *releases*. The planning process is depicted as a game in which business and development determine the scope of releases and iterations. The customer describes features via *user stories*, informal use cases that fit on an index card. The developers estimate each of the user stories. User stories are the starting point for the planning, design, implementation, and acceptance test activities conducted in XP.

Two key practices of XP play an important role within the scope of our study, namely testing and refactoring. In XP (and most other agile methods) *tests* are written in parallel with (or even *before*) the production code by the programmers. The tests are collected and they must all pass at any time. Customers write acceptance tests for the stories in an iteration, if needed supported by the development team. Tests are typically fully automatic, making it cheap to run them frequently. To write tests, testing frameworks such as JUnit [49] are used (see the next section).

The second key practice of interest is *refactoring*: improving the design of existing code without changing functionality. The guiding design principle is "do the simplest thing that could possibly work". In XP, continuous refactoring during coding replaces the traditional (big) up front design effort.

Note that although this chapter uses agile software development methods and XP to discuss the interaction between software evolution and software testing, this does not mean that the issues observed only apply to agile methods; they are just as likely to come up in any other development process where developer testing and refactoring plays an important role. We choose agile methods as showcase because of its explicit focus on testing and inclusion of evolutionary steps in the development cycle.

8.3 Program Comprehension

A major cost factor in the life cycle of a software system is *program understanding*: trying to understand an existing software system for the purpose of planning, designing, implementing, and testing changes. Estimates put the total cost of the understanding phase at 50% of the total effort [125]. This suggests that paying attention to program comprehension issues in the software process could well pay off in terms of higher quality, longer life time, fewer defects, lower costs, and higher job satisfaction.

This is especially true in the case of extreme programming since the need for people to understand pieces of code is at the very core of XP.

Based upon a thorough analysis of (1) literature on XP [50, 254, 48]; (2) online discussion covering XP subjects[3]; and (3) our own experiences made during an ongoing (industrial) extreme programming project[4], we made the following observation:

Observation 1 An extensive test suite can stimulate the program comprehension process, especially in the light of continuously evolving software.

For our study, we specifically focus on how program comprehension and unit tests interact in the XP software process. We analyze risks and opportunities, look at the effect on the team (whether and how the team gets a better understanding of the code) as well as on the source code (whether and how the code gets more understandable).

8.3.1 Program Understanding

We define program understanding (comprehension) as *the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution, and re-engineering purposes* [383].

An important research area in program understanding deals with the cognitive processes involved in constructing a mental model of the software system (see, e.g., [530]). A common element of such cognitive models is generating hypotheses about code and investigating whether they hold or must be rejected. Several *strategies* can be used to arrive at relevant hypotheses, such as bottom up (starting from code), top down (starting from a high-level goal and expectations), and opportunistic combinations of the two [125]. Strategies guide two understanding mechanisms that produce information: *chunking* creates new, higher level abstraction structures from lower level structures, and *cross referencing* relates different abstraction levels [530]. We will see how the XP practices relate to these program understanding theories.

The construction of mental models at different levels of abstraction can be supported by so called software exploration tools [378]. These tools use reverse engineering techniques to (1) identify a system's components and interrelationships; and (2) create representations of a system in other forms or at higher levels of abstraction [112].

8.3.2 Unit Testing and XP

Unit testing is at the heart of XP. Unit tests are written by the developers, using the same programming language used to build the system itself. Tests are small, take a white box view on the code, and include a check on the correctness of the

[3] Most notably, the *C2 wiki* at http://www.c2.com/cgi/wiki and http://groups.yahoo.com/group/extremeprogramming/. Last visited January, 2007.

[4] Program understanding tools by the *Software Improvement Group*: http://www.software-improvers.com/.

results obtained, comparing actual results with expected ones. Tests are an explicit part of the code, they are put under revision control, and all tests are shared by the development team (any one can invoke any test). A unit test is required to run in almost zero time. This makes it possible (and recommended) to run all tests before and after any change, however minor the change may be.

Testing is typically done using a testing framework such as *JUnit* developed by Beck and Gamma [49, 262]. The framework caters for invoking all test methods of a test class automatically, and for collecting test cases into test suites. Test results can be checked by invoking any of the *assert* methods of the framework with which expected values can be compared to actual values. Testing success is visualized through a graphical user interface showing a growing green bar as the tests progress: as soon as a test fails, the bar becomes red.

The XP process encourages writing a test class for every class in the system. The test code/production code ratio may vary from project to project and in practice we have seen ratios as high as 1:1. Moreover, XP encourages programmers to use tests for documentation purposes, in particular if an interface or method is unclear, if the implementation of a method is complicated, if there are circumstances in which the code should work in a special way, and if a bug report is received [50]. In each of these situations, the test is written *before* the corresponding method is written (or modified) [52].

Also, tests can be added while understanding existing code. In particular, whenever a programmer is tempted to type something into a print statement or debugger instruction, XP advises to write a test instead and add it to the system's test suite [49].

8.3.3 Comprehension Benefits

This section discusses a number of benefits that an automated unit testing regime has for program comprehension.

First, *XP's testing policy encourages programmers to explain their code using test cases.* Rather than explaining the behavior of a function using prose in comments or documentation, the extreme programmer adds a test that explicitly shows the behavior.

Second, *the requirement that all tests must run 100% at all times, ensures that the documentation via unit tests is kept up-to-date*. With regular technical documentation and comments, nothing is more difficult than keeping them consistent with the source code. In XP, all tests must pass before and after every change, ensuring that what the developer writing the tests intended to communicate remains valid.

Third, *adding unit tests provides a repeatable program comprehension strategy*. If a programmer needs to change a piece of code that he is not familiar with, he will try to understand the code by inspecting the test cases. If these do not provide enough understanding, the programmer will try to understand the nature of the code by developing and testing a series of hypotheses, as we have seen in Section 8.3.1. The advise to write tests instead of using print statements or debugger commands applies here as well: *program understanding hypotheses can be translated into unit tests*, which then can be run in order to confirm or refute the hypotheses.

Fourth, *a comprehensive set of unit tests reduces the comprehension space when modifying source code*. To a certain extent a programmer can just try a change and see whether the tests still run. This reduces the risks and complexity of conducting a painstakingly difficult impact analysis. Thus, the XP process attempts to minimize the *size* of the mental model that needs to be build and maintained since the tests help the programmer to see what parts are not affected by the current modifications.

Last but not least, *systematic unit testing helps build team confidence*. In the XP literature, it is said that the tests help the team to develop *courage* to change the code [344].

The XP testing process not only affects the way the team works, it also has a direct effect on the understandability of the production code written [254, p.199]. *Writing unit tests requires that the code tested is split into many small methods each responsible for a clear and testable task*.

In addition, if the tests are written after the production code, it is likely that the production code is difficult to test. For that reason, XP requires that the unit tests are written *before* the code (the "test-driven" approach) [52]. In this way, *testing code and production code are written hand-in-hand, ensuring that the production code is set up in a testable manner*.

8.3.4 Comprehension Risks

Using tests for documentation leads to the somewhat paradoxical situation that in order to understand a given piece of code a programmer has to read another piece of code. Thus, to support program comprehension, XP increases the code base and this code needs to be maintained as well. We experienced that maintaining such test code requires special skills and refactorings, which we describe in Section 8.5.

