

Exploring the Composition of Unit Test Suites

Bart Van Rompaey and Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
{bart.vanrompaey2,serge.demeyer}@ua.ac.be

Abstract

In agile software development, test code can considerably contribute to the overall source code size. Being a valuable asset both in terms of verification and documentation, the composition of a test suite needs to be well understood in order to identify opportunities as well as weaknesses for further evolution. In this paper, we argue that the visualization of structural characteristics is a viable means to support the exploration of test suites. Thanks to general agreement on a limited set of key test design principles, such visualizations are relatively easy to interpret. In particular, we present visualizations that support testers in (i) locating test cases; (ii) examining the relation between test code and production code; and (iii) studying the composition of and dependencies within test cases. By means of two case studies, we demonstrate how visual patterns help to identify key test suite characteristics. This approach forms the first step in assisting a developer to build up understanding about test suites beyond code reading.

1 Introduction

Pushed by the adoption of agile development methodologies as well as the availability of free testing frameworks, a lot of unit tests have been written over the last few years. Such tests are specified persistently, thereby contributing to the size of a software project's artifacts. Studies report a ratio of test to production code which can extend till 2:3; occasionally even 1:1 [10, 23]. As such, unit testing considerably impacts a software project's development cost.

The benefits of unit testing are well known. In the short term, the application of unit testing results in software of higher quality [21, 19, 8]. Unit testing is observed to find other defects [22, 11] and is also reported to be considerably cheaper than strategies relying solely on testing later in the development cycle [11]. In the long term, unit tests are a valuable asset during regression testing, able to notice undesired side effects of changes [9, Ch.6].

On the down side, unit test code needs to co-evolve with production code in order to remain useful. Moreover, a test

suite is subject to the problem of design erosion as well, gradually losing the initially intended design and thereby becoming harder to understand and modify. Constructs in the tests that hinder modification, e.g. complex test cases or a resource dependent test [10], directly affect developer productivity thereby amplifying the overall maintenance cost. Studies indicate that regression testing can account for as much as one-third of the total cost of a software system [16].

Therefore, in the context of a legacy system, the associated test suite contains both opportunities, in the form of well designed, isolated unit tests with a high coverage, as well as weaknesses, in the form of maintenance intensive test cases, components lacking coverage, etc. Evaluating the overall condition first requires identifying the location of test code and relating it to the corresponding production code. A first notion of coverage per component can be obtained by comparing production and test code size-wise. Next, to explore the amount and kind of test cases for individual production components, a developer has to study their interdependencies. Detecting maintenance intensive test cases, finally, requires studying their internals: typically through code reviewing.

Code reviewing, with known reviewing rates around 150 to 200 lines of code per hour [4], was quickly found not to be scalable and therefore research went to look for design recovery techniques at a higher level of abstraction [5, 20]. We identified two ways in which general design recovery techniques can exploit the more constrained context of test code. First, in contrast to the heterogeneous design heuristics for production code, design guidelines for test code are quite strict, emphasizing recurrent design idioms such as the setup-stimulate-verify-tear down cycle (S-S-V-T). Secondly, the abstractions typically used in program comprehension - e.g. classes, methods, invocations etc. - lack testing semantics. Testers reason in terms of test cases, fixtures and assertions, suggesting a semantic layer on top of the abstractions employed in general design recovery.

Accordingly, this work contributes to the general body of knowledge on software visualization by introducing a test suite representation that does exploit the more constrained context of test code, allowing developers to explore the composition of a test suite, navigating between correspond-

ing production units and test cases, identifying co-evolution needs or spotting test design anti-patterns.

This paper is structured as follows. In Section 2, we recapitulate desired unit test characteristics. We clarify how the use of the S-S-V-T cycle as well as unit testing frameworks assist in composing well-structured tests. The visualization technique introduced in Section 3 exploits software design elements in its abstraction. Next, we present three visual presentations and discuss their interpretation (Section 4). In Section 5, we report about two case studies, the findings of which we validate by means of design documentation, reports as well as an interview with a developer. After discussing related work (Section 6) we wrap up (Section 7).

2 Test Suite Design

In this section, we briefly introduce terminology, design guidelines and strategies that are commonly used during unit testing.

Unit Testing Terminology – The standard unit testing terminology stems from Beck’s pattern system [3]:

- a *Unit under Test* is the set of production classes (classes that contribute to the final software product) that is exercised together during testing. In a strict unit testing approach a unit corresponds to a single class.
- a *Test Case* groups a set of tests performed on the same unit under test. Within JUNIT¹, a test case is specified as a class, inheriting from the generic `TestCase` class offered by the test framework.
- a *Test Case Fixture* is the set of attributes a test case requires to bring the unit under test into the desired state. The fixture consists of an instance of the unit under test as well as some shared test data.
- a *Test Command* is a container for a single test. It is typically encapsulated in a method of a test case.
- the *Test Case Setup* is a method of the test case in which the fixture is initialized into the desired state for testing. A corresponding *Test Case Tear Down* method releases resources again.

Design Guidelines – Unit test design guidelines propose a strict structure for specifying tests: (i) acquire and initialize the necessary resources, (ii) send one or more *stimuli* to the unit under test, (iii) verify that the unit responds properly; and finally (iv) release the acquired resources. These four calls are referred to as the setup-stimulate-verify-tear down cycle (S-S-V-T). The first step, performed in a Test Case Setup, is repeated before every Test Command in the

¹JUNIT is the Java implementation of the xUNIT family of testing frameworks, the *de facto* framework for unit testing today

Test Case. Each Test Command stimulates and verifies the unit under test.

