Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining

**Andy Zaidman · Bart Van Rompaey ·
Arie van Deursen · Serge Demeyer**

Abstract Many software production processes advocate rigorous development testing alongside functional code writing, which implies that both test code and production code should co-evolve. To gain insight in the nature of this co-evolution, this paper proposes three views (realized by a tool called TeMo) that combine information from a software project's versioning system, the size of the various artifacts and the test coverage reports. We validate these views against two open source and one industrial software project and evaluate our results both with the help of log messages, code inspections and the original developers of the software system. With these views we could recognize different co-evolution scenarios (i.e., synchronous and phased) and make relevant observations for both developers as well as test engineers.

A. Zaidman (✉) · A. van Deursen
Delft University of Technology, Mekelweg 4, 2626 CD Delft, The Netherlands
e-mail: a.e.zaidman@tudelft.nl

A. van Deursen
e-mail: arie.vandeursen@tudelft.nl

B. Van Rompaey · S. Demeyer
University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium

B. Van Rompaey
e-mail: bart.vanrompaey@gmail.com

S. Demeyer
e-mail: serge.demeyer@ua.ac.be

1 Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less useful (Lehman 1980). When evolving software, the source code is the main arte-fact typically considered, as this concept stands central when thinking of software. Software, however, is multi-dimensional, and so is the development process behind it. This multi-dimensionality lies in the fact that to develop high-quality source code, other artifacts are needed, e.g., requirements, documentation, tests, etc. (Mens et al. 2005). A software development process aiming for quality should thus allow these artifacts to *co-evolve* gracefully alongside their respective dimensions.

One artifact which is of primary importance when developing high-quality soft-ware, is the so-called *developer test* (i.e., a codified unit or integration test written by developers (Meszaros 2007)). Indeed, a 2002 report from the NIST indicates that catching defects early during (unit) testing lowers the total development cost sig-nificantly (Tassey 2002). Consequently, newly added functionality should be tested as soon as possible in the development process, to provide quick feedback to the de-velopers (Runeson 2006). Moreover, when a software system evolves (e.g., through refactoring), developers should run the persistent tests to verify whether the external behavior is preserved (Demeyer et al. 2002, p. 159). In this context, Moonen et al. have shown that even while refactorings are behavior preserving, they potentially invalidate tests (Moonen et al. 2008). In the same vein, Elbaum et al. concluded that even minor changes in production code can significantly affect test coverage (Elbaum et al. 2001).

This leads to the almost paradoxical situation whereby tests are important for the success of the software and its evolution, while they are also a serious burden during evolution, because they need to be maintained as well. Writing test-code that co-evolves gracefully alongside the production code is not an easy task, and software engineers need tools and methods that help to assess the nature of the co-evolution relationship.

In this paper we *retrospectively* explore the co-evolution of production and test code by mining a version control system (VCS) such as Subversion (SVN) or the Concurrent Versions System (CVS) (Kagdi et al. 2007). Our exploration of these version control systems is aided by the use of lightweight techniques and visualiza-tions, which, as Storey et al. observed, are common to the field of studying software evolution (Storey et al. 2005).

Our research is driven by the following research question: *Is it possible to establish the co-evolution process between developer tests and the corresponding production code by mining a version control system (VCS)?* In order to answer that question, we refine it into a number of subsidiary research questions:

RQ1 Does co-evolution between test and production code happen synchronously or is it phased?

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

RQ3 Can we detect testing strategies, such as test-driven development (Maximilien and Williams 2003)?

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

These questions are particularly relevant during quality assessments of the developer testing process. Certainly when such quality assessment is performed by external consultants, where a quick analysis using tools is a good way to get an insight into the testing process. For instance during "first-contact" situations (Demeyer et al. 2002, p. 39), in which you want to do an initial assessment of the software system in a short period of time, knowing how tightly the testing is interwoven with the normal development process (RQ1 and RQ2) is an indicator for the reliability and effectiveness of the developer tests. But they are equally relevant for the quality engineers inside the development team who need convincing arguments to persuade their colleagues to increase test activities. For instance, a quality engineer might monitor the testing activities, to verify whether procedures are followed (RQ3) or to detect trends that may hamper future testing or product quality (RQ2, RQ4) (Slaughter et al. 1998).

In order to answer these research questions we set up an experiment in which we study the co-evolution of production and test code of two open source software systems and one industrial software system. This experiment will allow us to validate the visualizations that we use and the results that we have obtained from them. We evaluate our findings (i) internally, by inspection of the code and the log messages that were written during development; and (ii) externally, by presenting our observations to the original developers and analyzing their remarks.

This paper extends our previous work (Zaidman et al. 2008) with a new industrial case study, clearly contrasting the open source testing strategies that we have witnessed in our two other case studies. Furthermore, we have significantly extended the discussion of our proposed techniques.

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 describing the implementation of our Test Monitor (TeMo) tool suite. Section 4 clarifies the experimental setup. Sections 5, 6 and 7 present our three case studies on respectively CheckStyle, ArgoUML and the industrial case. In Section 8 we discuss our findings and in Section 9 we identify threats to validity. Section 10 then relates our work to other work in the field and we finish with our conclusions and future work in Section 11.

2 Test Co-Evolution Views

As studying the history of software projects involves large amounts of data, visualization can help to deal with the resulting complexity and to understand aspects of either product or process (Ball and Eick 1996; Van Rysselberghe and Demeyer 2004). The alternate strategy which does not make use of visualization, would involve tedious manual analysis of log messages and source code, which, for real-world systems is next to impossible. In this work, we make use of visualizations to answer test co-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 production and test code by the developers over time.
2. The **Growth History View** that shows the relative growth of production code and test code over time.

3. The **Test Coverage Evolution View**, where we plot the test coverage of a system
    against the fraction of test code (versus the complete source code base) in a
    system at key points in a project's timeline.

The three views are inspired by related work (see Section 10), but have been
adapted to fit the specific needs for studying the co-evolution of production and
test code. As such, the Change History View is loosely based on work of Gîrba
and Ducasse (2006) and Van Rysselberghe and Demeyer (2004), while the Growth
History View has its roots in work of, e.g., Godfrey and Tu (2000).

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.
–   The files are sorted according to time of introduction into the VCS.
–   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 (black square) and modified
    production code (blue dot); newly added (red triangle) and modified tests
    (yellow diamond).[1,2]
–   Vertical lines represent release points of the system.

We create the plot by default for changes in the the main development line. Note
that developers can create branches in the VCS, i.e., development lines parallel
to the main development line typically used for experimental development, rapid
prototyping, bug fixing, etc.