Also of importance is that tests are automated (with the possible exception of exploratory tests), as non-automated tests probably require knowledge or skill to activate the tests. Knowledge which is possibly not available during (initial) program comprehension [131].

Another concern is that XP uses the tests (in combination with oral communication and code written to display intent) as a *replacement* for technical documentation. The word "documentation" is mentioned once in Beck's book, where he explains why he decided *not* to write documentation [50, p. 156]. For addressing subjects not easily expressed in the tests or code of the system under development, a *technical memorandum* can be written [134]. These are short (one or two pages) papers expressing key ideas and motivations of the design. However, if the general tendency is not to write documentation, it is unlikely that the technical memoranda actually get written, leaving important decisions undocumented.

A final concern is that some types of code are inherently hard to test, the best known examples being user interfaces and database code. Writing tests for such code requires skill, experience, and determination. This will not be always available, leaving the hardest code without tests and thus without documentation.

A possible solution for these cases can be the use of so called *mock objects* which are "simulated" objects that can mimic the behavior of complex objects in a controlled way (often using a form of capture and replay) [336]. Setting up such a mock object can then serve as documentation of the interaction with the real object.

8.4 Test Smells and Refactorings

Continuous refactoring, one of the key practices of extreme programming and most other agile methods, is advocated for bringing the code into the simplest state possible. To aid in the refactoring process a catalog of "code smells" and a wide range of refactorings is available, varying from simple modifications up to ways to systematically introduce design patterns in existing code [273].

From our own experiences we know however that test code is different from production code and this has led us to the following observations:

> **Observation 2** Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other.

> **Observation 3** Improving test code involves a mixture of applying refactorings as identified by Fowler [183] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes and the way of grouping test cases.

In this section we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification.

For the remainder of this chapter, we assume some familiarity with the *x*Unit framework [49] and refactorings as described by Fowler [183]. We will refer to refactorings described in this book using *Name (F:page#)* and to our test specific refactorings described in Section 8.4.2 using *Name* (#).

8.4.1 Test Code Smells

This section gives an overview of bad code smells that are specific for test code.

Smell 1: *Mystery Guest*.
When a test uses external resources, such as a file containing test data, the test is no longer self contained. Consequently, there is not enough information to understand the tested functionality, making it hard to use that test as documentation.

Moreover, using external resources introduces hidden dependencies: if some force changes or deletes such a resource, tests start failing. Chances for this increase when more tests use the same resource.

The use of external resources can be eliminated using the refactoring *Inline Resource* (1). If external resources are needed, you can apply *Setup External Resource* (2) to remove hidden dependencies.

Smell 2: *Resource Optimism.*
Test code that makes optimistic assumptions about the existence (or absence) and state of external resources (such as particular directories or database tables) can cause non-deterministic behavior in test outcomes. Situations where tests run fine at one time and fail miserably the next time are not where you want to find yourself in. Use *Setup External Resource* (2) to allocate and/or initialize all resources that are used.

Smell 3: *Test Run War.*
Such wars arise when the tests run fine as long as you are the only one testing but fail when more programmers run them. This is most likely caused by resource interference: some tests in your suite allocate resources such as temporary files that are also used by others. Apply *Make Resource Unique* (3) to overcome interference.

Smell 4: *General Fixture.*
In the *JUnit* framework a programmer can write a `setUp` method that will be executed before each test method to create a fixture for the tests to run in.

Things start to smell when the `setUp` fixture is too general and different tests only access part of the fixture. Such set-ups are harder to read and understand and may make tests run more slowly (because they do unnecessary work). The danger of having tests that take too much time to complete is that testing starts interfering with the rest of the programming process and programmers eventually may not run the tests at all.

The solution is to use `setUp` only for that part of the fixture that is shared by all tests using Fowler's *Extract Method (F:110)* and put the rest of the fixture in the method that uses it using *Inline Method (F:117)*. If, for example, two different groups of tests require different fixtures, consider setting these up in separate methods that are explicitly invoked for each test, or spin off two separate test classes using *Extract Class (F:149)*.

Smell 5: *Eager Test.*
When a test method checks several methods of the object to be tested, it is hard to read and understand, and therefore more difficult to use as documentation. Moreover, it makes tests more dependent on each other and harder to maintain.

The solution is simple: separate the test code into test methods that test only one method using Fowler's *Extract Method (F:110)*, using a meaningful name highlighting the purpose of the test. Note that splitting into smaller methods can slow down the tests due to increased setup/teardown overhead.

Smell 6: *Lazy Test.*
This occurs when several test methods check the same method *using the same fixture* (but for example check the values of different instance variables). Such tests often only have meaning when considering them together so they are easier to use when joined using *Inline Method (F:117)*.

Smell 7: *Assertion Roulette.*
You know *something* is wrong because your tests fail but it is unclear *what*. This smell comes from having a number of assertions in a single test method that do not

have a distinct explanation. If one of the assertions fails, you do not know which one it is. Use *Add Assertion Explanation* (5) to remove this smell.

Smell 8: *Indirect Testing*.
A test class is supposed to test its counterpart in the production code. It starts to smell when a test class contains methods that actually perform tests on other objects (for example because there are references to them in the class-to-be-tested). Such indirection can be moved to the appropriate test class by applying *Extract Method (F:110)* followed by *Move Method (F:142)* on that part of the test. The fact that this smell arises also indicates that there might be problems with data hiding in the production code.

Note that opinions differ on indirect testing. Some people do not consider it a smell but a way to guard tests against changes in the "lower" classes. We feel that there are more losses than gains to this approach: it is much harder to test anything that can break in an object from a higher level and understanding and debugging indirect tests is much harder.

Smell 9: *For Testers Only*.
When a production class contains methods that are only used by test methods, these methods either (1) are not needed and can be removed, or (2) are only needed to set up a fixture for testing. Depending on functionality of those methods, you may not want them in production code where others can use them. If this is the case, apply *Extract Subclass (F:330)* to move these methods in the testcode and use that subclass to perform the tests on. You will often find that these methods have names or comments stressing that they should only be used for testing.

Fear of this smell may lead to another undesirable situation: a class without corresponding test class. The reason then is that the developer (1) does not know how to test the class without adding methods that are specifically needed for the test and (2) does not want to pollute his production class with test code. Creating a separate subclass helps to deal with this problem.

Smell 10: *Sensitive Equality*.
It is fast and easy to write equality checks using the `toString` method. A typical way is to compute an actual result, map it to a string, which is then compared to a string literal representing the expected value. Such tests, however may depend on many irrelevant details such as commas, quotes, spaces, etc. Whenever the `toString` method for an object is changed, tests start failing. The solution is to replace `toString` equality checks by real equality checks using *Introduce Equality Method* (6).

Smell 11: *Test Code Duplication*.
Test code may contain undesirable duplication. In particular the parts that set up test fixtures are susceptible to this problem. Solutions are similar to those for normal code duplication as described by Fowler [183, p. 76]. The most common case for test code will be duplication of code in the same test class. This can be removed using *Extract Method (F:110)*. For duplication across test classes, it may be helpful to mirror the class hierarchy of the production code into the test class hierarchy. A word of caution

however: moving duplicated code from two separate classes to a common class can introduce (unwanted) dependencies between tests.