Unit tests are typically specified in the same programming language as the system under test, yet are not tested extensively themselves. To support code reviewing as the main means for test quality assurance, test cases are required to be *concise*, *transparent in their objectives* and *isolated* in implementing the S-S-V-T cycle, forming an *encapsulated* test. Test Commands exercising the same unit under test are gathered in a Test Case, thereby sharing an *explicit* Test Case Fixture and Setup.

Unit Testing Strategies – The testing plan of a typical software system entails many strategies: unit tests verify the functionality of small units at a time, integration tests are focused on the interaction between components, system tests consider the behaviour of the system overall, etc. Despite the clearly distinctive objectives for each strategy, overlap between strategies occurs, such as unit tests bearing properties of another testing strategy:

- Units that are tightly coupled with many other units require more effort to isolate, e.g. by means of test stubs taking the place of external units. Therefore, a tester may decide to unit test without isolating - i.e. setting up a larger unit under test only part of which is the actual unit under test. Such a test case can be considered as being more integration testing.
- Certain units are always used by the same other (set of) units. During testing that unit might also be exercised via this set for ease of setup or because it closer resembles the actual usage scenario, resulting in multiple units to be considered. This testing approach is called Indirect testing. Moonen and van Deursen argue that this makes understanding and debugging harder [10].
- When large data sets are required for testing certain units, this data is sometimes stored in files and loaded during test setup. The reading functionality is typically abstracted and shared among test cases. Test code classes that implement this functionality are referred to as *test helpers*.
- System-wide input/output testing approach can be fed with test input data specifically chosen to exercise particular units.

Considering this variation, determining the kind of unit tests present in a test suite is worthwhile, as it may steer upcoming re-engineering tasks.

3 Visualizing Test Suites

In this work, we propose a visualization technique assisting re-engineers to analyze and comprehend the structure and quality of the test suite of large systems. To describe our visualization, we use the five-dimensional framework

of Maletic et al. [18]. This framework was proposed for development and maintenance of large-scale software and stimulates the user to describe how a technique assists in completing a particular software development task.

3.1 Tasks

The proposed visualization supports three program understanding tasks that we call *First Contact*, *Understand Unit(s)* and *Assess Test Case(s)*. Together, they form a top-down and phased approach to explore a test suite.

Perform First Contact. In this task, a developer builds up an overall mental model of a system at a high level of abstraction [9]. Initial understanding of the associated test suite is obtained by: (i) localizing the test suite code in the source tree, (ii) looking at overall coverage to get a notion on covered² as well as uncovered components; and (iii) studying test suite design to grasp the granularity of units that have been used.

Understand Unit(s). Koenemann and Koch observed that programmers only study code in case they are convinced of the relevance in the context of a particular task [17]. As unit tests also serve the purpose of live documentation, explaining in simple scenario’s how a unit is (and is not) supposed to be used, information about relevant test cases forms a next step towards the actual modification of the code. Secondly, coverage information can reveal important parts, as we assume that critical, frequently changing or core parts are tested more extensively.

Assess Test Case(s). After having identified relevant unit tests, i.e. test cases directly invoking production units of interest, the next step consists of evaluating the internal structure of individual test cases. Well-designed unit tests are most suited for documentation purposes, as they (i) are easy to understand; and (ii) specify how a particular unit is (and is not supposed to be) used in an isolated scenario. Test cases may present certain maintenance challenges as well. Weakly isolated test cases, requiring a complex setup or bearing unrealistic run-time expectations, become costly to maintain due to frequent changes, slow execution or seemingly random failures [10].

Summarizing, for three maintenance tasks we identified three information topics concerning the test suite: Test Location, Test Coverage and Test Design. Table 1 shows which information topics are required during the three tasks. For each of them we will introduce a separate view, i.e., a filter mechanism on the overall visualization technique.

²In the context of this paper, we use a coarse grained definition for coverage: a class is covered by a test case when at least one of its methods is invoked by a test command

Topic	Task <i>Perform First Contact</i>	<i>Understand Unit(s)</i>	<i>Assess Test Case(s)</i>
Test Location	✓	✓	
Test Coverage	✓	✓	
Test Design	✓		✓

Table 1. How are test suite exploration topics covered by the presented views?

3.2 Audience

The visualization technique is intended to assist software engineers in exploring the test suite of an unfamiliar system, as required in the following typical scenarios:

- A newcomer to the project who is asked to build up knowledge quickly and relatively independent from the existing team.
- A team of developers assigned to a major modification of a stabilized, long running legacy system may use it to characterize the available tests, thereby serving as a reference for communication.
- A re-engineer, asked to analyze a system X’s opportunities and threats when considered to be integrated into system Y, will be interested in the test suite as well.

3.3 Target

The target defines the characteristics of the software system to be visualized [18]. In this work, we are interested in the structure of test suites as well as the relationship between test suites and production code. In a first step, we fetch information from the system’s source code using a static fact extractor. This results in a model of the system according to the formalism of the Object Oriented Framework for Coupling and Cohesion (OOFCC) specified by Briand et al. [6], i.e. a formalism to query and count in terms of classes, methods, invocations, etc. Secondly, we identify entities that belong to test code and adapt the model representation according to the OOFCC refinement that we proposed for test code that (i) formalizes unit test concepts as entities and relationships in a test model, (ii) describes how OOFCC entities map onto test concepts for common implementations of xUnit and (iii) provides the heuristics (type, inheritance and ownership properties as well as naming conventions) required to implement a refining model transformation [26]. Due to space constraints, we do not reproduce the formalism here. Next, for each view we query this model and compose the input for a graph visualization tool. These steps are automated in a tool called FETCH³.

