Delft University of Technology
Software Engineering Research Group
Technical Report Series

Mining Software Repositories to Study Co-Evolution of Production & Test Code

Andy Zaidman, Bart Van Rompaey, Serge Demeyer and Arie van Deursen

Report TUD-SERG-2008-003

TUD-SERG-2008-003

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

Note: Accepted for publication in the Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST2008), 2008, IEEE Computer Society.

Mining Software Repositories to Study Co-Evolution of Production & Test Code

Andy Zaidman[1], Bart Van Rompaey[2], Serge Demeyer[2], and Arie van Deursen[3]

[1]*Delft University of Technology, The Netherlands – a.e.zaidman@tudelft.nl*
[2]*University of Antwerp, Belgium – {bart.vanrompaey2, serge.demeyer}@ua.ac.be*
[3]*Delft University of Technology & CWI, The Netherlands – arie.vandeursen@tudelft.nl*

Abstract

Engineering software systems is a multidisciplinary activity, whereby a number of artifacts must be created — and maintained — synchronously. In this paper we investigate whether production code and the accompanying tests co-evolve by exploring a project's versioning system, code coverage reports and size-metrics. Our main aim for studying this co-evolution is to create awareness with developers and managers alike about the testing process that is followed. We explore the possibilities of our technique through two open source case studies and observe a number of different co-evolution scenarios. We evaluate our results both with the help of log-messages and the original developers of the software system.

1 Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less useful [15]. When evolving software, the source code is the main artefact typically considered, 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, e.g. specifications, constraints, documentation, tests, etc. [17].

In this paper we explore two dimensions of the multidimensional software evolution space, as we focus on how tests evolve with regard to the related source code. In order to study the co-evolution of production and test code, we rely on the data that is stored in version control systems (VCS's). Using a VCS to study *co-evolution* implies the prerequisite that the tests should be committed to the VCS alongside the production sources. Therefore, our primary focus for this study is the co-evolution of production code versus persistent software tests such as unit and integration tests.

Knowing the necessity of a software system's evolution, the importance of having a test suite available and the cost implications of building [4, 14] (and maintaining) a test suite, we wonder how test and production code co-evolve during a software project's lifetime. We understand that, ideally, test code and production code should be developed and maintained synchronously, for at least two reasons:

- Newly added functionality should be tested as soon as possible in the development process, e.g. via unit testing [19].
- When changes, e.g. refactorings, are applied, the preservation of behavior needs to be checked [6, p. 159].

In this context, Moonen et al. have shown that even while refactorings are behavior preserving, they potentially invalidate tests [18]. Elbaum et al. concluded that even minor changes in production code can have serious consequences on test coverage, or the fraction of production code tested by the test suite [7]. These observations reinforce the claim that production and test code need to co-evolve.

This leads to the almost paradoxical situation whereby tests are quasi essential for the success of the software (and its evolution), while also being a serious burden during maintenance. It is exactly this paradox that has lead us to study the co-evolution of production and test code. In this context we propose to use lightweight techniques and visualizations, which, as Storey et al. observed, are common to the field of studying software evolution [20].

For this study then, our main question is: *How does testing happen in open-source software systems?* In order to steer our research, we refine this question into a number of subsidiary research questions:

RQ1 Does co-evolution happen synchronously or is it phased?

RQ2 Can an increased test-writing effort be witnessed right before a major release or other event in the project's lifetime?

RQ3 Can we detect testing strategies, e.g., test-driven development [16]?

RQ4 Is there a relation between test-writing effort and test coverage?

In a more practical context, knowing an answer to these questions provides us with the opportunity to gain a deeper insight into the current practice of testing in the real world. This allows:

- Software engineers to quickly *assess* the testing process in the light of future maintenance operations, e.g. during first-contact situations [6].
- Quality assurance to *monitor* the testing process, to identify trends and to compare the observed process against the intended process.

In this paper we set up an experiment in which we study the co-evolution of production and test code of two open source software systems. Subsequently, we evaluate our findings internally, by means of log messages that were written during development, and externally, by presenting our findings to the original developers and recording their remarks.

The structure of this paper is as follows. The next section introduces three views on the two-dimensional software evolution space, followed by Section 3 clarifying the evaluation procedure for these views. Sections 4 and 5 then present our two case studies on respectively CheckStyle and ArgoUML. While Section 6 provides discussion, Section 7 relates our work to other work in the field, and Section 8 presents our conclusion and future work.

2 Test co-evolution views

As studying the history of software projects involves large amounts of data, we make use of visualizations to answer evolution-related questions. More specifically, we introduce three distinct, yet complementary views, namely:

1. The **change history view**, wherein we visualize the *commit*-behavior of the developers.
2. The **growth history view** that shows the relative growth of production code and test code over time.
3. The **test quality evolution view**, where we plot the test coverage of a system against the fraction of test code at discrete times.

2.1 Change History View

Goal. With the change history view, we aim to learn whether (i) production code has an associated (unit) test; and (ii) whether these are added and modified at the same time. As such, we seek to answer RQ1 and RQ3.

Description. In this view:

- We use an XY-chart wherein the X-axis represents time and the Y-axis source code entities.
- We make a distinction between production files and test files. A unit test is placed on the same horizontal line as its corresponding unit under test. Furthermore, we also distinguish between files that are introduced and files that are modified based upon the data obtained from the VCS.
- We use colors to differentiate between newly added (red square) and modified production code (blue dot); newly added (green triangle) and modified tests (yellow diamond).