Interpretation   Consider the example view in Fig. 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 red
triangles plotted on top of black squares. Test files that are changed alongside
production code show as yellow diamonds on top of blue dots. Vertical red (commit
1 in Fig. 1) or yellow (commit 9 in Fig. 1) *bars* indicate many changes to test code
files, i.e., more than would be expected during an iterative development process.
Horizontal bars in the Change History View on the other hand stand for frequently

---

[1]Please note that symbols that we use—square, dot, triangle, ...—are not all the same size. When
drawing this in the right order, we prevent overplotting.

[2]We have chosen these colors as to make sure that people with red-green color blindness would not
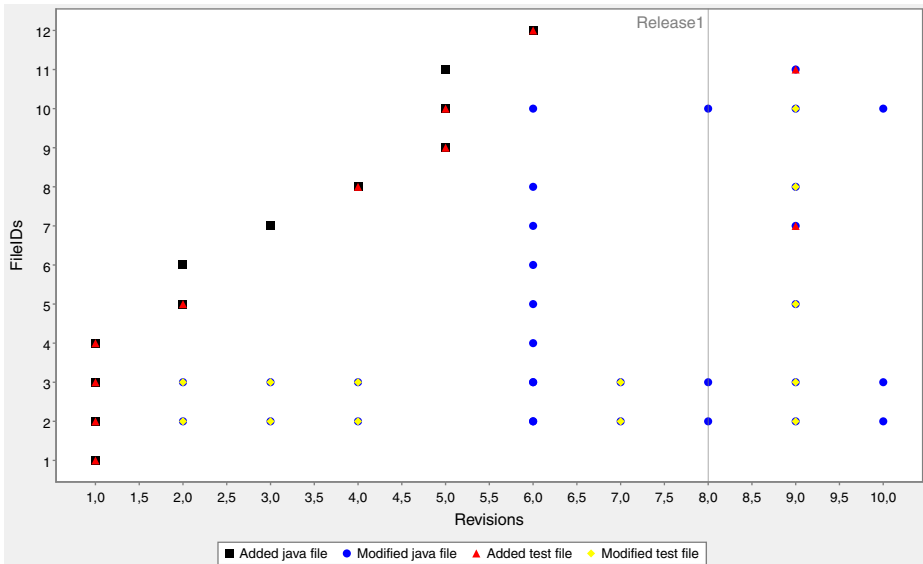be hampered (Prevalence and incidence of color blindness 2010).

Fig. 1 Example Change History View

changed files. Other patterns not specifically involving the tests, e.g., vertical or horizontal blue bars, have been studied by others (Van Rysselberghe and Demeyer 2004; Gall et al. 1998).

Limitations   The Change History View is mainly aimed at investigating development behavior, 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 activity 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 X-axis shows the time and is annotated with release points at the bottom and time stamps at the top. The Y-axis shows metrics presented as a relative 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 five metrics that we take into consideration are: Lines of production code
   (pLOC), Lines of test code (tLOC), Number of production classes (pClasses),
   Number of test classes (tClasses) and Number of test commands (tCommands—
   a test command is a container for a single test (Van Rompaey et al. 2007); in
   terms of xUnit these are the methods that start with `test` (Meszaros 2007)).

We apply this view to the main development line as well.

Interpretation First of all, we can observe phases of relatively weaker—e.g., the
perfectioning of existing functionality—or stronger—e.g., the addition of new
functionality—growth throughout a system's history. Typically, in iterative software
development new functionality is added during a certain period after a major re-
lease, after which a "feature freeze" (Hindle et al. 2007a) prevents new functionality
from being added. At that point, bugs get fixed, testing activity 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 production and test code is written synchronously when the
two curves are similar in shape (see Fig. 2a). A horizontal translation indicates a time
delay between one activity and a related one (Fig. 2b), whereas a vertical translation
signifies that a historical testing or development backlog has been accumulated over
time (Fig. 2c). Such a situation occurs, e.g., when the test-writing is delayed com-
pared to writing production code for many subsequent releases. In the last version
considered in the view, both activities reach the 100% mark, which is because we are
measuring relative activity for both test and production code.

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 "intense devel-
opment" phase. The sixth line represents the introduction of test case skeletons, i.e.,
empty test classes that serve as placeholder for the eventual test cases. Of importance
to note in the context of Table 1 is that a "→" indicates a strict requirement of staying
constant, while an empty cell does not indicate a strict requirement, i.e., the value
may change.

Limitations Both the Change History and Growth History View are deduced from
quantitative data on the development process, hence do not incorporate any infor-
mation about the quality of the test. Therefore, we introduce a view incorporating
test coverage as an indicator for the test quality.

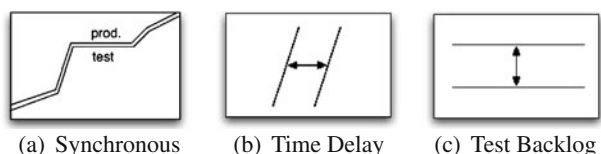**Fig. 2** Example patterns of
synchronous co-evolution

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

Table 1 Co-evolution scenarios

| Scenario	pLOC	tLOC	pClasses	tClasses	tCommands	Interpretation
1	↗	→				intense development
2	→	↗				intense testing
3	↗	↗				co-evolution
4	→	↗	→	→		test refinement
5	→	→	↗	↗		skeleton co-evolution
6		→		↗		test case skeletons
7		→			↗	test command skeletons
8	→	↘				test refactoring

2.3 Test Coverage Evolution View

Goal   Test coverage is widely used as an indicator of under-tested software compo-
nents and is often seen as an indicator of "test quality" (Zhu et al. 1997). While we
are not claiming that a high level of test coverage implies good test quality (Glover
2006; Berner et al. 2007), because of, e.g, the fact that doing good boundary testing is
not reflected in coverage measures, we are interested in observing how test coverage
relates to other test-related metrics. In particular, we are interested in observing
(i) how much test code is actually being written for reaching a particular level of
test coverage and (ii) whether fluctuations in the test code base in terms of lines of
code are reflected in the test coverage levels of the system. Such fluctuations might
indicate a (future) maintenance problem, e.g., if we signal dropping levels of test
coverage.

Description   In this view:

– We use an XY-chart representing $tRatio = tLOC/(tLOC + pLOC)$ 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 shape and color of the dots):
  class, method, statement and branch/block 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. As such, we expect that new dots appear to the upper
right area in Fig. 3 as new releases are created. Test reinforcements in subsequent
releases should show as dots that shift more over the vertical axis than over the
horizontal axis. Efforts to reduce the size of a test suite, e.g., to improve regression
test time and maintenance cost due to duplicated tests (Bible et al. 2001) should
result in a shift to the left on the horizontal axis. Hopefully, scenarios where new dots
appear to the down right do not appear, as this would indicate that additional tests
for new development are adding to the *tRatio*, without at least maintaining the level
of test coverage. Severe fluctuations between consecutive releases imply weaker test
health, i.e., a degrading quality of the test suite in the long term.