The model entities of [26] were already introduced (informally) in Section 2 (Unit Testing Terminology). We fur-

³stands for Fact Extraction Tool CHain, developed at the University of Antwerp. Available at <http://www.lore.ua.ac.be/Research/Artefacts/>

thermore distinguish three types of relations:

- *Containment* relations represent the hierarchical decomposition of a software system in containers. Classes belong to a parent package, methods belong to a class, etc. This decomposition applies to both production and test code.
- A *coverage* relation is a relation between a test entity *A* and a production entity *B*, where *A* has at least one invocation towards *B*. As such, we only consider direct relations between production and test code, in line with a developer’s information needs during the exploration of a test suite’s composition, rather than obtaining finer-grained, exact coverage measures.
- *Test dependencies* are relations between test entities, e.g., revealing certain abstractions or recurrent helper functions in the test design. In practice, such mechanisms are typically implemented in an (abstract) test case superclass. As such, we denote an inheritance relation between two test cases as a test dependency.

3.4 Representation & Medium

Directed graphs have been shown to be natural representations of software systems [15, 24]. Table 2 explains which entities and relations we represent as nodes and edges. The symbols for class and test case vary per view. For the ‘medium’ dimension, characterizing where the visualization is to be rendered [18], we adopted GUESS. GUESS is an *exploratory data analysis and visualization tool for graphs and networks* [1], as a part of FETCH. This environment assists the user in graph exploration through capabilities such as applying graph layouts, highlighting, zooming and moving as well as customizable filtering.

Model Entity	Type	Prod/Test	Repr.
Package	Node	Prod	□
Class	Node	Prod	□/○
Method	Node	Prod	○
Test Case	Node	Test	■/●
Test Command	Node	Test	●
Containment	Edge	-	→
Coverage	Edge	-	→
Dependency	Edge	-	→

Table 2. Visualization Legend

4 Three Test Suite Views

In this section we present three views as filters on the overall graph representation introduced above. Each view corresponds to an exploration task. We expand upon the intent and motivation for each view, and the interpretation that should be given to any of the observed indicators.

4.1 System-wide Test Suite View

The core of this visualization is the hierarchical decomposition of a software system into packages and classes. The filter we apply on the graph therefore skips all entities below the class level. To distinguish packages entities from class entities we use square and circle shapes respectively. The Graph EMbedder (GEM) algorithm applied on the containment edges provides both an easy to interpret as well as aesthetically pleasing layout [13].

In line with the re-engineering pattern “Study the exceptional entities” [9], entities exhibiting either a lack of, oppositely, plenty of incoming and outgoing edges deserve special attention. In the former case, one has to look further for other testing strategies. In the latter case, an important role during testing can be assumed.

4.1.1 Test Location

Intent. Localize the system test code.

Motivation. Typically, localization of test code is a configuration management responsibility, as it is a consequence of source tree structure in the version control system. Feathers discusses pro and contras of possible test code localization: the whole test suite may be gathered in a common location, test cases may be stored per component, but may as well reside among production code [12].

Interpretation. A consistent location means that we identified earlier project conventions that we can rely on. In case of no dominant visual indicators, we can deduce the absence of such conventions. The view still helps to determine the test code associated with a particular component.

Visual Indicators. We demonstrate two kinds of locations deducible from the system-wide view (see Figure 1).

- Test cases located in the *same package* as production code will result in package nodes containing both white and black nodes.
- We observe two packages, one filled with white nodes next to one with black nodes, connected by coverage edges when test code resides in *another package* than production code.

4.1.2 Test Coverage

Intent. Obtain a basic notion of test coverage.

Motivation. The desired notion of test coverage is cheap to obtain and scalable. It helps to make assumptions about earlier test efforts and system-critical components. Observing coverage for evolving components gives an impression about the risks (and possible counteractions) for further modifications.

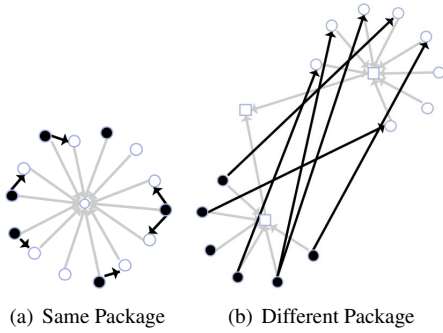


Figure 1. Possible locations of test cases

Interpretation. Components that are not directly tested might be trivial (e.g. data holders), have been decided only to be tested in conjunction with other components or might not have been tested, e.g., due to time constraints. For stronger tested components we assume a more important role such as being critical, frequently changing, belonging to the system core, etc. Assumptions need to be verified further on in subsequent exploration tasks, eventually by obtaining actual, fine-grained coverage measures.

Visual Indicators. Components not covered by unit tests will show as clusters of classes (i) without test cases in the same package; and (ii) without incoming test coverage edges (e.g., Figure 2(a)). The components in Figure 1 serve as examples of better covered components. *Highly covered classes*, such as class A in Figure 2(b), are represented as nodes receiving many test coverage edges.