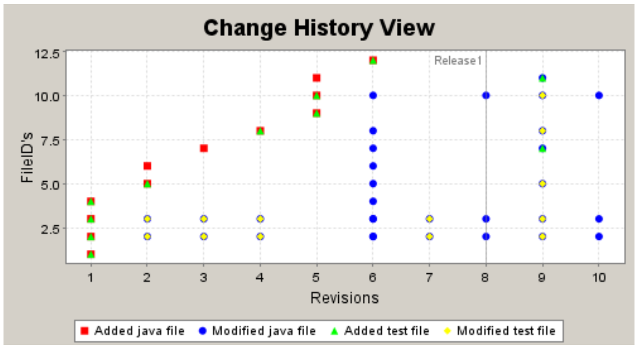


Figure 1. Example Change history view.

Interpretation. Consider the example view in Figure 1, created from synthetic data. We are looking for patterns in the plotted dots that signify co-evolution. Test files introduced together with the associated production units are represented as green triangles plotted on top of red squares. Test files that are changed alongside production code show as yellow diamonds on top of blue dots. Vertical green or yellow *bars* indicate many changes to the test code, whereas horizontal bars stand for frequently changed files. Other patterns not specifically involving the tests, e.g., vertical or horizontal blue bars, have been studied by others [9, 22].

Technicalities. The connection between production and test code is established on the basis of file naming conventions (e.g., a test case that corresponds to a certain production class has the same file name with postfix *“Test”*). Unit tests that cannot be correlated are considered to be integration tests and are placed on the top lines of the graph.

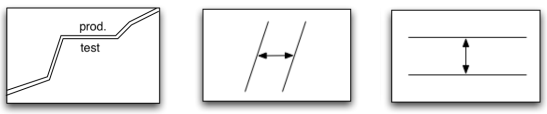
Trade-off. The change history view is mainly aimed at investigating the development behavior of the developers. However, it provides no information regarding, e.g., the total size of the system (throughout time) or the proportion of test code in the system. It also does not show the size-impact of a change. For these reasons, we introduce the growth history view in the next section to complement the change history view.

2.2 Growth History View

Goal. The aim of the growth history view is to identify growth patterns indicating (non-)synchronous test and production code development (RQ1), increased test-writing effort just before a major release (RQ2) and evidence of test-driven development (RQ3).

Description. In this view:

- We use an XY-chart to plot the co-evolution of a number of size metrics over time.
- The five metrics that we take into consideration are: Lines of production code (pLOC), Lines of test code (tLOC), Number of production classes (pClasses),



(a) Synchronous (b) Time Delay (c) Test Backlog

Figure 2. Example patterns of synchronous co-evolution.

Number of test classes (tClasses) and Number of test commands[1] (tCommands).

- Metrics are presented as a cumulative percentage chart up to the last considered version (which is depicted at 100%), as we are particularly interested in the co-evolution and not so much in the absolute growth.
- The X-axis is annotated with release points.

Interpretation. First of all, we can observe phases of relatively weaker or stronger growth throughout a system's history. Typically, in iterative software development new functionality is added during a certain period after a major release, after which a "feature freeze" prevents new functionality to be added. At that point, bugs get fixed, testing effort is increased and documentation written.

Secondly, the view allows us to study growth co-evolution. We observe (lack of) synchronization by studying how the measurements do or do not evolve together in a similar direction. The effort of writing production and test code is spent synchronously when the two curves are similar in shape (see Figure 2(a)). A horizontal translation indicates a time delay between one activity and a related one (2(b)), whereas a vertical translation signifies that a historical testing or development backlog has been accumulated over time (2(c)). Such a situation occurs, e.g., when the test-writing effort is lagging the production code writing effort for many subsequent releases. In the last version considered in the view, both activities reach the 100% mark, which is explained through the fact that we are measuring relative efforts for both activities.

Thirdly, the interaction between measurements yields valuable information as well. In Table 1 a number of these interactions are outlined. For example, the first line in Table 1 states that an increase in production code and a constant level of test code (with the other metrics being unspecified) points towards a "pure development" phase.

Technicalities. To separate production classes from test classes we use regular expressions to detect JUnit test case classes. As a first check, we look whether the class extends `junit.framework.TestCase`. If this fails, e.g., because of an indirect generic test case [21], we search for a combination of `org.junit.*` imports and `setUp()` methods.

Counting the number of test commands was done on the basis of naming conventions. More specifically, when we found a class to be a test case, we looked for methods that

[1]A test command is a container for a single test [21].

pLOC	tLOC	pClasses	tClasses	tCommands	interpretation
↗	→				pure development
→	↗				pure testing
↗	↗				co-evolution
→	↗	→	→		test refinement
→	→	↗	↗		skeleton co-evolution
	→		↗		test case skeletons
	→			↗	test command skeletons
→	↘				test refactoring

Table 1. Co-evolution scenarios.

would start with `test`. We are aware that with the introduction of JUnit 4.0, this naming convention is no longer necessary, but the projects we considered still adhere to them.

Trade-off. Both the change history and growth history view are deduced from quantitative data on the development process. To contrast this with the resulting quality of the tests, we introduce a view incorporating test coverage.

2.3 Test Quality Evolution View

Goal. Test coverage is often seen as an indicator of "test quality" [24]. To judge the long-term *"test health"* of a software project, we draw the test coverage of the subject system in function of the fraction of test code *tLOCRatio* ($tLOCRatio = tLOC/tLOC + pLOC$) and in function of time.

Description. In this view:

- We use an XY-chart representing *tLOCRatio* on the X-axis and the overall test coverage percentage on the Y-axis. Individual dots represent releases over time.
- We plot four coverage measures (distinguished by the color of the dots): class, method, statement and block[2] coverage.