Fig. 3 Example patterns in the Test Coverage Evolution View
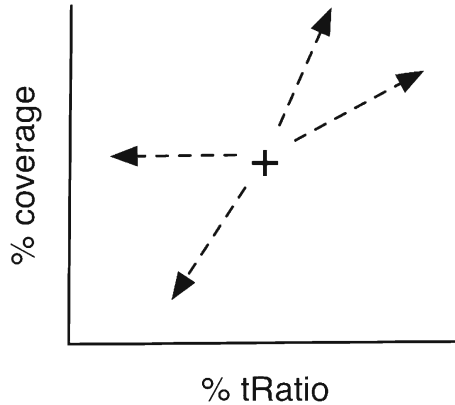
Limitations   For now we only compute the test coverage for the major and minor releases of a software system, as computing test coverage (for a single release) is quite time-consuming and difficult to automate. Since the Test Coverage Evolution View is mainly aimed towards detecting long-term trends in test-coverage, such an approach is typically good enough.

3 The TeMo toolchain

In order to instantiate the views that we have described in the previous section, we have built the *Test Monitor (TeMo)* tool suite. This Java-built tool-suite is available for download.[3] In this section we expand on the technologies that we have used for this set of tools, the level of automation and the limitations of TeMo.

Change History View   In order to create a Change History View (Section 2.1) we use the *SVNKit* library (SVNKit 2010) to extract the entire history of changes and log messages from a project's history. We store this information in XML, after which we are able to query it with XQuery (2010). Through querying we relate production and test code to each other: 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"*). Test classes that cannot be correlated in this way are considered to be integration tests and are placed on the top lines of the view. We render the view with JFreeChart (2010).

Note that the number of units shown in this visualization is often higher than the number of classes present in the latest version of the software system. This is due to when a file gets deleted at a certain point in time, it remains present in the visualization. Similarly, a renamed file shows up as a new line at the top of the chart at the time of its introduction, yet the original file line remains (and is not changed anymore). The same goes for a file that is moved within the repository. Moreover, a test is associated with both instances in such a case.

---

[3]http://swerl.tudelft.nl/testhistory

Growth History View   The Growth History View (Section 2.2) is created with the use of the *SVNKit* library. For each version of the software project that we consider in our analysis, we check out the entire 'working copy' of the project, after which we apply the *Lines of Code Counter* (LOCC 2010) tool to determine the number and size of artifacts. We render the resulting metric data into a chart using gnuplot (2010).

Our current implementation of TeMo is targeted towards a combination of JUnit/Java. More specifically, in order to separate production classes from test classes we use regular expressions to detect JUnit 3.x 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 (Van Rompaey et al. 2007), we search for a combination of `org.junit.*` imports and `setUp()` methods. Counting the number of test commands is done on the basis of naming conventions. More specifically, when we find a class to be a test case, we look for methods that start with `test`. We are aware that with the introduction of JUnit 4.x, this naming convention is no longer necessary, but the projects we consider in our case studies still adhere to them. In JUnit 4.x, a similar approach to identify the test cases would consist of searching for the `@Test` annotation.

Test Coverage Evolution View   Generating the Test Coverage Evolution View is mainly a manual effort, due to the challenges involved in computing coverage for historical software releases. These challenges include:

–   Integration of a coverage tool into the build system proved difficult to automate (i) due to varying build systems across projects and (ii) due to changing build configurations over time within the same project.
–   Building a historical release sometimes proved difficult due to missing external dependencies in the VCS.

We limit ourselves to computing the test coverage for the major and minor releases of a software system, as computing test coverage for a single revision can be time-consuming. This supports our interest of studying long-term trends in contrast to fluctuations between releases due to the development process.

For generating the view, we use Emma (2010), an open source test coverage measurement solution. We have integrated Emma in the Ant (2010) build process of the open source case studies with the help of scripts and manual tweaking, as completely automating this process proved difficult. Once we have integrated *Emma* into the build process of a particular version of one of our open source case studies, we are able to generate the coverage report by running the already present unit tests.

For the industrial case study we made use of the fact that the company is already using the *Clover* test coverage tool (Clover 2010). As Clover is integrated into the *Maven* build system (Maven 2010) we did not have any problems with missing external dependencies. Obtaining the coverage reports from the industrial case study thus proved relatively easy.

Note that Emma and Clover do not always calculate the same coverage levels, as Emma determines block coverage (Mathur 2008), while Clover computes branch coverage (Binder 2000).

4 Experimental Setup

To evaluate the value of the three aforementioned test co-evolution views we use an explorative case study with multiple cases (Runeson and Höst 2009; Yin 2002). That is, we apply our three views to three realistic cases (two open source software projects and one industrial system) which result in a number of observations about the project's history. To validate whether these observations correspond with actual events, we cross-validate from an internal (inspection of VCS log messages and source code) and external perspective (feedback from the original developers). Below we give a detailed account and motivation for the design of the case study.

Selection of Cases   Our main prerequisites when selecting the cases were: (i) written in Java, as our tool is currently targeted towards Java, (ii) the availability of JUnit tests, (iii) the presence of the project history in a version control system (SVN or CVS). The open source projects Checkstyle and ArgoUML matched these prerequisites. Furthermore, the Software Improvement Group (SIG)[4] provided us with a third (industrial) case, which adheres to these prerequisites as well.

These cases are representative for software systems produced in smaller teams (5 to 10 core developers) where team members take up varying roles (analyst-designer, programmer, tester, debugger) and where quality assurance is mostly integrated into the daily activities of the team members. In such a context, our lightweight visualizations are easily adoptable into the normal practices of the team. Nevertheless, interviews with the developers involved reveals that these cases showed quite different test cultures, and hence represent relevant samples in the universe of small team software development.

Units of Analysis   Given the research questions, the unit of analysis for each case correspond to events in the project's history that are representative of co-evolution between developer tests and the corresponding source code. Consequently, for each of the cases, we generated the three visualisations and made a number of observations about relevant events (see Sections 5.1, 6.1 and 7.1). To confirm or reject these observations, we used a combination of internal and external perspectives.

Internal evaluation   Since the observations are our interpretation of the project's history, we need to evaluate whether they correspond with actual events, hence we first inspect the log messages that were written during development, and when we needed more details, we also checked the corresponding source code. To avoid bias during this inspection, we first look for evidence that contradicts our interpretation and we reject the observation if we found any such evidence. Next, we try to find evidence that confirms our interpretation and if we do, we accept the observation. If we do not find contradicting or supporting evidence, we classify the observation as "undecided". Again, to avoid any bias, all pieces of evidence were discussed among the first two authors of the paper.

External evaluation   To complement the internal evaluation we also verify our findings with team members of the organization producing the system under study.

---

[4]Software Improvement Group (SIG), Amsterdam, The Netherlands (http://www.sig.nl).