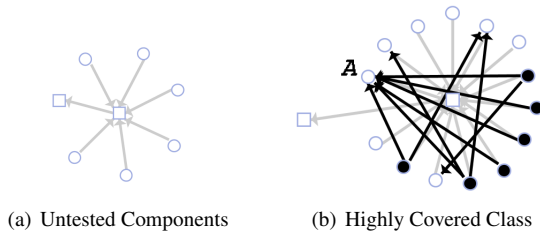


Figure 2. Test Coverage Indicators

4.1.3 Test Design

Intent. Grasp the overall test suite design.

Motivation. The test design reveals first hand information on what kind of testing strategies have been applied in the past. This tells us at which point such tests become most useful as well as how difficult tests will be to understand and modify.

Interpretation. To grasp testing strategies, we mainly look at the size of a unit and the presence of test helpers. Units of a limited size (e.g. a single class) can play an important role as test harness for local changes, due to being easier to understand and modify. When units are larger or when

dealing with a more integration testing style, tests are more suited to give feedback about the overall status of a modified system. Such tests risk to be more complex and change sensitive, however, due to the many dependencies. Test helpers are typically used to abstract away recurring setup, stimulate or verification behavior, but also to facilitate access to more complex test data and input/output tests. As reusable entities, test helpers help to avoid duplication in test code.

Visual Indicators. Units tested in isolation are shown as package-clustered nodes receive a limited number of edges from similarly clustered test case nodes (Figure 3(a)). We identify *indirect tests* in case test cases do not cover the units expected from the identified test location, but rather (i) access units in other packages or (ii) multiple test cases target a single unit in a package as in Figure 3(b). A test case receiving many test dependency edges, such as A in Figure 3(c), serves, at least partially, as *test helper* and forms as such an opportunity during test suite extension.

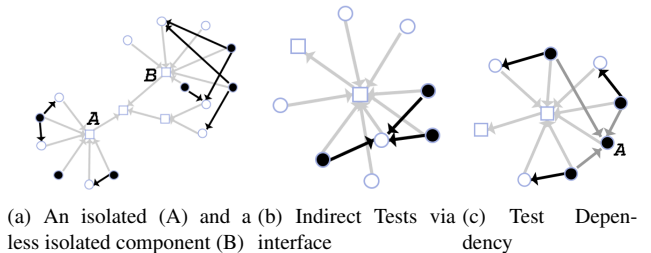


Figure 3. Test Design Indicators

4.2 Unit under Test View

This view focuses on an individual production class visualized in terms of its accessible methods. Test case invoking one or more methods of this unit are displayed as well, with coverage edges drawn from the test commands to the involved methods. To distinguish classes from their methods, these entities are drawn as squares and circles respectively.

4.2.1 Test Location

Intent. Identify test cases for a particular unit under test.

Motivation. During evolution of component, a developer gathers the set of production classes as well as test cases that require modification.

Interpretation. Depending on past test strategies, a unit can be exercised by one or more test cases. In case unit testing as well as integration testing are specified in test code, units are covered by test cases of each strategy. Test cases might also be split up when growing too large [10].

Visual Indicators. Trivially, production classes that are not covered directly, and as such are not likely to be the

subject of a focused unit test, will show up in the view as *untested units* – components without black test nodes. Every *involved test case* will be represented in terms of its test commands. Coverage edges make the relationship with the unit explicit (see Figure 4).

4.2.2 Test Coverage

Intent. Which parts of a unit are being tested?

Motivation. Once the scope of interest for a certain maintenance task has been reduced to a couple of units, the Unit under Test view presents how these units are covered by test commands of involved test cases. These test commands are worth exploring in detail, because of their documentation power as well as their co-evolution needs.

Interpretation. Complementary to assumptions derived from studying the location of involved test cases, in the context of coverage assessment the focus lies on identifying combinations of production methods being exercised. This allows the developer to understand which methods are not directly tested, methods covered by means of simple scenarios and methods tested together within a test command.

Visual Indicators. An example of *Multi-Test Case Coverage* is shown in Figure 4: a unit receiving coverage edges from multiple test cases A, B and C. If only A would have existed, we encountered a unit with *partial coverage*, i.e. only a small part of its methods are directly covered.

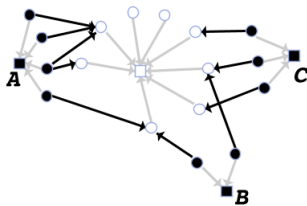


Figure 4. Multi-Test Coverage

4.3 Test Case View

The Test Case View centers around individual test cases, which are represented in terms of the S-S-V-T entities and the exercised production units. Again, classes are represented as squares; methods as circles. In addition to the nodes introduced in the visualization technique, this view adds two meta-nodes named *Fixture* and *Test Commands*, thereby making these two test concepts explicit.

4.3.1 Test Design

Intent. Identify opportunities in the form of well designed test cases as well as possible maintenance threats by studying the internal structure of selected test cases.

Motivation. Test cases that are designed according to strict unit test design guidelines, with explicit fixture and concise setup and test commands, are an opportunity to understand (i) typical usage of the unit under test as well as (ii) how the test suite covers such units. Integration-style test cases demonstrate how components interact. Using method-level information a developer can better motivate whether a certain test case is a possible threat or rather a manifestation of a certain test strategy.

Interpretation. Deviations from the design guidelines are potential maintenance threats. A list of complex and thus undesired test structures can be found in [10].

- Test cases can lack an explicitly defined fixture, thereby removing the distinction between the defined unit of interest and surrounding helper units.
- A test command invoking many production methods, possibly from multiple production classes, entails a complex (integration) test scenario.
- Test cases with a large fixture that is only partially used by individual test commands indicates that the contained test commands do not logically belong together, therefore violating the guidelines of encapsulation and transparency in test objective.