A special case of code duplication is *test implication*: test *A* and *B* cover the same production code, and *A* fails if and only if *B* fails. A typical example occurs when the production code gets refactored: before this refactoring, *A* and *B* covered different code, but afterwards they deal with the same code and it is not necessary anymore to maintain both tests. Because it fails to distinguish between the various cases, test implication impedes comprehension and documentation.

8.4.2 Test Refactorings

Bad smells seem to arise more often in production code than in test code. The main reason for this is that production code is adapted and refactored more frequently, allowing these smells to escape.

One should not, however, underestimate the importance of having fresh test code. Especially when new programmers are added to the team or when complex refactorings need to be performed, clear test code is invaluable. To maintain this freshness, test code also needs to be refactored.

We define *test refactorings* as changes (transformations) of test code that: (1) do not add or remove test cases, and (2) make test code better understandable/readable and/or maintainable [518].

The remainder of this section presents refactorings that we encountered while working on test code. Not all of these refactorings are directly linked with the elimination of the test smells of Section 8.4.1, but when a link is there, it is described.

Refactoring 1: *Inline Resource*.
To remove the dependency between a test method and some external resource, we incorporate that resource in the test code. This is done by setting up a fixture in the test code that holds the same contents as the resource. This fixture is then used instead of the resource to run the test. A simple example of this refactoring is putting the contents of a file that is used into some string in the test code.

If the contents of the resource are large, chances are high that you are also suffering from *Eager Test* (5) smell. Consider conducting *Extract Method (F:110)* or *Reduce Data* (4) refactorings.

Refactoring 2: *Setup External Resource*.
If it is necessary for a test to rely on external resources, such as directories, databases, or files, make sure the test that uses them explicitly creates or allocates these resources before testing, and releases them when done (take precautions to ensure the resource is also released when tests fail).

Refactoring 3: *Make Resource Unique*.
A lot of problems originate from the use of overlapping resource names, either between different tests run done by the same user or between simultaneous test runs done by different users.

Such problems can easily be prevented (or repaired) by using unique identifiers for all resources that are allocated, e.g. by including a time-stamp. When you also include the name of the test responsible for allocating the resource in this identifier, you will have less problems finding tests that do not properly release their resources.

Refactoring 4: *Reduce Data.*
Minimize the data that is setup in fixtures to the bare essentials. This will have two advantages: (1) it makes them better suitable as documentation, and (2) your tests will be less sensitive to changes.

Refactoring 5: *Add Assertion Explanation.*
Assertions in the JUnit framework have an optional first argument to give an explanatory message to the user when the assertion fails. Testing becomes much easier when you use this message to distinguish between different assertions that occur in the same test. Maybe this argument should not have been optional...

Refactoring 6: *Introduce Equality Method.*
If an object structure needs to be checked for equality in tests, add an implementation for the "equals" method for the object's class. You then can rewrite the tests that use string equality to use this method. If an expected test value is only represented as a string, explicitly construct an object containing the expected value, and use the new equals method to compare it to the actually computed object.

8.4.3 Other Test Smells and Refactorings

Fowler [183] presents a large set of bad smells and refactorings that can be used to remove them. Our work focuses on smells and refactorings that are typical for test code, whereas Fowler focuses more on production code. The role of unit tests in [183] is also more geared towards proving that a refactoring did not break anything than to be used as documentation of the production code.

Instead of focusing on cleaning test code which already has bad smells, Schneider [454] describes how to prevent these smells right from the start by discussing a number of best practices for writing tests with JUnit.

The C2 Wiki contains some discussion on the decay of unit test quality and practice as time proceeds [98], and on the maintenance of broken unit tests [542]. Opinions vary between repairing broken unit tests, deleting them completely, and moving them to another class in order to make them less exposed to changes (which may lead to our *Indirect Testing* (8) smell).

Van Rompaey et al. present an approach in which test smells are detected and then ranked according to their relative significance [521]. For this, they rely on a metric-based heuristic approach. They focus on the "General Fixture" and "Eager Test" test smells (Smell 4 & 5 in Section 8.4.1).

Besides the test smells we described earlier, Meszaros [372] discusses an additional set of process-oriented test smells and their refactorings.

8.5 How Refactoring Can Invalidate Its Safety Net

When evolving a piece of software, the change activities can roughly be divided into two categories. The first category consists of those operations that preserve behavior, i.e. refactorings, while the second category contains those changes that do not necessarily preserve behavior. Intuitively, when non-behavior-preserving changes are applied to production code, one would expect that the associated test code would need to evolve as well, as the end-result of the computation is bound to be different.

When thinking of refactorings of production code however, the picture is not that clear whether the associated unit tests need to evolve as well. Refactoring, which aims to improve the internal structure of the code, happens e.g. through the removal of duplication, simplification, making code easier to understand, adding flexibility, ... Fowler describes it as: *"Without refactoring, the design of software will decay. Regular refactoring helps code retain its shape."* [183, p.55].

One of the dangers of refactoring is that a programmer unintentionally changes the system's behavior. Ideally, it can be verified that this did not happen by checking that all the tests pass after refactoring. In practice, however, we have noticed that there are refactorings that will invalidate tests, as tests often rely, to a certain extent, on the code structure, which may have been affected by the refactoring (e.g., when a method is moved to another class and the test still expects it in the original class).

From this perspective, we observed the following:

> **Observation 4** The refactorings as proposed by Fowler [183] can be classified based on the type of change they make to the code, and therefore on the possible change they require in the test code.

> **Observation 5** In parallel to test-driven design, *test-driven refactoring* can improve the design of production code by focusing on the desired way of organizing test code to drive refactoring of production code (i.e., refactor for testing).

To explore the relationship between unit testing and refactorings, we take the following path: we first set up a classification of the refactorings described by Fowler [183], identifying exactly which of the refactorings affect class interfaces, and which therefore require changes in the test code as well (see Section 8.5.1). Subsequently, we look at the video store example from [183], and assess the implications of each refactoring on the test code (Section 8.5.2). We explore *test-driven refactoring*, which analyzes the test code in order to arrive at code level refactorings (Section 8.5.3), before we discuss the relationship between code-level refactorings and test-level refactorings (Section 8.5.4). We then integrate our results via the notion of a *refactoring session* which is a coherent set of steps resulting in refactoring of both the code and the tests (Section 8.5.5).

8.5.1 Types of Refactoring

Refactoring a system should not change its observable behavior. Ideally, this is verified by ensuring that all the tests pass before and after a refactoring [50, 183].

In practice, it turns out that such verification is not always possible: some refactorings restructure the code in such a way that tests only can pass after the refactoring if they are modified. For example, refactoring can move a method to a new class while some tests expect it in the original class (in that case, the code will probably not even compile).

This unfortunate behavior was also noted by Fowler: “Something that is disturbing about refactoring is that many of the refactorings do change an interface.” [183, p.64]. Nevertheless, we do not want to change the tests together with a refactoring since that will make them less trustworthy for validating correct behavior afterwards.

In the remainder of this section, we will look in more detail at the refactorings described by Fowler [183] to analyze in which cases problems might arise because the original tests need to be modified.