Table 2 Developer survey

Part 1: 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?
Part 2: Questions on the evaluation of our observations.
  – Which observations 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?
Part 3: 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?

The questionnaire that we use features open-ended questions, aimed at gathering (i)
qualitative data that supports or rejects our observations and (ii) anecdotal evidence
on the usefulness of our visualizations.

We sent a survey to lead developers of the considered projects as outlined in
Table 2. First, we ask them to chronicle the system's (test) evolution (Part 1); af-
terwards, we encourage them to read about the proposed views and to accept or
reject the corresponding observations (Part 2). Finally, we ask them to give feedback
about the usefulness and possible improvements (Part 3).

5 Case 1: Checkstyle

Checkstyle[5] is a tool that checks whether Java code adheres to a certain coding
standard. Six developers made 2,260 commits in the interval between June 2001 and
March 2007, resulting in 738 classes and 47 kSLOC (kSLOC = thousand source lines
of code).

5.1 Observations

Change History View   The Change History View of Checkstyle (Fig. 4)[6] 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 (which comes after release

---

[5]http://checkstyle.sourceforge.net/

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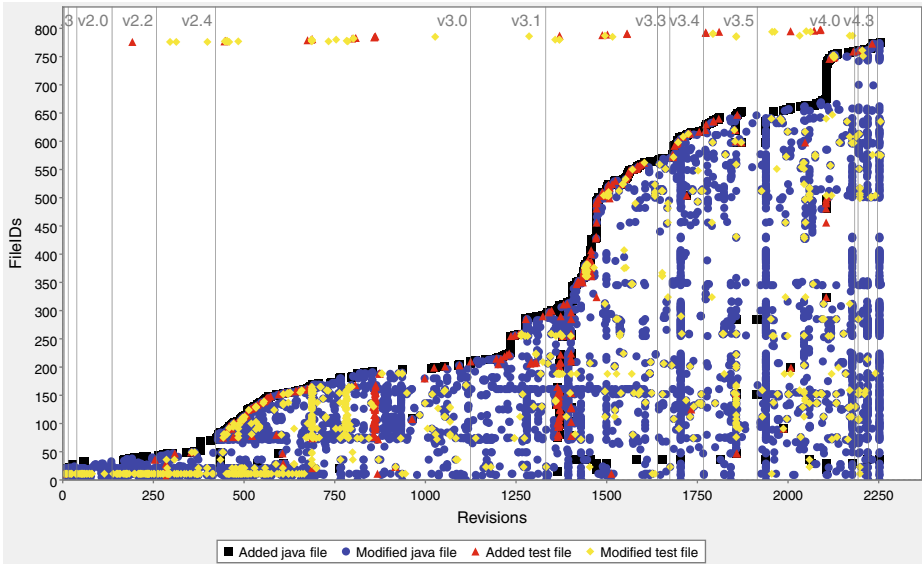at http://swerl.tudelft.nl/testhistory.

Fig. 4 Checkstyle Change History View

2.2), there is only one test (with file ID 11), which is changed very frequently (visible through the yellow horizontal bar). This is our observation *Checkstyle.O.1*. At that point, a number of new tests are introduced. From commit #440 (comes after release 2.4) onwards, a new testing strategy is followed, whereby the introduction of new production code (a black square) almost always entails the immediate addition of a new unit test (a red triangle). While the first integration test appears around #220, more unit tests only start to appear from #670 onwards, integration tests appear (visible by red triangles and 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 (*Checkstyle.O.2*), suggesting that several of the unit test files are affected by the adoption of integration tests. This pattern of phased testing returns around commit #780 (*Checkstyle.O.2*). Furthermore, around #870 and #1375 test additions can be seen through the vertical bar of red triangles. Due to the several tens of unit tests involved this might indicate (i) a "phased testing approach", where an increased test activity 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 statement optimization).

Creating Fig. 4 also provided us with the following statistics of Checkstyle's evolution: in total 776 classes were added to the system over time, of which 181 have an associated unit test. We also counted 23 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 Fig. 5, displays curves that show the same trends most of the time. This indicates a relatively 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. During certain time periods however, development or testing activities take the upper hand.

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 intense testing; see *Checkstyle.O.3* – annotation 1 in Fig. 5). Another period of test re-inforcement happens before release 3.0 (*Checkstyle.O.3*): the amount of test code increases while the number of test cases barely changes (ann. 2). In the period from release 2.2 until beyond 2.4, development and testing happen synchronously (*Check-style.O.4*), 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 (*Checkstyle.O.5* – ann. 3). Shortly after that there is some additional test activity (increases in test code, test classes as well as test commands). Thereafter, testing happens more phased until 3.5 (step-wise increases; *Checkstyle.O.6* – ann. 4). In the last period, the co-evolution is again synchronous, with a gradually decreasing time delay towards the last considered version.
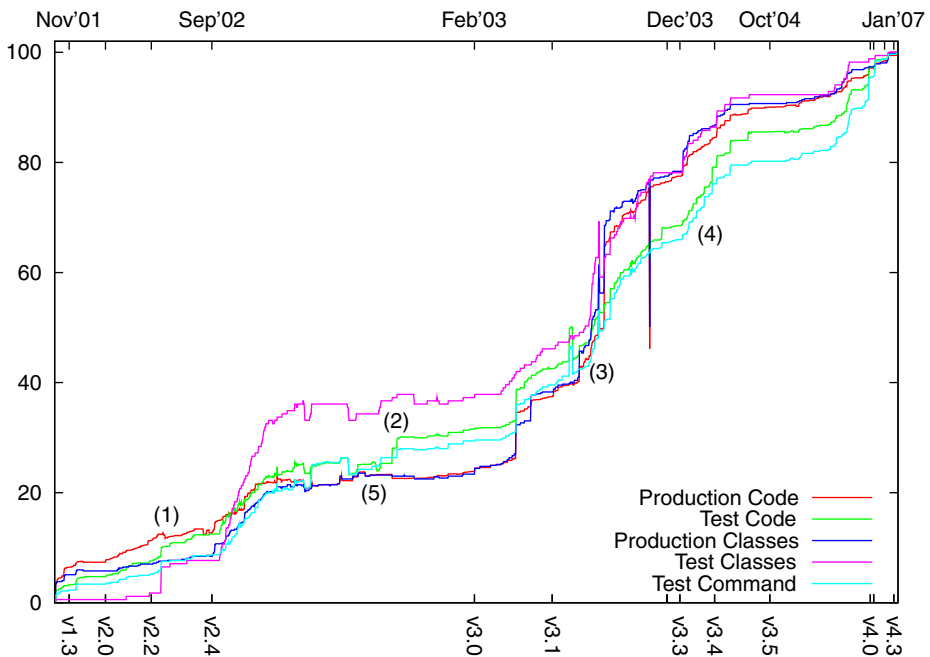

Fig. 5 Checkstyle Growth History View

When relating our observations to the Change History View, we see how certain vertical bars (see Fig. 4) representing work on the tests do not result in growth of artifacts (#780), while others do impact system size (#870 and #1,375). The former category represents more shallow changes such as refactoring or code beautification; the latter category concerns periods of reinforcement of tests.