Visual Indicators. Figure 5(a) shows an example of a *well designed test case*. It contains an isolated and explicit fixture, the methods of which are consistently tested by single test commands. The *Lack of Explicit Fixture* (Figure 5(b)) renders a test case more difficult to understand, as the common unit under test (if at all present) is implicitly interwoven within every single test command. Making the fixture explicit implies introducing a test case attribute that is initialized by the Test Case Setup. A *Complex Test Scenario* in a test command can be recognized by the many coverage edges that target production methods (e.g. test command A in Figure 5(c)). Moreover, the *Large Fixture* of this test case is diagnosed by the many production class entities in the fixture that are extensively, yet not fully, shared among the test commands. Within a unit test suite, we identify the more *Integration Test* type of test case (e.g., Figure 5(d)) by the multiple production classes that are accessed without a dominant unit under test.

5 Case Studies

In this section we report on two case studies to evaluate our visualization technique. We used FETCH to statically extract the required information and compose the graphs. As a first project, we selected the open source build system APACHE ANT, a middle-sized, industry-strength software system. Secondly we analyzed a small system, CPP2FAMIX, that is developed using a strict unit testing approach. This allows us to confront test suites resulting of two testing

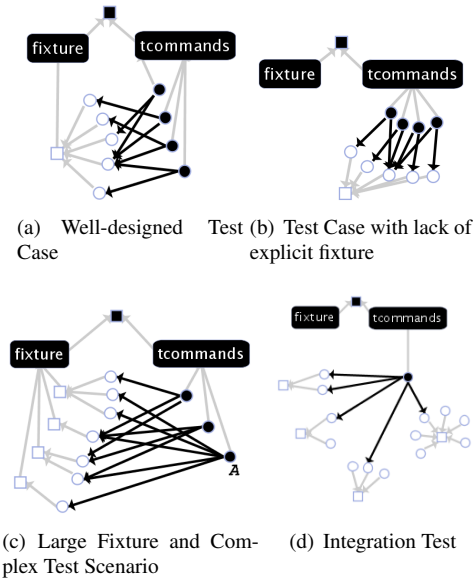


Figure 5. Test Design Indicators

strategies. For each case study, we undertake the three identified program understanding tasks and formulate our findings based upon what we derive from the three views. Next, we validate these findings by skimming through the documentation, by looking at external references or, in the case of CPP2FAMIX, by interviewing the developer.

5.1 Apache Ant

As a first case study we use the well-known APACHE ANT project. The 1.6.5 release consists of about 104 kSLOC, 18 kSLOC (17%) of which is JUNIT test code.

5.1.1 Findings

First Contact. Figure 6 gives an overview of the system-wide view for test suites, based on part of the system core (due to scalability constraints on paper, we filtered out entities and relations other than the core packages *o.a.t.ant*⁴ and *o.a.t.ant.taskdefs*). We observe the presence of a considerable amount of test cases, residing in the same packages as production code. For packages outside of the core, the testing strategy seems different: some components seem not tested at all (e.g. *o.a.t.ant.helper*), others are tested in isolation (e.g. *o.a.t.zip*). Although located among production classes, we notice that ANT’s test cases do not cover the units in the same package. Combined with the fact that many tests are covering *o.a.t.ant.Project* and depend upon *o.a.t.ant.tools.BuildFileTest*, we assume an indirect testing approach. Next to *o.a.t.ant.Project*, we identify *o.a.t.ant.util.{FileUtils,JavaEnvUtils}* and

⁴abbreviation for org.apache.tools

o.a.t.ant.types.{AbstractFileSet,Path} as key classes through their extensive test coverage.

Understand Units. In the 1.6.4 release of ANT, several bugs were found in the *directory scanner* as well as the *unzip* and *untar* features⁵. Therefore, we analyze the existing testing facilities for these units. The concept of a directory scanner is implemented in the class *o.a.t.ant.DirectoryScanner*. Using the Unit under Test view in Figure 7, we note that four test cases exercise this production class: one test case is exercising eight out of twenty-two production methods, the other three invoke just two methods. Therefore, we derive that the actual unit test for this unit is *o.a.t.ant.DirectoryScannerTest*, while the three other test cases are only using the directory scanner as a helper unit. Indeed, these test cases only invoke so called *getter* methods, fetching data from the *DirectoryScanner* object to evaluate the expected result for their unit under test against the actual test result.

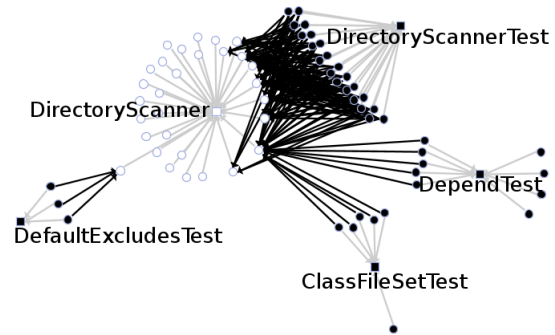


Figure 7. Unit o.a.t.DirectoryScanner.

The *untar* functionality is implemented in *o.a.t.ant.taskdefs.Untar*, which shows as untested. However, we do know, from the System-Wide View, that for the ANT project test cases are located in the same package as the production classes. We identify *UntarTest* as the actual test case based on naming.