Interpretation. Constant or growing levels of coverage over time indicate good testing health, as such a trend indicates that the testing-process is under control. The fraction of test code, however, is expected to remain constant or increase slowly alongside coverage increases. Severe fluctuations or downward spirals in either measure implies weaker test health.

Technicalities. For now we only compute the test coverage for the major and minor releases of a software system. We do not compute coverage for every commit as: (i) we are specifically interested in long-term trends in contrast to fluctuations between releases due to the development process; (ii) computing test coverage (for a single release) is time-consuming; and (iii) automating this step for all releases proved difficult, due to changing build systems and

[2]A basic block is a sequence of bytecode instructions without any jumps or jump targets, also see http://emma.sourceforge.net/faq.html (accessed April 13, 2007)

(varying) external dependencies that were not always available in the VCS.

3 Experimental setup

To evaluate the value of the three test co-evolution views we proposed, we apply the approach to two open source software projects. First, we generate the three views for each project and use them to summarize our interpretation of the project's history. Next, we apply both an internal and an external evaluation to validate our observations.

Tool. Our tool[3] is built around the *Subversion* VCS. With the help of the *cvs2svn*[4] script we can also address *CVS*. Using the *SVNKit* library[5], we are able to query Subversion directly from our Java-built toolchain that automatically generates the change history view (Section 2.1) and the growth history view (Section 2.2).

For the coverage history view, we used *Emma*[6], an open source test coverage measurement solution. We integrated Emma in the Ant build process of the case studies with the help of scripts and manual tweaking, as automating this process proved difficult.

Case studies. Our main criteria for selecting the case studies were: (i) the possibility of having a local copy of the project, for performance reasons, (ii) Java, as our current tool is targeted towards Java, and (iii) the availability of JUnit tests. Checkstyle and ArgoUML matched these criteria. When discussing them, note that not every type of visualization is shown for both cases because of space restrictions. However, all views can be seen in the online appendix[3].

Evaluation. In the *internal evaluation* we verify our findings using (i) log messages posted by developers during commits to the versioning system and (ii) code inspections. We split up our interpretation into individual statements that we try to counter with the logs. A successful counter then validates our claim. For the *external evaluation*, we send a survey to lead developers of the considered projects, asking to (i) chronicle the system's (test) evolution; (ii) read about the proposed views and our corresponding interpretation; and (iii) to accept or reject our statements. With this structured survey we aim to prevent influencing developers with our findings and techniques before they were interviewed.

Finally, we ask them to give feedback about the usefulness and possible improvements. The survey that we have sent to the developers is outlined in Table 2.

4 Case 1: Checkstyle

Checkstyle[7] is a tool that checks whether Java code adheres to a certain coding standard. Six developers made 2260 commits in the interval between June 2001 and March 2007, resulting in 738 classes and 47 kSLOC.

Questions on the developer's view of the project's test history.
• How would you summarize the test history of the project (which kind of tests, when to test)? • Within your project, do you have a policy regarding (codified) tests? Has this policy been modified over time? • When do developers commit? Is there a variation in commit style (in time, in size?) • Which testing tools do you use (testing framework, coverage measurements, mutation testing, lint-style code checkers)? When have such tools been introduced? • Is there an interplay between reported/fixed bugs and associated tests? e.g. do developers write a codified test to demonstrate the bug or is a test written afterwards to demonstrate that a bug has been fixed?
Questions on the evaluation of our interpretation.
• Which statements correspond with your experience based expectations? Which ones are new to you? Which ones are not true? • Which interesting events during the project's history did we miss?
Concluding questions.
• How could you as developer or team lead benefit from such visualizations? • Which additional aspects would you like to see in visualizations like these that try to summarize the project's history?

Table 2. Developer Survey.

4.1 Observations

Change history view. The change history view of Checkstyle (Figure 3)[8] results in the following observations with regard to the testing behavior of the developers. At the very beginning of the project up until commit #280, there is only one test (with file ID 20), which is changed very frequently (visible through the yellow horizontal bar). At that point, a number of new tests are introduced. From commit #440 onwards, a new testing strategy is followed, whereby the introduction of new production code (a red square) almost always entails the immediate addition of a new unit test (a green triangle). From #670 onwards, integration tests appear (visible by the yellow diamonds at the top of the chart). This commit is also interesting because it shows a vertical yellow bar, indicating that a large number of unit tests are modified, suggesting that several of the unit test files are affected by the adoption of integration tests. This pattern returns around commit #780. Furthermore, around #870 and #1375 test additions can be seen through the vertical bar of green triangles. Due to the massive number of unit tests involved this might indicate (i) a "phased testing approach", where an increased test effort is taking place at certain points in time (with little or no testing in between); or (ii) shallow changes to the test code (e.g., import-optimization).

Note that the number of units shown in this visualization is often higher than the number of classes present in

[3]See http://swerl.tudelft.nl/testhistory
[4]http://cvs2svn.tigris.org/
[5]http://svnkit.com/
[6]http://emma.sourceforge.net/
[7]http://checkstyle.sourceforge.net/

[8]Ideally, these visualizations should be seen in color. High-resolution color images are also available at http://swerl.tudelft.nl/testhistory