Taxonomy

If we start with the assumption that refactoring does not change the behavior of the system, then there is only one reason why a refactoring can break a test: *because the refactoring changes the interface that the test expects*. Note that the interface extends to all visible aspects of a class (fields, methods, and exceptions). This implies that one has to be careful with tests that directly inspect the fields of a class since these will more easily change during a refactoring[5].

So, initially, we distinguish two types of refactoring: refactorings that do not change any interface of the classes in the system and refactorings that do change an interface. The first type of refactorings has no consequences for the tests: since the interfaces are kept the same, tests that succeeded before refactoring will also succeed after refactoring (if the refactoring indeed preserves the tested behavior).

The second type of refactorings can have consequences for the tests since there might be tests that expect the old interface. Again, we can distinguish two situations:

Incompatible: the refactoring destroys the original interface. All tests that rely on the old interface must be adjusted.

Backwards Compatible: the refactoring extends the original interface. In this case the tests keep running via the original interface and will pass if the refactoring preserves tested behavior. Depending on the refactoring, we might need to add more tests covering the extensions.

A number of incompatible refactorings that normally would destroy the original interface can be made into backwards compatible refactorings. This is done by extending the refactoring so it will retain the old interface, for example, using the *Adapter* pattern or simply via delegation. As a side-effect, the new interface will already partly be tested. Note that this is common practice when refactoring a published interface to prevent breaking dependent systems. A disadvantage is that a larger interface has to be maintained but when delegation or wrapping was used, that should not be too much work. Furthermore, language features like deprecation can be used to signal that this part of the interface is outdated.

[5] In fact, direct inspection of fields of a class is a test smell that could better be removed beforehand [518].

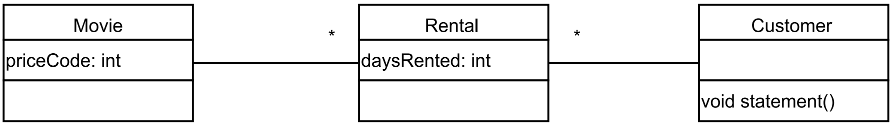

Fig. 8.1. Classes before refactoring

Classification

We have analyzed the refactorings in [183] and divided them into the following classes:

A. *Composite:* The four big refactorings *Convert Procedural Design to Objects*, *Separate Domain from Presentation*, *Tease Apart Inheritance*, and *Extract Hierarchy* will change the original interface, but we will not consider them in more detail since they are performed as series of smaller refactorings.
B. *Compatible:* Refactorings that do not change the original interface. Refactorings in this class are listed in Table 8.1.
C. *Backwards Compatible:* Refactorings that change the original interface and are inherently backwards compatible since they extend the interface. Refactorings in this class are listed in Table 8.2.
D. *Make Backwards Compatible:* Refactorings that change the original interface and can be made backwards compatible by adapting the new interface to the new one. Refactorings in this class are listed in Table 8.3.
E. *Incompatible:* Refactorings that change the original interface and are not backwards compatible (for example, because they change the types of classes that are involved). Refactorings in this class are listed in Table 8.4.

Note that the refactorings *Replace Inheritance with Delegation* and *Replace Delegation with Inheritance* are listed both in the *Compatible* and *Backwards Compatible* tables since they can be of either category, depending on the actual case.

8.5.2 Revisiting the Video Store

In this section, we study the relationship between testing and refactoring using a well-known example of refactoring. We revisit the video store code used by Fowler [183, Chapter 1], extending it with an analysis of what should be going on in the accompanying video store test code.

The video store class structure before refactoring is shown in Figure 8.1. It consists of a *Customer*, who is associated with a series of *Rentals*, each consisting of a *Movie* and an integer indicating the number of days the movie was rented. The key functionality is in the Customer's *statement* method printing a customer's total rental cost. Before refactoring, this statement is printed by a single long method. After refactoring, the statement functionality is moved into appropriate classes, resulting in the structure of Figure 8.2 taken from [183, p. 51].

Fowler emphasizes the need to conduct refactorings as a sequence of small steps. At each step, you must run the tests in order to verify that nothing essential has

Table 8.1. Compatible refactorings (type B)

Change Bidirectional Association to Unidirectional
Replace Nested Conditional with Guard Clauses
Replace Magic Number with Symbolic Constant
Consolidate Duplicate Conditional Fragments
Replace Conditional with Polymorphism
Replace Inheritance with Delegation
Replace Delegation with Inheritance
Replace Method with Method Object
Remove Assignments to Parameters
Replace Data Value with Object
Introduce Explaining Variable
Replace Exception with Test
Change Reference to Value
Split Temporary Variable
Decompose Conditional
Introduce Null Object
Preserve Whole Object
Remove Control Flag
Substitute Algorithm
Introduce Assertion
Extract Class
Inline Temp

Table 8.2. Backwards compatible refactorings (type C)

Replace Inheritance with Delegation
Replace Delegation with Inheritance
Consolidate Conditional Expression
Replace Record with Data Class
Introduce Foreign Method
Pull Up Constructor Body
Replace Temp with Query
Duplicate Observed Data
Self Encapsulate Field
Form Template Method
Extract Superclass
Extract Interface
Push Down Method
Push Down Field
Pull Up Method
Extract Method
Pull Up Field

Table 8.3. Refactorings that can be made backwards compatible (type D)

Change Unidirectional Association to Bidirectional
Replace Parameter with Explicit Methods
Replace Parameter with Method
Separate Query from Modifier
Introduce Parameter Object
Parameterize Method
Remove Middle Man
Remove Parameter
Add Parameter
Rename Method
Move Method

Table 8.4. Incompatible refactorings (type E)

Replace Constructor with Factory Method
Replace Type Code with State/Strategy
Replace Type Code with Subclasses
Replace Error Code with Exception
Replace Subclass with Fields
Replace Type Code with Class
Change Value to Reference
Introduce Local Extension
Replace Array with Object
Encapsulate Collection
Remove Setting Method
Encapsulate Downcast
Collapse Hierarchy
Encapsulate Field
Extract Subclass
Hide Delegate
Inline Method
Inline Class
Hide Method
Move Field

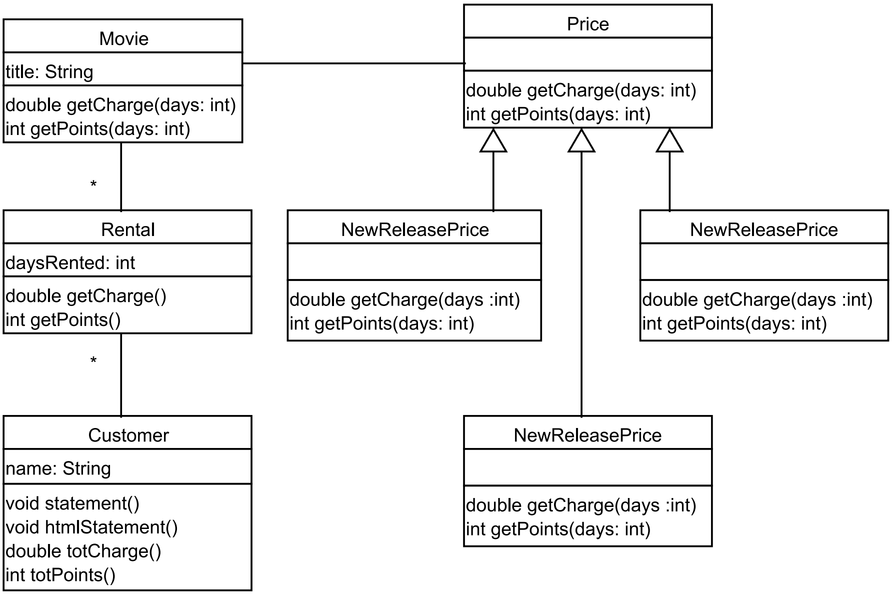

Fig. 8.2. Class structure after refactoring

changed. His testing approach is the following: “I create a few customers, give each customer a few rentals of various kinds of films, and generate the statement strings. I then do a string comparison between the new string and some reference strings that I have hand checked” [183, p. 8]. Although Fowler does not list his test classes, this typically should look like the code in Figure 8.3.