Assess Test Cases. *o.a.t.ant.DirectoryScannerTest* is observed to be a test command-rich test case. Most of these test commands appear to be similar in composition, targeting the same side objects of *Project* and *FileUtils*. *UntarTest* is characterized by indirect testing behaviour by test cases relaying via *BuildFileTest*, *Project* and *FileUtils*. Neither of the two test cases has an explicitly defined fixture. Summarizing, both test cases make use of key system classes to exercise the unit under test.

5.1.2 Validation

Using the code coverage tool EMMA⁶, we compute a method coverage of 65% (80% class coverage) for ANT,

⁵mentioned in the release notes of version 1.6.5

⁶<http://emma.sourceforge.net/>

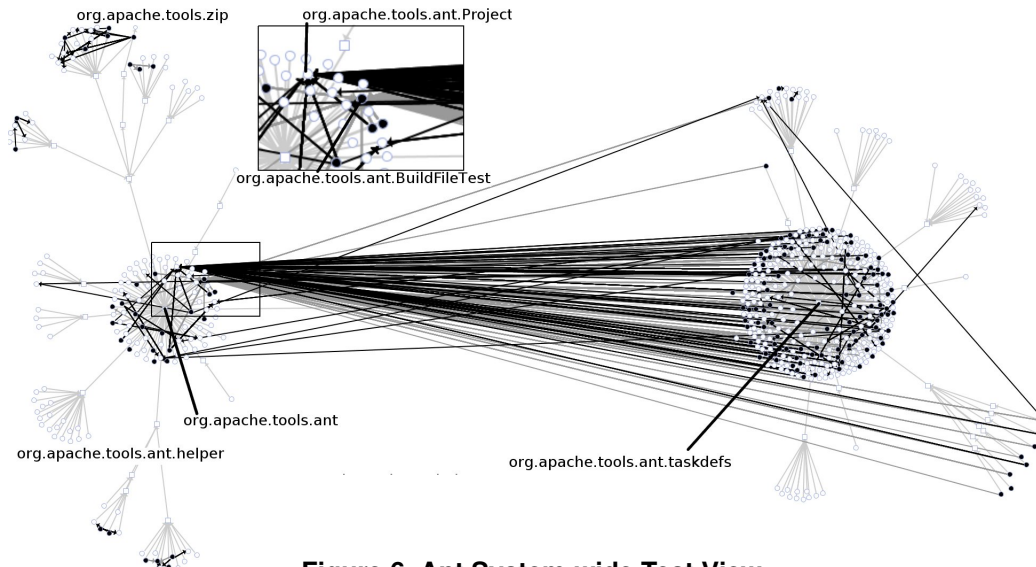


Figure 6. Ant System-wide Test View

confirming our initial impression of a reasonably tested system. From ANT’s documentation⁷, we derive that the key classes in the design as identified by its architects are a.o. *Project*, *Task* and *Target*. The documentation header of *o.a.t.ant.Project* describes this class as the central representation of an ANT project. As this class also provides the means to start a build, its frequent usage by test cases confirms the indirect testing approach in which several project scenarios are constructed, executed and verified using this generic Project class. The focus of ANT on Java software and the fact that build instructions are specified in XML files explains the importance of the other classes we noted. By looking into one of the test cases making use of the *FileUtils* class, we were able to find the XML files containing test data in the distribution. Thus, ANT’s documentation confirms our assumption that production classes invoked by many test cases play a major role in the system itself.

Our claim that *o.a.t.ant.tools.BuildFileTest* is an important test class gets backed up by Van Geet and Zaidman, who identify it as *an abstraction of a unit test that uses a build file as test data* [25]. For each test run, a Project instance is created which loads this XML file and executes the contained build instructions. This class thus indeed serves as a *test helper*, used by no less than 414 test commands.

The ANT case reveals a major limitation of our visualization technique: when dealing with indirect tests, we fail to trace coverage relationships between test cases and corresponding unit under test. We argue however that in a first contact phase, where overall system comprehension as well as identification of components worth investigating further are the prime objectives, information such as actual, complete coverage measurements requiring dynamic analysis are too expensive and less suited for visualization. Given

⁷http://www.codefeed.com/tutorial/ant_config.html

the underlying test model, graph querying is also a viable alternative to reveal indirect testing relations.

5.2 CPP2FAMIX

As a second case study we opt for the CPP2FAMIX, a C++ fact extractor of about 13.5 kSLOC Java code. CPP2FAMIX extracts information about a C++ software system out of the AST unit dumps of the GNU Compiler Collection. This information is transformed into a re-engineering model. The JUNIT test suite accounts for 29% of the overall system size. We chose this system because the developer is a colleague of ours, hence we can thoroughly interview him.

5.2.1 Findings

First Contact. From Figure 8, we derive a consistent, per production class unit testing approach. The unit test code resides in a subpackage *test* of each component, except for four components that are weakly or even completely uncovered. Furthermore, we identify *cpp2famix.test.TestGCCTreeDumpParser* as an integration test, exercising the parsing and filtering of a GCC tree dump into the system’s internal tree representation. For test dependencies, we noticed *test helpers* *cpp2famix.node.traversal.test.NodeTraversalTest* and *cpp2famix.test.TestWithTreeFragment*, helping test cases with traversing AST representations and with composing small test data trees respectively.