Test Coverage Evolution View   The test coverage evolution view in Fig. 6 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. The measurements for the different levels of coverage grow closer to each other over time, suggesting that additional activity went to increasing the coverage at finer levels of granularity (e.g., block and statement coverage). This makes us assume that test coverage is 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 in class and method coverage, with a mild drop of block coverage (*Checkstyle.O.7*). Secondly, there is a decline in coverage (at all levels) between release 2.4 and 3.0, yet the *tRatio* increases by more than 10%. This indicates that new test code being written in this period does not maintain the previous level of test coverage. The shift in release number may indicate that a major restructuring has been applied to the system.
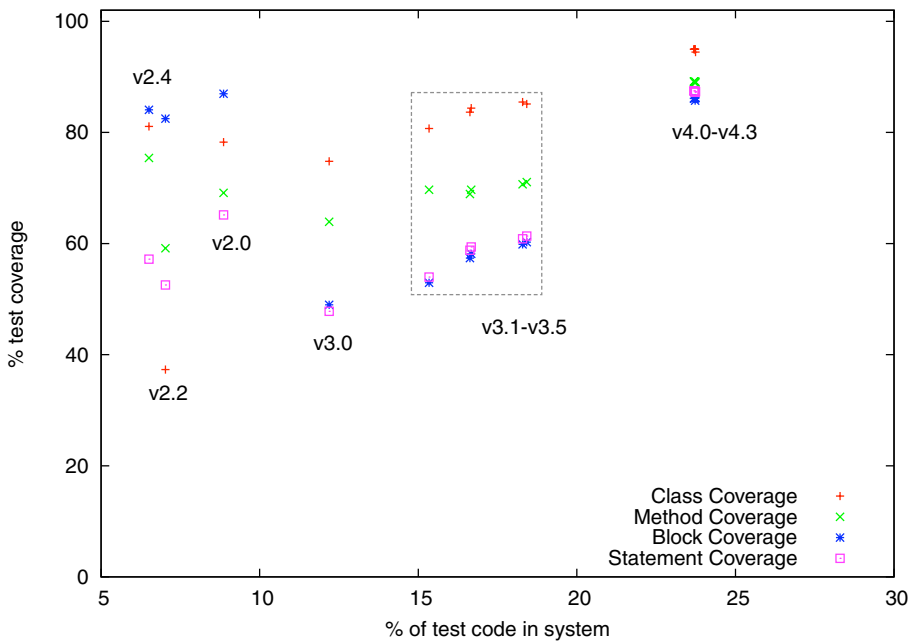

Fig. 6 Checkstyle Test Coverage Evolution View

5.2 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" (*Checkstyle.O.1*). The single test with file ID 11 is called `CheckerTest`, which provided us with a first clue. Subsequently, we used regular expressions to search through the source code to find test-related terminology (e.g., the phrases "test" and "import junit.". We found that `CheckerTest` was indeed the only test, hence we accepted the observation. However, we must point out that this actually was not a typical unit test, but rather a system test (Binder 2000). Indeed, `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" (*Checkstyle.O.7*). Inspection reveals that this coverage drop is due to the introduction of a large number (39) of anonymous classes, that are not tested. These new classes are relatively simple and only introduce a limited number of blocks per class. 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. In-depth inspection teaches 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, rejecting the observation.

"There is a period of intense testing right after release 2.2 and before 3.0" (*Checkstyle.O.3*). In order to try and reject this observation, we first sought for evidence that tests are neglected during this period, but instead we encountered log messages around revision 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. Therefore we accepted the observation.

"From version v2.2 until beyond v2.4, synchronous co-evolution happens" (*Checkstyle.O.4*). Again we tried to reject this observation by looking for signs that intense development was happening, e.g., through 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. Hence, we accept the observation.

"Around #670 and #780, developers were performing phased testing." (*Checkstyle.O.2*) The message of #687 mentions *"Upgrading to JUnit 3.8.1"*, which makes us conclude that it concerns shallow changes. For the period around #780, the test cases are (i) modified to use a new test helper function; and (ii) rearranged across packages. As such, these changes concern the test design, but are not really test reinforcements. As such, we reject the observation.

"Halfway between release 3.1 and 3.2 is a period of intense development" (*Checkstyle.O.5*). For this period, we could not find any traces of new test cases for the newly created 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). Thus we accept the observation.

"Between 3.4 and 3.5 testing happens more phased (ann. 4, Fig. 5*), followed by more synchronicity again"* (*Checkstyle.O.6*). We could not really accept nor reject both phases by means of the log messages or code inspections. We do notice that this period mainly concerns fixes of bugs, code style, spelling, build system and documentation.

5.3 External Evaluation

Two Checkstyle developers have completed the survey from Table 2, sharing their opinions about our observations. As an answer to questions about the system's evolution and test process, 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 regularly been 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 2.4 and 3.0—the presence of the test suite was deemed invaluable. Regarding the synchronicity of development and test writing activity, 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 failing test documenting the bug, before fixing it and 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 for a unit of change.

6 Case 2: ArgoUML

ArgoUML[7] 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, 7,477 subversion commits were registered. The final release we considered for this study was built by 42 developers who wrote 1,533 classes totaling 130 kSLOC.

6.1 Observations

Change History View We observe that around commit 600 the first tests appear (Fig. 7; *ArgoUML.O.1*). 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 (*ArgoUML.O.2*), e.g., the vertical red and yellow bars around commits #2,700 and
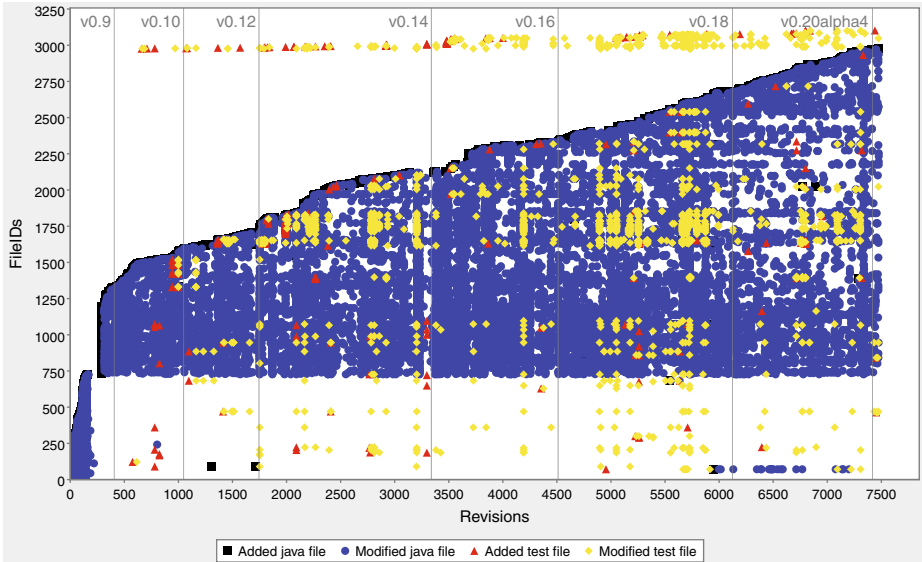
---

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

Fig. 7 ArgoUML Change History View

#4,900. Certain tests appear to change frequently throughout ArgoUML's history, evidenced by horizontal yellow bars.