Studying this string-based testing method, we make the following observations:

- The setup is complicated, involving the creation of many different objects.
- The documentation value of the test is limited: it is hard to relate the computation of the charge of 4.5 for movie `m1` to the way in which charges are computed for the actual movies rented (in this case a children’s and a regular movie, each with their own price computation).
- The tests are brittle. All test cases include a full statement string. When the format changes in just a very small way, all existing tests (!) must be adjusted, an error prone activity we would like to avoid.

Unfortunately, there is no other way to write tests for the given code. The poor structure of the long method necessarily leads to an equally poor structure of the test cases. From a testing perspective, we would like to be able to separate computations from report writing. The long statement method prohibits this: it needs to be refactored in order to be able to improve the testability of the code.

This way of reasoning naturally leads to the application of the *Extract Method* refactoring to the *statement* method. Fowler comes to the same conclusion, based on the need to write a new method printing a statement in HTML format. Thus, we

```
Movie m1 = new Movie("m1",Movie.CHILDRENS);
Movie m2 = new Movie("m2", Movie.REGULAR);
Movie m3 = new Movie("m3", Movie.NEW_RELEASE);
Rental r1 = new Rental(m1, 5);
Rental r2 = new Rental(m2, 7);
Rental r3 = new Rental(m3, 1);
Customer c1 = new Customer("c1");
Customer c2 = new Customer("c2");

public void setUp() {
  c1.addRental(r1);
  c1.addRental(r2);
  c2.addRental(r3);
}

public void testStatement1() {
  String  expected =
    "Rental Record for c1\n" +
    "\tm1\t4.5\n" +
    "\tm2\t9.5\n" +
    "Amount owed is 14.0\n" +
    "You earned 2 frequent renter points";
  assertEquals(expected, c1.statement());
}
```

Fig. 8.3. Initial sample test code

extract *getCharge* for computing the charge of a rental, and *getPoints* for computing the “frequent renter points”.

Extract Method is of type C, the *backwards compatible* refactorings, so we can use our existing tests to check the refactoring. However, we have created new methods, for which we might like to add tests that document and verify their specific behavior. To create such tests, we can reuse the setup of movies, rentals, and customers used for testing the *statement* method. We end up with a number of smaller test cases specifically addressing either the charge or rental point computations.

Since the correspondence between test code and actual code is now much clearer and better focused, we can apply white box testing, and use our knowledge of the structure of the code to determine the test cases needed. Thus, we see that the *getCharge* method to be tested distinguishes between 5 cases, and we make sure our tests cover these cases.

This has solved some of the problems. The tests are better understandable, more complete, much shorter, and less brittle. Unfortunately, we still have the complicated setup method. What we see is that the setup mostly involves rentals and movies, while the tests themselves are in the customer testing class. This is because the extracted method is in the wrong class: applying *Move Method* to *Rental* simplifies the set up for new test cases. Again we use our analysis of the test code to find refactorings in the production code.

The *Move Method* is of type D, refactorings that can be made backwards compatible by adding a wrapper method to retain the old interface. We add this wrapper so we can check the refactoring with our original tests. However, since the documentation of the method is in the test, and this documentation should be as close as possible to the method documented, we want to move the tests to the method's new location. Since there is no test class for Rental yet, we create it, and move the test methods for *getCharge* to it. Depending on whether the method was part of a published interface, we might want to keep the wrapper (for some time), or remove it together with the original test.

Fowler discusses several other refactorings, moving the charge and point calculations further down to the *Movie* class, replacing conditional logic by polymorphism in order to make it easier to add new movie types, and introducing the *state* design pattern in order to be able to change movie type during the life time of a movie.

When considering the impact on test cases of these remaining video store refactorings, we start to recognize a pattern:

- Studying the test code and the smells contained in it may help to identify refactorings to be applied at the production code;
- Many refactorings involve a change to the structure of the unit tests as well: in order to maintain the documenting value of these unit tests, they should be changed to reflect the structure of the code being tested.

In the next two sections, we take a closer look at these issues.

8.5.3 Test-Driven Refactoring

In *test-driven refactoring*, we try to use the existing test cases in order to determine the code-level refactorings. Thus, we study *test* code in order to find improvements to the *production* code.

This calls for a set of *code smells* that helps to find such refactorings. A first category is the set of existing code smells discussed in Fowler's book [183]. Several of them, such as long method, duplicated code, long parameter list, and so on, apply to test code as well as they do to production code. In many cases solving them involves not just a change on the test code, but first of all a refactoring of the production code.

A second category of smells is the collection of *test smells* discussed in Section 8.4 (also see [518]). In fact, in our movie example we encountered several of them already. Our uneasy feeling with the test case of Figure 8.3 is captured by the *Sensitive Equality* smell [518, Smell 10]: comparing computed values to a string literal representing the expected value. Such tests depend on many irrelevant details, such as commas, quotes, tabs, ... This is exactly why the customer tests of Figure 8.3 become brittle.

Another *test smell* we encountered is called *Indirect Testing* [518, Smell 8]: a test class contains methods that actually perform tests on other objects. Indirect tests make it harder to understand the relationship between test and production code. While moving the *getCharge* and *getPoints* methods in the class hierarchy (using *Move Method*), we also moved the corresponding test cases, in order to avoid *Indirect Testing*.

The test-driven perspective may lead to the formulation of additional test smells. For example, we observed that setting up the fixture for the CustomerTest was complicated. This indicates that the tests are in the wrong class, or that the underlying business logic is not well isolated. Another smell appears when there are many test cases for a single method, indicating that the method is too complex.

Test-driven refactoring is a natural consequence of test-driven design. Test-driven design is a way to get a good design by thinking about test cases first when adding functionality. Test-driven refactoring is a way to improve your design by rethinking the way you structured your tests.

In fact, Beck's work on test-driven design [51, 52] contains an interesting example that can be transferred to the refactoring domain. It involves testing the construction of a mortality table. His first attempt requires a complicated setup, involving separate "person" objects. He then rejects this solution as being overly complex for testing purposes, and proposes the construction of a mortality table with just an age as input. His example illustrates how test case construction guides design when building new code; likewise, test case refactoring guides the improvement of design during refactoring.