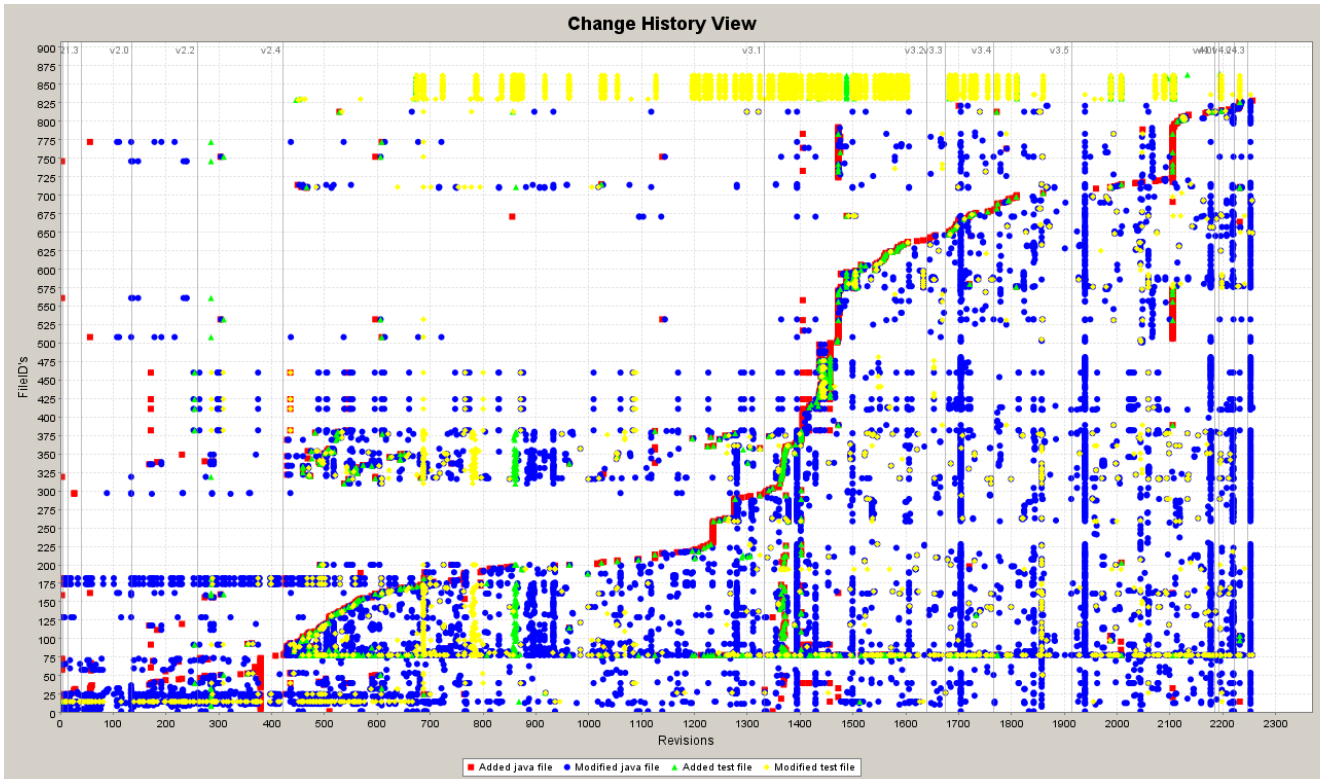


Figure 3. Checkstyle change history view.

the latest version of the software system. This is due to the fact that when a file gets deleted at a certain point in time, it remains present in the visualization. In this context, we also want to note the presence of *"outliers"* in the visualization, dots that lie above the growing curve of classes that are added. These outliers are caused by successive *move* operations in the subversion repository, but remain associated with their original introduction date.

Creating Figure 3 also provided us with the following statistics of Checkstyle's evolution: in total 826 classes were added to the system over time, of which 363 have an associated unit test. We also counted 36 integration tests.

Growth history view. From the change history view we learned that Checkstyle's classes and test classes are usually changed together, apart from a series of edit sequences to the test files specifically. What cannot be seen from the change history view, is how much of the code was affected by the actual changes made. For that purpose, the growth history view can be used.

The growth history view for Checkstyle, shown in Figure 4, displays curves that grow together, indicating a synchronous co-evolution. In general, increases as well as decreases in the number of files and code in production are immediately reflected in the tests. Complementing this observation with the change history view, the most likely explanation of the phases is the occurrence of many concurrent, shallow changes.

In particular, the figure confirms the initial single test code file that gradually grows and extensively gets reinforced after release 2.2 (during a phase of pure testing; see annotation 1). Another period of test reinforcement happens before release 3.0 (ann. 2): the amount of test code increases while the number of test cases barely changes. In the period from release 2.2 until beyond 2.4, development and testing happen synchronously, with an additional effort to distribute test code over multiple classes. This development approach is maintained until approximately halfway between release 3.1 and 3.2, where a development-intensive period results in a testing time backlog (ann. 3). Shortly after that there is some additional test effort (increases in test code, test cases as well as test commands). Thereafter, testing happens more phased until 3.5 (ann. 4). In the last period, the co-evolution is again synchronous, with a gradually decreasing time delay towards the last considered version. In the figure, we also observe test refactorings (see ann. 5 and Table 1).

Test Quality Evolution View. The test quality evolution view in Figure 5 shows a generally relatively high level of test coverage, with class coverage around 80%, climbing towards 95% in the later versions of the software. For the other levels of coverage, a similar steady increase can be seen. Throughout the evolution, the fraction of test code grows as well. This makes us assume that test coverage is

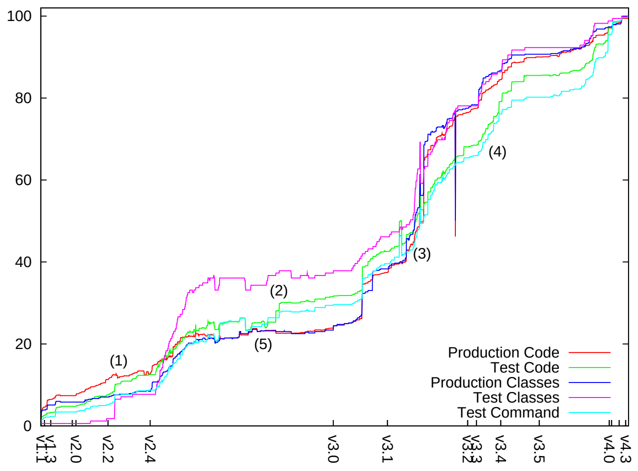


Figure 4. Checkstyle growth history view.

considered an attention point that is monitored carefully.