Understand Units. The developer points out six production classes that are currently being modified: *ClassExtractor*, *FieldExtractor*, *FieldsIterator*, *Attribute*, *Clazz* and

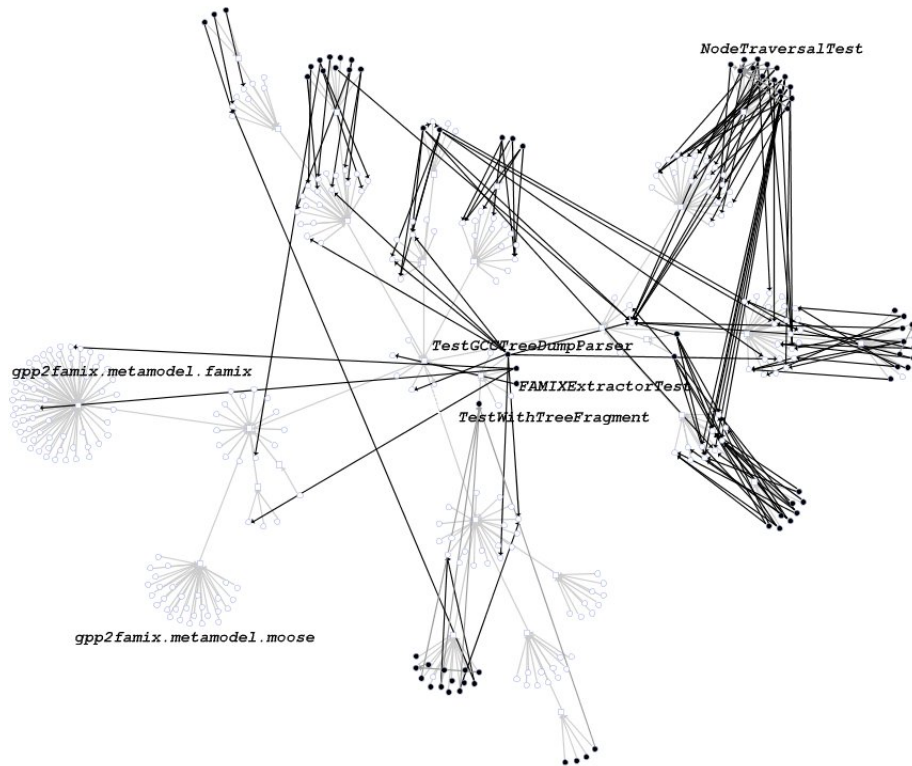


Figure 8. `cpp2famix` System-wide Test View

StatementIterator. Using the Unit under Test Views, we identify and navigate to the test cases involved:

- *Attribute* and *Clazz* are not directly tested as they belong to generated code, conform to our earlier findings using the System-Wide view.
- *ClassExtractor*, *FieldIterator* and *StatementIterator* are exercised by corresponding **Test* test cases.
- *FieldExtractor* is covered by *FieldsIteratorTest* as well as by *FieldExtractorTest*.

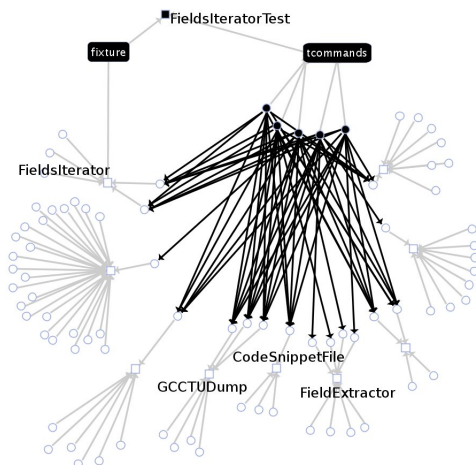


Figure 9. Test case `FieldsIteratorTest`

Assess Test Cases. From the Test Case View, we deduce that quite some helper objects are needed to test the behavior of the Extractor and Iterator classes. The class names of the test helpers, however, reveals that sample pieces of data are composed to exercise the units under test (Figure 9). As such we conclude that these are broadly isolated unit tests.

5.2.2 Validation

During an interview, we confronted the system’s developer with our analysis. He testifies that a tight unit testing approach (using JUNIT) has been undertaken, with test cases being written either just before (test-driven) or just after the corresponding production code. This results in a class and method coverage of 90% and 79% respectively. The developer acknowledges the presence of untested components, explaining that he did not see the need to test the generated components `cpp2famix.metamodel.famix` and `cpp2famix.metamodel.moose`. For the classes in `cpp2famix.metamodel`, he commented that we were looking at dead code that was replaced by the classes in the two subpackages. `cpp2famix.extractors.*`, at last, only contains simple data holders and as such it was not considered worthwhile to be unit tested.

`cpp2famix.test.TestGCCTreeDumpParser` is confirmed to be an integration test, but the developer stated that it

is an old test that even failed when we tried to execute it. Instead, he points to `cpp2famix.test.FAMIXExtractorTest` as being the current integration test, although it is not implemented in a traditional S-S-V-T style, but merely a "user" of the top level production class. That explains why we did not identify this test case as one that deserves special attention.

6 Related Work

We identified the following work in the domain of test suite reverse engineering.

Agrawal et al. introduce a set of techniques to enhance program understanding, debugging and testing [2]. Among others, the χ Suds tool suite contains tools to assist developers in achieving high test coverage, locating errors as well as minimizing regression sets. Via source code coloring, the developer perceives the coverage level, erroneous locations or execution frequency.

Gaelli et al. observe that not all unit tests are alike [14]. Therefore, a taxonomy that distinguishes unit tests based on the focus on one or more methods, type of expected outcome, etc. Their automated classification approach for SUnit tests using heuristics achieves a high overall precision (89%) and a moderate recall (52%). One of the steps the authors identify as future work involves making explicit the relationship between unit tests and methods under test.