8.5.4 Refactoring Test Code

In our study of the video store example, we saw that many refactorings on the code level can be completed by applying a corresponding refactoring on the test case level. For example, to avoid *Indirect Testing*, the refactoring *Move Method* should be followed by "*Move Test*". Likewise, in many cases *Extract Method* should be followed by "*Extract Test*". To retain the documentation value of the unit tests, their structure should be in sync with the structure of the source code.

In our opinion, it makes sense to extend the existing descriptions of refactorings with suggestions on what to do with the corresponding unit tests, for example in the "mechanics" part.

The topic of refactoring test code is discussed extensively in Section 8.4. An issue of concern when changing test code is that we may "lose" test cases. When refactoring production code, the availability of tests forms a safety net that guards us from accidentally losing code, but such a safety net is not in place when modifying test code. A solution is to measure coverage [346] before and after changing the tests, e.g. with the help of Clover [108] or Emma [469]. One step further is *mutation testing*, using a tool such as Jester [379, 470]. Jester automatically makes changes to conditions and literals in Java source code. If the code is well-tested, such changes should lead to failing tests. Running Jester before and after test case refactorings helps to verify that the changes did not affect test coverage.

8.5.5 Refactoring Sessions

The meaningful unit of refactoring is a sequence of steps involving changes to both the code base and the test base. We propose the notion of a *refactoring session* to capture such a sequence. It consists of the following steps:

1. Detect *smells* in the code or test code that need to be fixed. In test-driven refactoring, the test set is the starting point for finding such smells.
2. Identify candidate refactorings addressing the smell.
3. Ensure that all existing tests run.
4. Apply the selected refactoring to the code. Provide a backwards compatible interface if the refactoring falls in category D. Only change the associated test classes when the refactoring falls in category E.
5. Ensure that all existing tests run. Consider applying mutation testing to assess the coverage of the test cases.
6. Apply the testing counterpart of the selected refactoring.
7. Ensure that the modified tests still run. Check that the coverage has not changed.
8. Extend the test cases now that the underlying code has become easier to test.
9. Ensure the new tests run.

The integrity of the code is ensured since (1) all tests are run between each step; (2) each step changes either code or tests, but never both at the same time (unless this is impossible).

8.6 Measuring Code and Test Code

In the previous sections we have seen how test suites affect program comprehension, how test suites themselves can be subjected to refactoring, and how refactoring of the production code is reflected in the test code. The last thing we investigate is whether there is a relation (correlation) between certain properties of the production code and those of the test code. We look at one property in particular, namely the *testability* of production code, based on our earlier work on finding testability metrics for Java systems [89].

For our investigation, we take advantage of the popularity of the JUnit framework [262]. JUnit's typical usage scenario is to test each Java class C by means of a dedicated test class C_T, generating pairs of the form $\langle C, C_T \rangle$. The route then that we pursue is to use these pairs to find source code metrics on C that are good predictors of test-related metrics on C_T.

To elaborate this route, we first define the notion of testability that we address, then describe the experimental design that can be used to explore the hypothesis, followed by a discussion of initial experimental results.

8.6.1 Testability

The ISO defines testability as "attributes of software that bear on the effort needed to validate the software product" [240]. Binder [65] offers an analysis of the various factors that contribute to a system's testability, which he visualizes using the fish bone diagram as shown in Figure 8.4. The major factors determining test effort that Binder distinguishes include the test adequacy criterion that is required, the usefulness of the documentation, the quality of the implementation, the reusability and structure of the test suite, the suitability of the test tools used, and the process capabilities.

Of these factors, we are concerned with the structure of the implementation, and with source code factors in particular. One group of factors we distinguish are *test case generation* factors, which influence the *number* of test cases required. An example is the testing criterion (test all branches, test all inherited methods), but directly related are characteristics of the code itself (use of if-then-else statements, use of inheritance). The other group of factors we distinguish are *test case construction factors*, which are related to the effort needed to create a particular test case. Such factors include the complexity of creating instances for a given class, or the number of fields that need to be initialized.

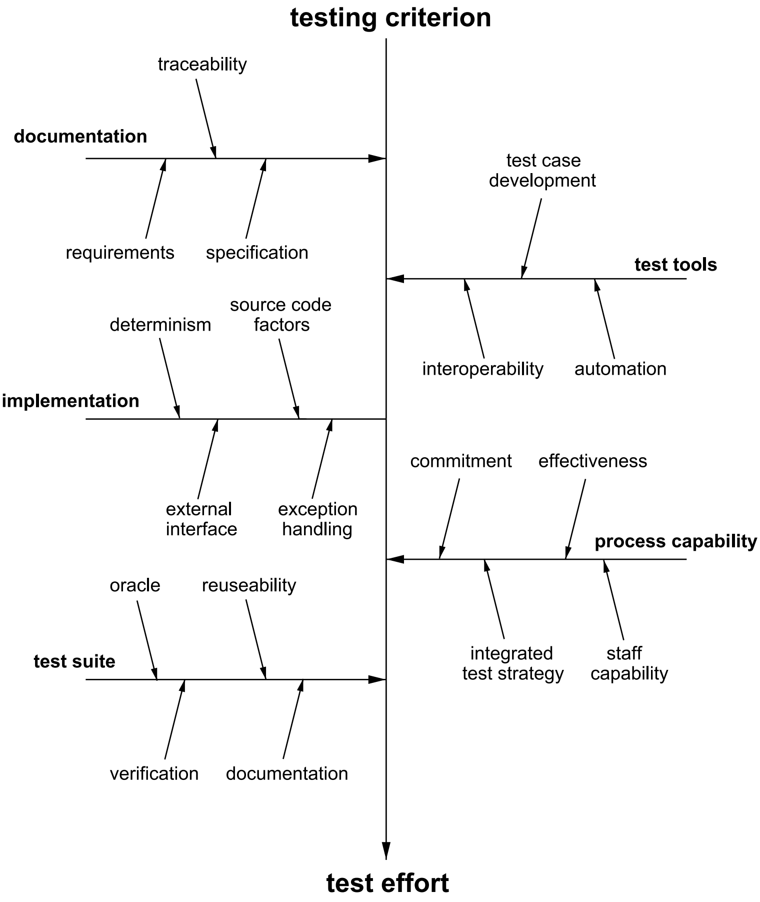

Fig. 8.4. The testability fish-bone [65, 89]

8.6.2 Experimental Design

Our goal is to assess the capability of a suite of object-oriented metrics to predict testing effort. We assess this capability from a class level perspective, i.e., we assess whether or not the values of object-oriented metrics for a given class can predict the required amount of effort needed for unit testing that class. The particular environment in which we conduct the experiments consists of Java systems that are unit tested at the class level using the JUnit testing framework.

To help us translate the goal into measurements, we pose questions that pertain to the goal:

> **Question 1:** Are the values of the object-oriented metrics for a class associated with the required testing effort for that class?

To answer this question, we must first quantify “testing effort.” To indicate the testing effort required for a class we use the size of the corresponding test suite. Well-known cost models such as Boehm’s COCOMO [72] and Putnam’s SLIM model [421] relate development cost and effort to software size. Test suites are software in their own right; they have to be developed and maintained just like ‘normal’ software. Below we will see which metrics we use to measure the size of a test suite.

Next, we can refine our original question, and obtain the following new question:

> **Question 2:** Are the values of the object-oriented metrics for a class associated with the size of the corresponding test suite?