Two other observations stand out. First, release 2.2 has an interesting phenomenon: a sudden sharp decline for class, method and statement coverage, with a mild drop of block coverage. Secondly, there is a decline in coverage (at all levels) between release 2.4 and 3.0. The version numbers suggest that the system has undergone major changes.

4.2 Internal evaluation

To evaluate these observations, we first contrasted them with log messages at key points.

"Up until #280 there is a single unit test". The single test with file ID 20 is called `CheckerTest`. Inspection of this file pointed out that this actually was not a typical unit test, but rather a system test [3]. `CheckerTest` receives a number of input files and checks the output of the tool against the expected output.

"Testing has been neglected before the release 2.2". Inspection reveals that this coverage drop is due to the introduction of a large number (39) of anonymous classes, that are not tested. These anonymous classes are relatively simple and only introduce a limited number of blocks per class,

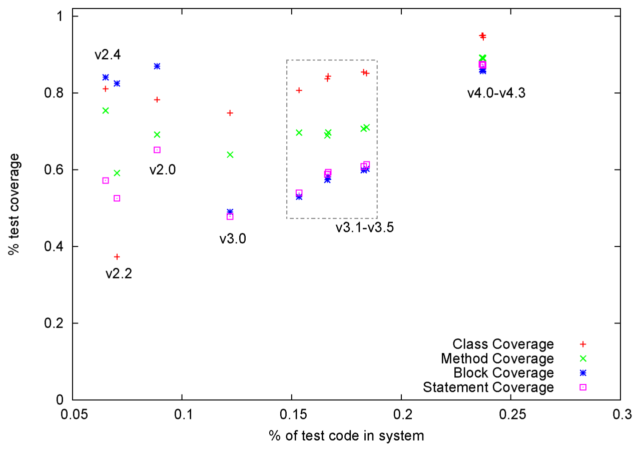


Figure 5. Checkstyle Test Quality Evolution

and therefore, their introduction has a limited effect on the block coverage level. Class coverage however, is more affected because the number of classes (29) has more than doubled with the 39 additional anonymous classes. In-depth inspection taught us that the methods called by the anonymous classes *are* tested separately. In the next version, all coverage levels increase because of the removal of most of the anonymous classes. The drop is thus due to irregularities in the coverage measurement, falsifying the statement.

"There is a period of pure testing right after release 2.2 and before 3.0". We sought for evidence that tests are neglected during this period, but instead we encountered logs for 2.2 such as *Added [6 tests] to improve code coverage* (#285), *updating/improving* the coverage of tests (#286 and #308) and even *Added test that gets 100% code coverage* (#309). The assumption of a test reinforcement period before 3.0 is backed up by several messages between #700 and #725 mentioning *improving test coverage* and adding or updating tests.

"From version v2.2 until beyond v2.4, synchronous co-evolution happens". To counter this, we looked for signs that pure development was happening, e.g., by new features being added. Investigation of the log messages around that time however showed that it concerns a period of bug fixing or patching (#354,#356,#357,#369,#370,#371,#415) and refactoring (#373,#374,#379,#397,#398,#412). Moreover, during this period production classes and test cases were committed together.

"Halfway between release 3.1 and 3.2 is a period of pure development". For this period, we could not find back the habit of committing corresponding test cases alongside production classes. Rather, a couple of large commits consisting of batches of production files occur, with log messages reporting the addition of certain functionality (#1410-#1420). Shortly after that, developers mention the addition of new tests (#143x and #1457).

"Between 3.4 and 3.5 testing happens more phased (ann. 4, Figure 4), followed by more synchronicity again". We could not really confirm this behavior nor distinguish both phases by means of the log messages, as we deduce that this period concerns mainly fixes of bugs, code style, spelling, build system and documentation.

"Around #670 and #780, developers were performing phased testing." The message of #687 mentions *"Upgrading to JUnit 3.8.1"*, which makes us conclude that it concerns shallow changes. The same accounts for the period around #780: test cases are (i) modified to use a new test helper function; and (ii) rearranged across packages.

4.3 External evaluation

Two Checkstyle developers completed the survey we sent, sharing their opinions about our observations. As an answer to questions about the system's evolution and test processs, they indicate that automated tests have always been valued very highly. The JUnit suite is integrated in the build system as a test target. Coverage measurements (with Emma) as well as code checks (using Checkstyle on itself) have been regularly performed since Checkstyle's origin. There is however no formal policy regarding their use.

The JUnit tests are implemented as I/O tests focused towards a specific module. Especially while changing Checkstyle's internal architecture — between versions two and three — the presence of the test suite was deemed invaluable. Regarding the synchronicity of development and test writing effort, one developer confirms that code and regression tests are typically committed at the same time. Even more, both developers indicate that they try to write a test that fails first before fixing the bug, making the test pass.

Currently, the code base is considered mature and stable. As a result, changes are smaller yet self-contained, i.e. contain all code, tests and documentation.