Van Geet and Zaidman hypothesize that unit tests covering multiple units are less suited as documentation as such tests are harder to understand [25]. In a case study involving the ANT project, the median number of methods executed by a test command is more than 200, which make them conclude that the test suite of this particular project is not well suited for documentation purposes.

To gain knowledge about the inner working of a software system, Cornelissen et al. use sequence diagrams obtained from test execution [7]. The use of abstraction, separation of test stages and stack depth limitations make such diagrams scalable.

7 Conclusion & Future work

In this work, we proposed a visualization technique assisting re-engineers to explore the composition of an object-oriented system's unit test suite. We propose three graph-based views, representing (aspects of) a test suite in terms of the S-S-V-T cycle's test concepts. These views assist a re-engineer in building initial understanding and assessing opportunities and weaknesses for further evolution. We describe how certain visual indicators in these views reveal information about the location of test cases, the coverage level (in an exploration context) as well as the followed unit test strategy.

We validated the technique by means of two case studies. In the first one, we compared the results of our analysis with ANT's system documentation as well as with finding of other authors. Our initial findings regarding coverage, key system classes as well as test design were confirmed. Secondly, we investigated a system which has been developed with a tight unit testing approach. The lead developer of this system, CPP2FAMIX, confirmed most of our claims about the test suite.

Based on these two case studies, we conclude that the visual exploration technique, as a first contact technique, serves its purpose. As a next step in the reverse engineering of test suites, we identify a need for finer-grained analysis, such as (i) obtaining actual coverage measurements via test execution and (ii) incorporating information about size and complexity of components for a more detailed assessment. We identify the integration of such information, e.g., via polymetric views, as future work.

Acknowledgements – This work was executed in the context of the ITEA project if04032 entitled *Software Evolution, Refactoring, Improvement of Operational&Usable Systems* (SERIOUS) and has been sponsored by IWT, Flanders.

References

- [1] E. Adar. Guess: a language and interface for graph exploration. In *Proc. of the SIGCHI Conf. on Human Factors in computing systems*, pages 791–800. ACM Press, 2006.
- [2] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.
- [3] K. Beck. Simple smalltalk testing: With patterns. *The Smalltalk Report*, 4(2):16–18, 1994.
- [4] F. Belli and R. Crisan. Empirical performance analysis of computer-supported code-reviews. In *Empirical Performance Analysis of Computer-Supported Code-Reviews*, page 245. IEEE Computer Society, 1997.
- [5] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.
- [6] L. Briand, J. Daly, and J. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.
- [7] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'07)*, 2007.
- [8] L. Crispin. Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):70–71, 2006.
- [9] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.
- [10] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok. Refactoring test code. In *Proc. Extreme Programming and Flexible Processes (XP)*, pages 92–95, 2001.
- [11] M. Ellims, J. Bridges, and D. C. Ince. Unit testing in practice. In *Proc. of the 15th Int'l Symp. on Softw. Reliability Engineering*, pages 3–13, 2004.

- [12] M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.
- [13] A. Frick, H. Mehdau, and A. Ludwig. A fast adaptive layout algorithm for undirected graphs. In *Proceedings of Graph Drawing '94, LNCS 894*, pages 388–403. Springer, 1994.
- [14] M. Gaelli, M. Lanza, and O. Nierstrasz. Towards a taxonomy of SUnit tests. In *Proceedings of 13th International Smalltalk Conference (ISC'05)*, Sept. 2005.
- [15] B. Gulla. Improved maintenance support by multi-version visualizations. In *Proc. of the 8th Int'l Conf. on Softw. Maintenance*, pages 376–383. IEEE Computer Society, 1992.
- [16] M. Harrold. Testing: a roadmap. In *Proc. of the Conf. on The Future of Software Eng. (ICSE 2000)*, pages 61–72, 2000.
- [17] J. Koenemann and S. Koch. Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI conference on Human factors in computing systems (CHI'91)*, pages 125–130, 1991.
- [18] J. Maletic, A. Marcus, and M. Collard. A task oriented view of software visualization. In *Proc. 1st Int'l WS on Visualizing Softw. for Understanding and Analysis*, page 32, 2002.
- [19] E. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Proc. of the 25th Int'l Conf. on Softw. Eng.*, pages 564–569, 2003.
- [20] M.Lanza and S.Ducasse. Polymetric views – a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.
- [21] G. Murphy, P. Townsend, and P. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, September 1994.
- [22] P. Runeson and A. Andrews. Detection or isolation of defects? an experimental comparison of unit testing and code inspection. In *Proc. of the 14th Int'l Symp. on Softw. Reliability Eng.*, pages 3–13, 2003.
- [23] R. Sangwan and P. Laplante. Test-driven development in large projects. *IT Pro*, 8(5):25–29, 2006.
- [24] T.Ball, J.Kim, A.Porter, and H.Siy. If your version control system could talk. In *Proc. of the ICSE Workshop on Process Modelling and Empirical Studies of Software Eng.*, 1997.
- [25] J. Van Geet and A. Zaidman. A lightweight approach to determining the adequacy of tests as documentation. In *Proc. of the 2nd Workshop on Program Comprehension through Dynamic Analysis*, pages 21–26, October 2006.
- [26] B. Van Rompaey, B. Du Bois, and S. Demeyer. Characterizing the relative significance of a test smell. In *Proc. of the 20th Int'l Conf. on Softw. Maintenance*, pages 391–400, 2006.