From these questions we can derive a hypothesis that our experiments test:

$\mathbf{H_0}(m,n)$: There is *no* association between object-oriented metric *m* and test suite metric *n*,

$\mathbf{H_1}(m,n)$: There is an association between object-oriented metric *m* and test suite metric *n*,

where *m* ranges over our set of object-oriented metrics, and *n* over our set of test-suite based metrics.

As a candidate set of object-oriented metrics, we use the suite proposed by Binder [65] as a starting point. Binder is interested in testability as well, and uses a model distinguishing “complexity” and “scope” factors, which are similar to our test case construction and generation factors. The metrics used by Binder are based on the well known metrics suite provided by Chidamber and Kemerer [111], who for some of their metrics (such as the Coupling Between Objects and the Response for Class) already suggested that they would have a bearing on test effort. The metrics that we have used in our experiments are listed in Table 8.5.

For our experiments we propose the dLOCC (Lines Of Code for Class) and dNOTC (Number of Test Cases) metrics to indicate the size of a test suite. The ‘d’ prepended to the names of these metrics denotes that they are the *dependent* variables of our experiment, i.e., the variables we want to predict. The dLOCC metric is defined like the LOCC metric.

Table 8.5. Metrics suite used for assessing testability of a class *c*

Metric	Description
DIT	Depth of inheritance tree
FOUT	Fan out, nr of classes used by *c*
LCOM	Lack of cohesion in methods—which measures how fields are used in methods
LOCC	Lines of code per class
NOC	Number of children
NOF	Number of fields
NOM	Number of methods
RFC	Response for class—Methods in *c* plus the number of methods invoked by *c*.
WMC	Weighted methods per class—sum of McCabe's cyclomatic complexity number of all methods.

The dNOTC metric provides a different perspective on the size of a test suite. It is calculated by counting the number of invocations of JUnit 'assert' methods that occur in the code of a test class. JUnit provides the tester with a number of different 'assert' methods, for example 'assertTrue', 'assertFalse' or 'assertEqual'. The operation of these methods is the same; the parameters passed to the method are tested for compliance to some condition, depending on the specific variant. For example, 'assertTrue' tests whether or not its parameter evaluates to 'true'. If the parameters do not satisfy the condition, the framework generates an exception that indicates a test has failed. Thus, the tester uses the set of JUnit 'assert' methods to compare the expected behavior of the class-under-test to its current behavior. Counting the number of invocations of 'assert' methods, gives the number of comparisons between expected and current behavior which we consider an appropriate definition of a test case.

Conducting the measurements yields a series of values $\langle m,n \rangle$ of object-oriented metric m and test suite metric n for a series of pairs $\langle C,C_T \rangle$ of a class C and its corresponding test class C_T. To test the hypotheses, we calculate Spearman's rank-order correlation (which does not require a normal distribution of the data), yielding values $r_s(m,n)$ for metrics m and n. The significance (related to the number of observations made) of the value of r_s found is subsequently determined by calculating the t-statistic, yielding a value p indicating the chance that the observed value is the result of a chance event, allowing us to accept $\mathrm{H}_\mathbf{1}(m,n)$ with confidence level $1-p$.

8.6.3 Experimental Results

Experiments were conducted on five software systems, of which four were closed source software products developed at the Software Improvement Group (SIG)[6]. Additionally, we included Apache Ant [18], an open source automation tool for software development. All systems are written in Java and the systems totaled over 290 KLOC.

[6] `http://www.sig.nl`.

The key results for the Ant case study are shown in Table 8.6; similar results were obtained for the other case studies. The experiment shows that there is a significant correlation between test level metrics dLOCC (Lines of Code for Class) and dNOT (Number of Testcases) and various class level metrics:

- There are several metrics related to *size*, in particular LOCC, NOM, and WMC. Since size can be considered a test case generation (we need more test cases) as well as a test case construction factor (larger classes become harder to test), it is natural that these metrics are correlated with test effort.
- The inheritance related metrics DIT (depth of inheritance tree) and NOC (number of subclasses) are *not* correlated with test metrics. In principle, test strategies in which, for example, extra subclasses lead to more intensive testing of the superclass, could cause NOC or DIT to be predictors of test effort. Apparently in the case studies these strategies were not adopted.
- Two metrics measuring external dependencies are Fan Out (FOUT) and Response-for-Class (RFC). Both are clearly correlated with both test suite metrics.
- The metrics LCOM (Lack of Cohesion of Methods) and NOF (Number of Fields) are correlated with the test metrics for the Ant case as well, but not for the four commercial case studies. One can expect NOF to be an indicator for test effort, for example, for initializing fields in a class. In cases where NOF is not an indicator, this may be due to the fact that the NOF metric only measures fields introduced in a particular class, and not fields inherited from superclasses.

Based on these findings, we conclude with the following observation:

> **Observation 6** Traditional object-oriented source code metrics applied to production code can indicate the effort needed for developing unit tests.

We refer to Bruntink and Van Deursen for a full account of the experiments described above [89].

Table 8.6. Correlation values and confidence levels found for Ant

r_s	dLOCC	dNOTC
DIT	-.0456	-.201
FOUT	.465	.307
LCOM	.437	.382
LOCC	.500	.325
NOC	.0537	-.0262
NOF	.455	.294
NOM	.532	.369
RFC	.526	.341
WMC	.531	.348

p	dLOCC	dNOTC
DIT	.634	.0344
FOUT	$< .01$	$< .01$
LCOM	$< .01$	$< .01$
LOCC	$< .01$	$< .01$
NOC	.575	.785
NOF	$< .01$	$< .01$
NOM	$< .01$	$< .01$
RFC	$< .01$	$< .01$
WMC	$< .01$	$< .01$

8.7 Concluding Remarks

In this section we first look back on the interplay between software testing and evolution. We then present a research agenda with a number of future research tracks, which are currently left unexplored.

8.7.1 Retrospective

Based upon *Observation 1* (see page 177), which states that an extensive test suite can stimulate the program comprehension process in the light of continuously evolving software, we have investigated the interactions between software evolution, software testing and program comprehension that exist in extreme programming in Section 8.3. Naturally, some (or all) of these elements are used in other development processes as well. For example, Humphrey stresses the importance of inspections, software quality assurance, and testing [236]. The Rational Unified Process emphasizes short iterations, architecture centric software development, and use cases [299]. Key publications on extreme programming [50, 254, 48] cover many issues related to comprehension, such as code expressing intent, feedback from the system, and tests to document code.

From our observation that test code has a distinct set of smells (see *Observation 2*, page 180), we looked at test code from the perspective of refactoring. Our own experiences are that the quality of test code is not as high as the quality of the production code. Test code was not refactored as mercilessly as production code, following Fowler's advice that it is acceptable to copy and edit test code, trusting our ability to refactor out truly common items later [183, p. 102]. When at a later stage we started refactoring test code more intensively, we discovered that test code has its own set of problems (which we translated into smells) as well as its own repertoire of solutions (which we formulated as test refactorings).

For each test smell that we identified, we have provided a solution, using either a potentially specialized variant of an existing refactoring from Fowler [183] or a dedicated test refactoring. We believe that the resulting smells and refactorings provide a valuable starting point for a larger collection based on a broader set of projects. This is in line with our *Observation 3* (see page 180).