Of interest to note in ArgoUML's development history is the fact that a white band exists at the bottom of the visualization, corresponding to fileIDs 0 to 725 (*ArgoUML.0.3*). The vertical black addition line at commit #289 seems to indicate that the VCS structure has been modified.

The derived statistics count 2,976 Java production classes, 126 of which have a associated test. In addition, there are 127 integration tests.

Growth History View   In the Growth History View in Fig. 8 we see that many test classes are introduced around v0.10, which still contain relatively little code, suggesting the introduction of test skeletons (*ArgoUML.O.1*). Next, the developers follow a more consistent use of codified tests from v0.12 onward: tests are added and extended periodically (in phases, see the stepwise growth: ann. 1), confirming the change-observations in the Change History View. We tag these as periods of intense testing, as most of the time these steps do not correspond with increases in production code (ann. 1; *ArgoUML.O.2*). Besides these periods of testing, the test code is barely modified, except for some test skeleton introductions early on (between v0.10 and v0.12—ann. 2) and periodic test refinements (ann. 3) and test refactorings (ann. 4). From v0.16 on, coding and testing happens in smaller increments, yet not synchronous as the curves are not moving in similar directions (*ArgoUML.O.4*).

Note that the initial "hill" in the production code curve is probably 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 (*ArgoUML.O.3*).
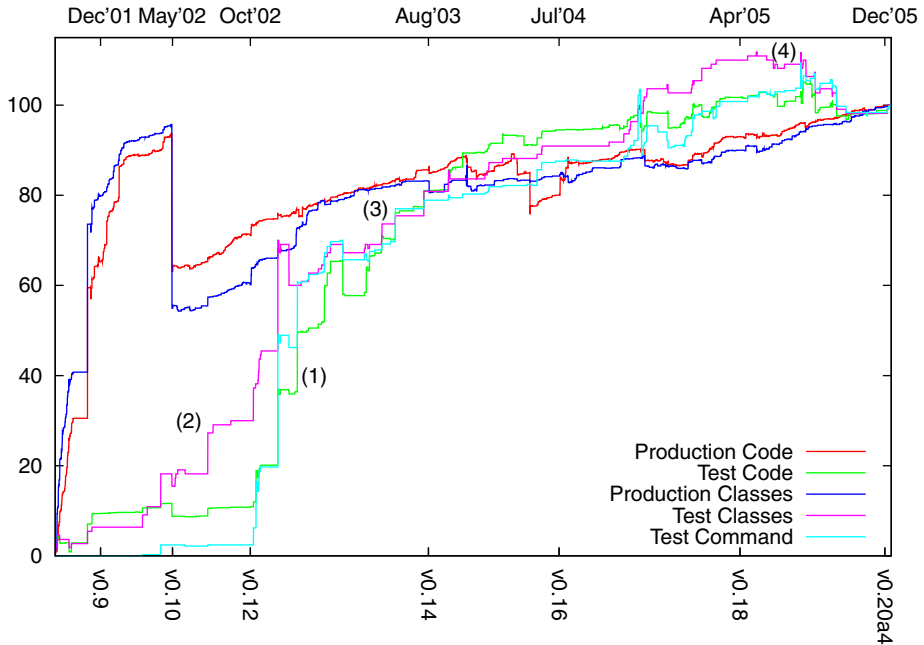
Fig. 8 Growth History View of ArgoUML

Test Coverage Evolution View   Even without this side-effect, we see in Fig. 9 that the
initial test-writing activity is rather low (*ArgoUML.O.1*) and only slowly increasing—
mind that we use a different scale compared to the previous case study. ArgoUML's
view shows an increasing coverage as the test code fraction grows over time between
v0.10 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 (*Argo
UML.O.5*).

6.2 Internal Evaluation

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

   "The initial test-writing activity is rather low and only slowly increasing." (*Ar-
goUML.O.1*) We looked for test case additions in the early phases of the project, but
could not find many. 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 test-
writing activity was rather low in the earlier phases, as JUnit was not yet well known
at that time. According to the change log, a first JUnit test has been introduced in
September 2001 (without JUnit being included in the repository). Follow up log
messages 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 onward. Around commit #1,750 the development branch
0.13 containing test cases is merged with the main development line. At that time, a
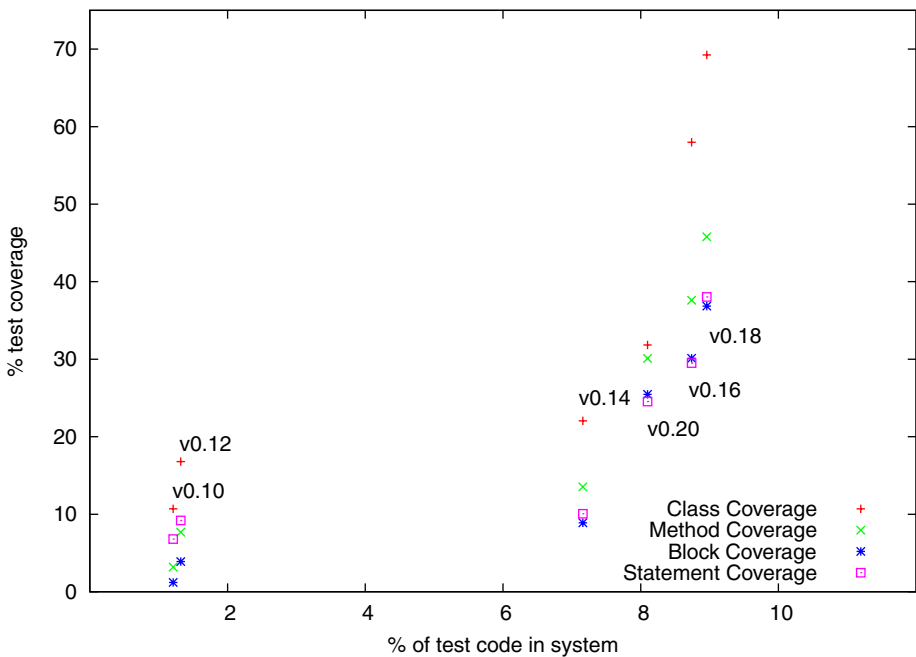
Fig. 9 Test Coverage Evolution View of ArgoUML

test suite as well as build targets for testing are first introduced. Therefore we accept this observation.