5 Case 2: ArgoUML

ArgoUML[9] is an open source UML modeling tool that includes support for all standard UML 1.4 diagrams. The first contributions to ArgoUML go back to the beginning of 1998, and up to December 2005, 7477 subversion commits were registered. The final release we considered for this study was built by 42 developers who wrote 1533 classes totaling 130 kSLOC.

5.1 Observations

Change history view. We observe that around commit 600 the first tests appear (figure not shown). The introduction of these first test cases does not coincide with the introduction of new production code, a trend that we witness throughout the project's history. Moreover, tests are typically also not changed together with their corresponding production classes. In addition, we observe periods of phased testing, e.g. the vertical bars around commits #2700 and #4900. Certain tests appear to change very frequently throughout ArgoUML's history, evidenced by horizontal yellow bars. The derived statistics count 4213 Java production classes, 402 of which have a associated test. In addition, there are 36 integration tests.

Growth history view. From the growth view in Figure 6, we deduce a slow introduction of test skeletons at around 0.10, followed by a more consistent use of codified tests from 0.12 on. These tests are added and extended periodically (in phases), confirming the change-observations in the change history view. We tag these as periods of pure testing, as most of the time these steps do not correspond with increases in production code (ann. 1). Besides these periods of testing, the test code is barely modified, except for some test skeleton introductions early on (between releases 0.10 and 0.12 (ann. 2) and periodic test refinements (ann. 3)

[9]http://argouml.tigris.org/

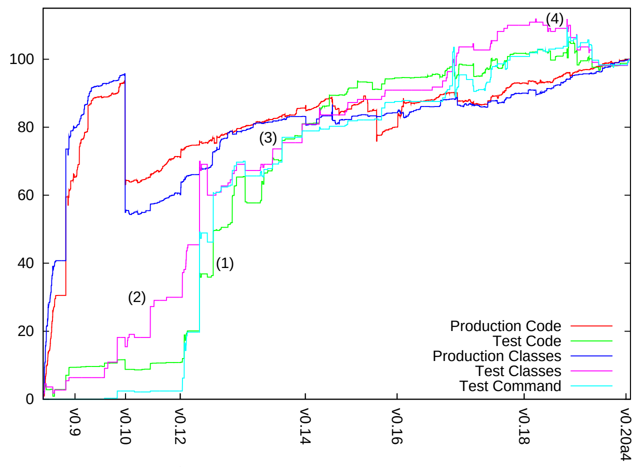


Figure 6. Growth history view of ArgoUML.

and refactorings (ann. 4). From 0.16 on, coding and testing happens in smaller increments, yet not synchronous as the curves are not moving in similar directions.

Note that the initial "hill" in the production code curve is due to architectural changes which are reflected in a changed layout in the versioning system, resulting in the source code residing in two locations at the same time. Later on, before release 0.10, the old layout structure and code-remains get deleted.

Test Quality Evolution View. Even without this side-effect, the initial test-writing effort is rather low and only slowly increasing. The fact that the first release of JUnit (beginning of 1998) more or less coincides with the start of the ArgoUML project might explain why the effort that went to test-writing was rather low in the earlier phases, as JUnit was not yet well known at that time. ArgoUML's view (not shown here due to space limitations) shows an increasing coverage as the test code fraction grows over time between v0.14 and v0.18 to 37% block coverage for 9% test code. The last considered version of ArgoUML, v0.20, is characterized by a sudden drop in test coverage.

5.2 Internal evaluation

We again first contrasted our observations with log messages from key points in the development history.

"The initial test-writing effort is rather low and only slowly increasing." We looked for test case additions in the early phases of the project, but could not find many. According to the change log, a first JUnit test has been introduced in September 2001 (without JUnit included in the repository). Follow up logs mention the introduction of *first version* (#781) and *simple* (#824) test cases, indicating the adoption of JUnit-style tests. Significant test reinforcements happen from release 0.12 on. Around commit #1750 the development branch 0.13 containing test cases is merged with the trunk. At that time, a test suite as well as build targets for testing are first introduced.

To counter the claim *"There are regular periods of* *phased testing"*, we search the log for commits where code and tests are changed together. This only happens during merges of branches to the trunk[10], where logs (e.g. #1991 and #2782) indicate that tests are reinforced before the commit (and where the actual development has been done before the merge). Other test commit logs confirm the phased nature of testing (e.g. #1796, #2166, #2411, #2811).

"From 0.16 on, coding and testing happens in smaller increments, yet not synchronous." We looked for log messages indicating synchronous co-evolution in the period #6100-#6800, yet we could only detect a few bug fixes with corresponding test case adaptations. Smaller coding commits happened in between test commits of limited size.

"Version v0.20 of ArgoUML is characterized by a sudden drop in test coverage." During the coverage measurement, we noticed that ArgoUML's `mdr` component, a storage backend, was extracted into a separate project. As a backend, this component was better tested than the remainder of the project, resulting in the coverage drop.

5.3 External evaluation

As a reaction to our inquiry, the ArgoUML project leader and a developer completed the survey. They indicate that codified testing within the project is done by developers in an informal way. Before a release, the policy requires the codified tests not to signal any problem. Furthermore, users are involved in ad hoc testing of the application during alpha and beta testing. Over the project's lifetime, many development tools have been adopted (and sometimes abandoned again). JUnit has been introduced in October 2002, JCoverage has been used as coverage tool until 2006. Test-driven development is not a habit.

The developers acknowledge the limited early testing as well as the phased testing approach, which they identify as periods where *the focus of different developers was periodically moving between testing and code*. However, these testing efforts were not coordinated. Addressing the lower coverage compared to Checkstyle, the project leader adds that ArgoUML being a desktop GUI application implies that most of the code is meant to control graphical components. To write, maintain and deploy test code for such systems is a larger effort than for batch-oriented applications.