Observation 4 (see page 185) states that when applying the refactorings as proposed by Fowler [183] on production code, a classification can be made based on whether these refactorings necessitate refactoring the test code as well. In Section 8.5 we have analyzed which of the documented refactorings affect the test code. It turns out that the majority of the refactorings are in category D (requiring explicit actions to keep the interface compatible) and E (necessarily requiring a change to the test code). We have shown the implications of refactoring tests with the help of Fowler's video store example. We then proposed the notion of *test-driven refactoring*, which uses the existing test cases as the starting point for finding suitable code level refactorings.

We have argued for the need to extend the descriptions of refactorings with a section on their implications on the corresponding test code. If the tests are to maintain their documentation value, they should be kept in sync with the structure of the code. As outlined in *Observation 5* (see page 185), we propose, as a first step, the notion of a *refactoring session*, capturing a coherent series of separate steps involving changes to both the production and the test code.

The impact of program structure on test structure is further illustrated through *Observation 6* (page 197), which suggests that traditional object-oriented metrics can be used to estimate test effort. We described an experiment to assess which metrics can be used for this purpose. Note that some of the metrics identified (such as fan-out or response-for-class) are also indicators for class complexity. This suggests that high values for such metrics may call for refactorings, which in turn may help to reduce the test effort required for unit testing these classes.

From our studies we have learned that the interplay between software evolution and software testing is often more complex than meets the eye. The interplay that we witnessed works in two directions: software evolution is hindered by the fact that when evolving a system, the tests often need to co-evolve, making the evolution more difficult and time-intensive. On the other hand, many software evolution operations cannot safely take place without adequate tests being present to enable a safety net. This leads to an almost paradoxical situation where tests are essential for evolving software, yet at the same time, they are obstructing that very evolution.

Another important factor in this interplay is *program comprehension*, or the process of building up knowledge about a system under study, which is of critical importance during software evolution. In this context, having a test suite available can be a blessing, as the tests provide documentation about how the software works. At the same time, when no tests are available, writing tests to understand the software is a good way of building up comprehension.

We have seen that software evolution and testing are intertwined at the very core of (re)engineering software systems and continue to provide interesting and challenging research topics.

8.7.2 Research Agenda

During our study we came across a number of research ideas in the area of software testing and software evolution that are as yet still unexplored. The topics we propose can be seen as an addition or refinement to the topics that were addressed by Harrold in her “Testing: A Roadmap” [224].

Model Driven Engineering

MDE [453] is a modeling activity, whereby the traditional activity of writing code manually is replaced by modeling specifications for the application. Code generation techniques then use these models to generate (partial) code models of the application. This setup ensures the alignment between the models and the executable implementation. A similar approach can be followed when it comes to testing the application: modeling both the application and the tests through specifications. Muccini et al. consider this as the next logical step [381]. Recently, Pickin et al. have picked up on this research topic in the context of distributed systems [415].

Aspect Oriented Programming

AOP [276] is a programming paradigm that aims to offer an added layer of abstraction that can modularize system-level concerns (also see Chapter 9). However, when these aspects are woven into the base code, some unexpected effects can occur that are difficult to oversee. This can happen (1) when the pointcut is not defined precisely enough, resulting in an aspect being woven in at an unexpected place in the base program, or (2) because of unexpected results because of aspect composition, when the advice of two separate aspects is woven in. McEachen et al. describe a number of possible fault scenarios that can occur [357], but further research into this area is certainly warranted to prevent such fault scenarios through testing.

Test Case Isomorphism

Various sources indicate that test cases should be independent of each other because this decreases testing time, increases test output comprehensibility and having concise and focused tests increases their benefit as documentation of a specific aspect of the code [149, 131].

As said, having concise and focused tests decreases the testing time, which partly alleviates the problem of having to do selective regression testing [444, 445]. Another problem situation that is overcome, is the one described by Gaelli et al., whereby broken unit tests are ordered, so that the most specific unit test can be dealt with first [189].

Research questions of interest are how we can characterize and measure this isomorphism and what refactorings can be used to improve this isomorphism. These are related to detecting and removing the test implication smell described earlier.

Service-Orientation

The current trend is to build software systems from loosely coupled components or services (see Chapter 7). These services have mostly not been designed to co-exist with each other from their phase of inception and their "integration" often depends on the configuration of parameters at run-time. Although the components (or services) themselves will probably be of a higher quality, due to the fact that these are shared by many different projects (this can e.g. be in the case of *Commercial Off The Shelf* (COTS) components), testing the integration of these components or services is all the more important.

Although work related to testing components [212, 539] is readily available, not so much can be found on testing service-orientation. Although it is probable that many existing testing techniques can be adapted to work in this context, additional research is warranted. One of the first attempts at tool support for testing services is Coyote [507]. Commercial tool-support comes from SOAPSonar and Ikto's LISA and also Apache's Jakarta JMeter is useful when testing services [467].

Empirical Studies

Although many testing techniques are currently in circulation, there are few academic publications documenting how these testing techniques are exactly used and combined in industrial projects. Performing empirical studies that involve professional software developers and testers can lead to a better understanding of how software testing techniques or strategies are used (e.g., the study of Erdogmus et al. [161]). The results from this research can be used to build the next generation of testing techniques and test tools. An added benefit of this line of research is that by providing cutting-edge testing techniques to the industrial partners helps with knowledge and technology transfer about testing from academia to industry.

Repository Mining

The a posteriori analysis of software evolution, through the mining of e.g. versioning systems, provides a view on how the software *has* evolved and on how the software *might* evolve in the future (also see Chapter 3).

Up until recently however, no specific research has been carried out in this context that looks at the co-evolution of the software system and its associated test suite. Zaidman et al. performed an initial study on how this co-evolution happens in open source software systems [562]. They offer three separate views that show (1) the commit behavior of the developers, (2) the growth evolution of the system and (3) the coverage through time. The major observation that was made is that testing is mostly a *phased* activity, whereas development is more continuous.

In the same context, further research might provide answers to questions such as:

- Is every change to the production code backed up by a change to the test suite? Are there specific reasons why this should or should not happen?
- Can IDE's provide warnings when adaptations to the production code lead to reduced quality of the test suite?

Test Coverage

Even when continuous testing is becoming more and more commonplace in the development process [448], determining the test coverage [346, Chapter 7] is often not part of the fixed testing routine. In combination with the findings of Elbaum et al. [159], who have determined that even minor changes to production code can have a serious impact on the test coverage, this might lead to situations where the testing effort might prove to be insufficient. As such, the development of features in integrated developments environments that preemptively warn against drops in test coverage will lead to a more efficient and thorough test process.

Regression Testing

Regression testing provides you with a safety net when letting software evolve, because it guards against introducing bugs into functionality that previously worked fine. Ideally, these tests should be run after each modification, but regression testing is often very expensive. Rothermel and Harrold provide a detailed survey of research in regression testing techniques, particularly in the domain of *selective* regression testing [445], where only that part of the regression test pertaining to the modification is re-run. Although selective regression testing can save costs, the process of determining which tests should be re-run is still expensive and the ultimate gain is thus relatively small. Further research into this topic is certainly warranted.