First we tried to reject the observation *"There are regular periods of phased testing"* (*ArgoUML.O.2*) by searching the log for commits where code and tests are changed together. We only found those during merges of branches to the main development line. The log indicates (e.g., #1,991 and #2,782) 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., #1,796, #2,166, #2,411, #2,811) and thus we accept the observation.

"The VCS structure has been heavily modified around commit #289" (*ArgoUML.O.3*). Looking at this period in the logs, we note how the original code is mainly stored in a `src/` directory of the main development line, while later additions and moves all happen in `src_new/`. Thus, we accept the observation.

"From 0.16 on, coding and testing happens in smaller increments, yet not synchronous." (*ArgoUML.O.4*) First we tried to reject this observation by looking for log messages indicating synchronous co-evolution in the period #6,100–#6,800, 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 thus we accept this observation.

"Version v0.20 of ArgoUML is characterized by a sudden drop in test coverage." (*ArgoUML.O.5*) During the coverage measurement, we discovered 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. We accept the observation, yet the explanation reveals that this is not due to a change in coding style but rather the result of an architectural decision.

6.3 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 during the period that we studied. 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 activities 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. They perceive that writing, maintaining and deploying test code for such systems requires a larger effort than for batch-oriented applications.

7 Case 3: Software Improvement Group

The industrial case study that we performed pertains to a software project from the *Software Improvement Group* (SIG). The SIG is a tool-based consultancy firm that is specialized in the area of quality improvement, complexity reduction and software renovation. The SIG performs static source code analysis to analyze software portfolios and to derive hard facts from software to assess the quality and complexity of a system.

For our study we investigate the development history of one of the SIG tools between March 2005 and January 2008. Over time 20 developers worked on this software project, which after around 3,500 commits results in 2,399 classes and 181 kSLOC.

As the build system of this case has built-in support for measuring coverage using the Clover coverage tool (Clover 2010), we opted to use those results rather than using Emma. Note however that different terminology is used (conditional coverage instead of block coverage) and that coverage measurements may differ for certain Java constructs (e.g., inner classes). Therefore one should not directly compare the results between this case and the previous two.

7.1 Observations

Change History View   At first sight, the Change History View of the SIG case is characterized by the large number of test dots (yellow and red). A new production

class is typically introduced together with the corresponding test case, which can be seen in Fig. 10 through the fact that the "addition line", which is normally black, is almost completely covered by red triangles. Furthermore, changes to production code are very frequently backed up by a change to the associated unit test. The steady growth of the code base over time happens incrementally, with no major jumps. Some larger increments occur right after v0.4 and v1.12 and before v1.5. Changes to many files occur seldomly, e.g., between v1.1 and v1.2, before v1.10 and v1.15.

From our analysis, we have also obtained some statistical info, namely that during the development of 1,859 production classes, 962 unit tests and 58 integration tests were written.

Growth History View The Growth History View of the SIG case shows a continuously synchronous development style, where coding and testing activities follow each other closely all the time (*SIG.O.1*—see Fig. 11). At no point during the considered time frame can we speak of testing time delays or backlogs. We interpret this as a development method where low-level testing happens alongside (or even before) development, such as eXtreme Programming (XP) (Beck 1999) or test-driven development (Beck 2003).

There do exist some larger increments during particular revisions that correspond timewise to somewhat larger commits in the Change History View (*SIG.O.2*). In the Change History View these larger increments are characterized by a vertical jump in the (black) addition line. These larger commits may be due to developer style or a merge from a branch. Even in those cases, all artifact entities grow together.

At certain points, the size of the artifacts drops. We expect that the developers of the SIG are refactoring or are deleting redundant, old or unused code fragments at revisions #1,981, #2,708, #3,105, #3,161 and #3,743 (*SIG.O.3*).
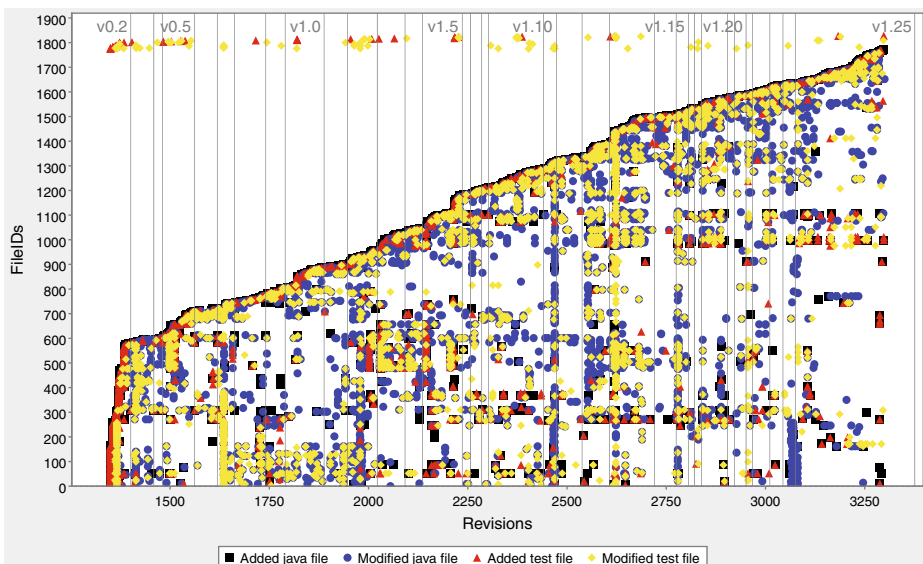