6 Discussion

We now address the research questions that we have defined in Section 1.

RQ1 *Does co-evolution always happen synchronously or is it phased?* From the change history view, we deduce whether production code and test code are modified together. Specifically, we witnessed (i) green or yellow vertical bars indicating periods of pure testing (both case studies) and (ii) test dots on top of production dots as indicators for the simultaneous introduction and modification of production code with corresponding unit tests (e.g., Checkstyle).

To characterize this co-evolution, e.g., *are (minimal) changes necessary to obtain a running test suite?*, or rather, *are additional tests being written?*, we use the growth history view. In that view, we saw (i) curves following each other closely denoting synchronous activities (e.g., Checkstyle), while (ii) stepwise curves point to a more phased testing approach (e.g., ArgoUML).

RQ2 *Can an increased test-writing effort be witnessed right before a major release or other event in the project's lifetime?* In the case studies that we performed, we saw no evidence of a testing phase preceding a release. We attribute this to the nature of the chosen case studies. The developers of these open source projects contribute in their free time. There are no strict schedules nor formal policies in use. Checkstyle's developers apply a continuous testing effort alongside development. ArgoUML's development process does prescribe a user testing phase before a release. As this approach does not result in codified tests, it can as such not be observed in these views.

RQ3 *Can we detect testing strategies, e.g., test-driven development?* From a commit perspective, test-driven development is translated as a simultaneous commit of a source file alongside its unit test. We found indications of test-driven development in Checkstyle, by means of "test" dots on top of "code" dots in the change history view, signifying concurrent introduction as well as co-evolution.

RQ4 *Is there a relation between test-writing effort and test coverage?* For the two considered case studies we observed that test coverage grows alongside test code fraction, especially during periods of steady, incremental development. As such, we expect that the required test-writing effort for a system can be approximated given the desired test coverage. In future work, we want to quantify this relation and compare the correlation across projects. Moreover, we expect the following factors to have an influence:

- *Kind of tests* under consideration. We took the overall coverage level into account, without making a distinction between unit tests and more integration kind of tests. For the case studies considered here, we noticed that Checkstyle indeed has a set of integration tests. ArgoUML has, next to the unit test suite, a separate suite of automated GUI tests.
- The *quality focus* of the developers of the respective projects. In the change log messages of Checkstyle, developers mention the use of a coverage tool to detect opportunities for increases in test coverage. Compared to a system with a similar fraction of test code, we noticed a considerable yield in test coverage.

[10]In a VCS, the trunk is the main development line. Sometimes, new features, bug fixes or experimental changes are first tried in isolation in a separate development line that is afterwards merged back into the trunk.

- The *testability* of the software system under test. Bruntink and Van Deursen observed a relation between class level metrics (especially Fan Out, Lines Of Code per Class and Response For Class) and test level metrics [5]. This means that the design of the system under test has an influence on the test-writing effort required to reach a certain coverage criterion.

Consideration. When studying the co-evolution of production code and test code, we have experienced the three introduced views as complementary to judge the *test health* of a software system. The log message of both projects confirm the test evolution need: regularly, test code is adapted, cleaned and refactored. We found regular indications that changes in production code resulted in a non-compilable test suite (also see Moonen et al. [18]), effectively removing the safety net that a test suite is able to provide. Even if the test suite remains compilable after changing production code, the structure of the test suite is likely to degrade over time [18]. Also, the test's effectiveness might drop, even with constant levels of test coverage, when, e.g., newly introduced *boundary values* might not be tested.

Threats to validity. We identified the following threats to validity. In order to create the change history view (see Section 2.1) we use a simple heuristic that matches the classname of the unit of production code to the classname of the unit test, e.g., we matched `String.java` to `StringTest.java`. Distinguishing unit tests from integration tests purely based upon naming conventions might not be generizable. At one hand, unit tests are not required to contain the unit under test's name in the file name. At the other hand, a test source file with a similar name as the unit under test does not have to be a unit test. Moreover, there is a thin line between unit tests and integration tests. The Checkstyle developers see their tests more as I/O integration tests, yet associate individual test cases with a single production class by name. For each of the case studies that we have performed in this study, we found that the heuristical matching worked well, but we cannot guarantee that this will be the case for other software projects. As such, future work will be to perform a semantic analysis to make the test/production code correlation.

The individual *commit style* — short cycles, one commit per day, ... — of developers can slightly influence the resulting visualization, but, as we are mainly looking for general trends, we expect this effect to be minimal. The Checkstyle developers informed us about a change in commit style over time: as the project has become more mature, it has become a habit to make commits self-contained, i.e., all changes to code, tests and documentation are added in a single commit. During the experiments, we focused on the main development line — the trunk in the versioning tree — of the considered projects. Developers may branch from the trunk to try different development paths (used for fixing bugs in ArgoUML) that may be merged back into the trunk. This gives a similar result in the views as a large commit.

During the internal evaluation, we use the versioning system's log messages to confirm or reject our observations. As the use of such messages is unconstrained, we expect large differences in its use across projects, tasks and developers. The developer survey complements these log messages as additional source of validation.

We acknowledge the fact that more than two case studies are needed to draw more general conclusions.

7 Related work

We identified two research domains that are relevant in the context of this work: software visualization (targeting evolution) and research on traceability and co-changes.

Visualizing the revision history of a set of source code entities has been used to study how these entities co-evolve, e.g. Gîrba and Ducasse [10], Van Rysselberghe and Demeyer [22] and Wu et al. [23]. Typically, these visualizations use one axis to represent time, while the other represents the source code entities. This visualization-approach has been used to detect logical coupling between files, determine the stability of classes over time, etc. These approaches however, do not make a clear distinction between different types of source code entities, e.g., between production code and test code. The use of source code metrics to characterize the evolution of a system has for example been used by Godfrey and Tu to investigate whether open source software and commercial software have different growth rates [11]. To a certain degree, our research interests are similar as we investigate whether production code and test code grow at similar or different points in time during a project's history. Other research in the same area does not rely on visualization but still identifies logical coupling, e.g., Gall et al. [9] and Ball et al. [1].

In the domain of co-changes, Beyer and Hassan visualize software history by displaying sequences of cluster layouts based upon co-change graphs [2]. These graphs consist of files as nodes and the level of co-change as weighted edges. To identify co-changing lines, Zimmermann et al. [25] build an annotation graph based upon the identification of lines across several versions of a file. Kagdi et al. [13] apply sequential pattern mining to file commits in software repositories to discover traceability links between software artifacts. The frequent co-changing sets are subsequently used to predict changes in newer versions of the system. Hindle et al. [12] studied the release-time activities for a number of artifacts — source, test, build and documentation — of four open source systems by counting and comparing the number of revisions in the period before and after a release. The observed behavior is summarized in a condensed notation. Fluri et al. examine whether source code and associated comments are changed together alongside the evolutionary history of a software system [8]. This work is similar in its (technical) approach to ours, i.e. mining the versioning repository and refining file changes into categories to quantify changes and observe (lack of) co-evolution.

8 Conclusion & Future Work

In this paper we studied the co-evolution between production code and test code. In this context, we made the following contributions:

1. We introduced three views: (i) the change history; (ii) the growth history; and (iii) the test quality evolution view. We combined them to study how test code coevolves over time.
2. We demonstrated the use of these views on two case studies and distinguished more synchronous coevolution (Checkstyle) from a more phased testing approach (ArgoUML).

Moreover, over a project's history we can identify the introduction of tests, pinpoint periods of pure testing and pure development, test reinforcement as well as coverage increases. We did not observe testing phases right before a release. Indications of test-driven development were found, as numerous unit tests were introduced alongside their corresponding production code in Checkstyle. We noticed a variation in the fraction of test code needed to reach a certain level of test coverage between Checkstyle and ArgoUML.

As for future work, it is our aim to extend this research to industrial software projects, as the results might differ greatly in a context where imposed testing standards are in place. We also plan to package our toolchain as a monitoring tool, so that the quality of the testing process can be continually monitored. In another step we aim to gain deeper insight into factors that influence the relationship between the fraction of test code and the level of test coverage.

Acknowledgments This work has been sponsored by (i) the Eureka ∑ 2023 Programme; under grants of the ITEA project if04032 (*SERIOUS*), (ii) the NWO Jacquard *Reconstructor* project, and (iii) the Interuniversity Attraction Poles Programme - Belgian State – Belgian Science Policy, project *MoVES*.

References

[1] T. Ball, J.Kim, A.Porter, and H.Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proc. Int'l Workshop on Program Comprehension*, pages 259–268. IEEE, 2005.

[3] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 2000.

[4] F. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975.

[5] M. Bruntink and A. van Deursen. An empirical study into class testability. *Journal of Systems and Software*, 79(9):1219–1232, 2006.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[7] S. Elbaum, D. Gable, and G. Rothermel. The impact of software evolution on code coverage information. In *Proc. Int'l Conf. on Soft. Maint. (ICSM)*, pages 170–179. IEEE, 2001.

[8] B. Fluri, M. Würsch, and H. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*. IEEE, 2007. *To appear*.

[9] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. Int'l Conf. Soft. Maint. (ICSM)*, pages 190–197. IEEE, 1998.

[10] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal on Software Maintenance and Evolution: Research and Practice*, 18(3):207–236, 2006.

[11] M. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM)*, pages 131–142. IEEE, 2000.

[12] A. Hindle, M. Godfrey, and R. Holt. Release pattern discovery: A case study of database systems. In *Proc. of the Int'l Conf. on Softw. Maint. (ICSM)*, pages 285–294. IEEE, 2007.

[13] H. Kagdi, J. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Proc. Int'l Conf. on Program Comprehension (ICPC)*, pages 145–154. IEEE, 2007.

[14] D. Kung, J. Gao, and C.-H. Kung. *Testing Object-Oriented Software*. IEEE, 1998.

[15] M. Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[16] E. Maximilien and L. Williams. Assessing test-driven development at IBM. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 564–569. IEEE, 2003.

[17] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Proc. of the Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 13–22. IEEE, 2005.

[18] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. *Software Evolution*, chapter The interplay between software testing and software evolution. Springer, 2008. Editors: T. Mens, and S. Demeyer.

[19] P. Runeson. A survey of unit testing practices. *IEEE Software*, 25(4):22–29, July/August 2006.

[20] M.-A. Storey, D. Čubranić, and D. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proc. of the Symp. on Soft. Visualization*, pages 193–202. ACM, 2005.

[21] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, December 2007.

[22] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proc. Int'l Conf. Soft. Maint.*, pages 328–337. IEEE, 2004.

[23] J. Wu, R. C. Holt, and A. E. Hassan. Exploring software evolution using spectographs. In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, pages 80–89. IEEE, 2004.

[24] H. Zhu, P. A. Hall, and J. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

[25] T. Zimmermann, S. Kim, J. Whitehead, and A. Zeller. Mining version archives for co-changed lines. In *Proc. Int'l Workshop on Mining Soft. Repositories*, pages 72–75. ACM, 2006.

TUD-SERG-2008-003
ISSN 1872-5392