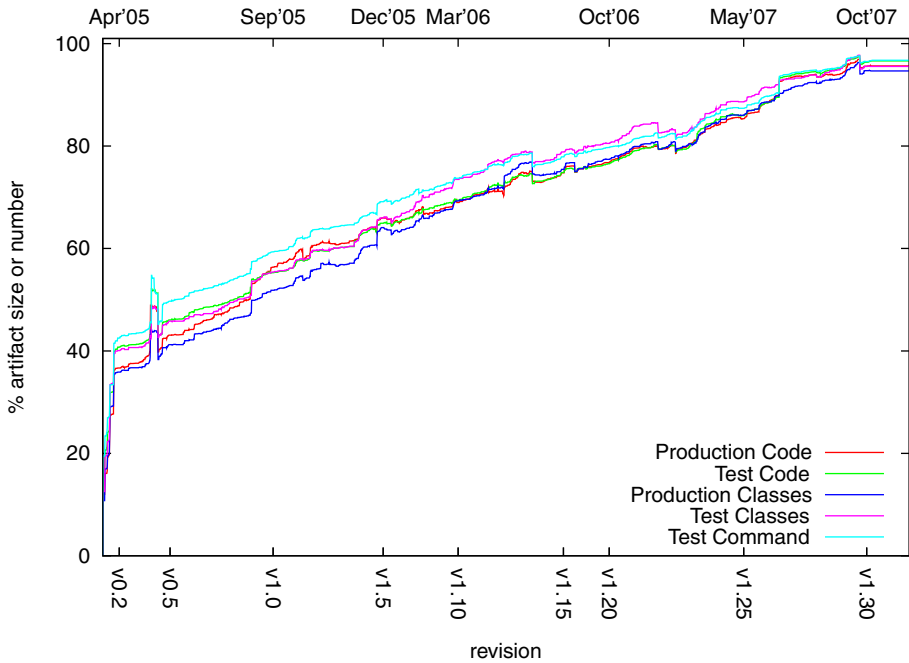**Fig. 10** SIG case Change History View

Fig. 11 Growth History View of the SIG case

To note in Fig. 11 is the bump just before release 0.5, which only exists for a limited number of revisions. This is probably due to some restructuring within the VCS system (*SIG.O.4*).

Test Coverage Evolution View   Figure 12 shows a stable development process over time as well as what looks like a linear relationship between coverage and *tRatio*. The (statement) coverage stays at a high level between 86% and 94% throughout the studied period, with *tRatio* only slightly varying between 57% and 60%. We observe a minor drop in coverage (−5%) at the time the project is reaching the 1.0 release. The coverage (as well as *tRatio*) remains at that somewhat lower level during the first 1.x releases, but increases again to above 90% later on. We derive from the figure that the unit testing approach being followed is uniform over time, based on the observation that statement coverage and *tRatio* increase and decrease together.

7.2 Internal Evaluation

"The spike around #1,500 (just before v0.5) is due to restructurings in the version control system" (*SIG.O.4*). Indeed, the log message at this points speaks about "CVS troubles".

  "Larger increments in development happen at revisions #1,503, #2,218 and #3,489" (*SIG.O.2*). The log entries for these revisions mention that files are imported from a branch and that a new analysis module is added, hence we confirm the observation.
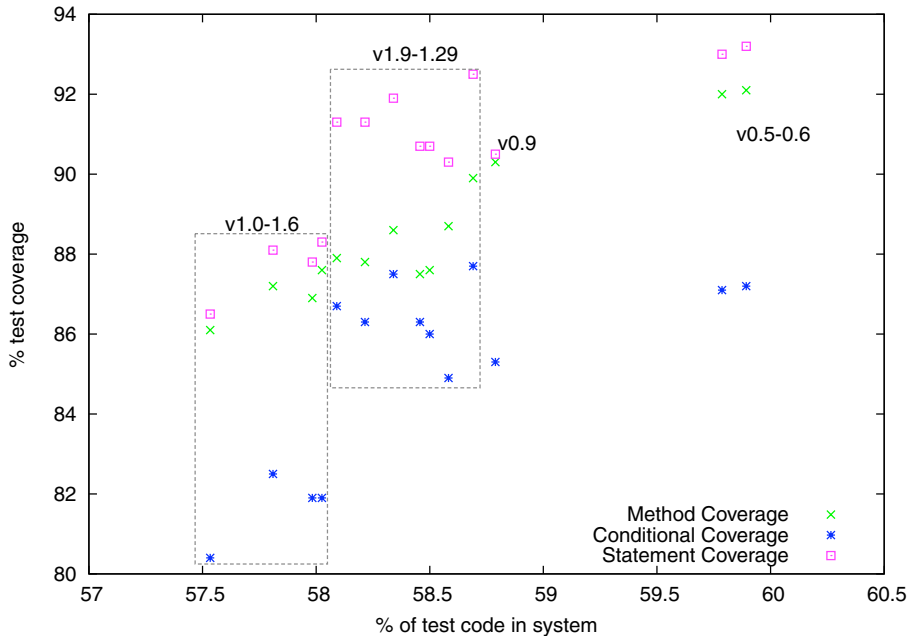
Fig. 12 SIG case Test Coverage Evolution View

"Old or redundant code is removed from the repository at revisions #1,981, #2,708, #3,105, #3,161 and #3,743." (*SIG.O.3*). The logs mention that untested source code and old analysis modules are being removed, hence we accept this observation.

7.3 External Evaluation

To evaluate our findings, we presented the results of our analysis to developers and managers of the company, followed by a general discussion. Afterwards, we interviewed two experienced developers of the SIG case in depth. For this interview, we used the survey questions of Table 2 as a starting point. However, as a result of the interactive discussion additional questions popped up.

The developers told us about the development strategy that they follow, which is a combination of Scrum (Schwaber and Beedle 2002) and eXtreme Programming. Test driven development is encouraged, yet not enforced. From the start, JUnit tests have been written to serve for verification as well as documentation purposes. Tests are written to demonstrate how classes are used as well as to document bugs, first to show their presence, next to show that they are fixed. This confirms our observation *SIG.O.1*. This approach, the developers claim, works well to integrate newcomers into the team. Other XP practices that are adopted at the SIG, such as pair programming, the shared code ownership, and a large, running test suite furthermore put social pressure on the developers to continue this strategy.

Unit tests are considered the responsibility of the individual developers. Developers commit feature-complete changes, containing code and tests directly into the main branch of the VCS. Daily builds report on the internal quality, including

test success and coverage results. Test coverage is desired to exceed 80%, which is perceived as feasible when testing happens right from the start. Developers avoid to build up a test backlog.

The tests are separated from production classes, and are tied to them using the "Test" suffix naming convention. These tests focus on single production methods. They sometimes grow large, as isolation from other tests is considered important and as such a lot of (recurring) setup is typically required.

Besides unit tests, the development team uses a front-end testing framework, as well as testing scripts that are part of the build system, to verify compilation, deployment and system launch. These tests were not considered in our analysis.

When presented with the results of the analysis, the audience was delighted to see the synchronous co-evolution patterns as well as the frequent test changes, as this reflects the test process that they intend to follow. Changes to the production code may only be committed together with the modified corresponding test code. The stability of the test process over time as seen in the views is due to the careful and continuous application of their development method.

Last but not least, the discussions at the SIG resulted in various small improvements to the TeMo tool and views, such as displaying time stamps, in addition to revision numbers, in the Growth History View. These additions make it easier for them to interpret the results in relation to time and releases. The SIG was quite happy with the potential of TeMo and asked an intern to apply TeMo on a number of other projects.

8 Discussion

8.1 Overall Evaluation of the Observations

For each of the cases, we generated the three visualisations and made a number of observations (see Sections 5.1, 6.1 and 7.1). As explained in the experimental set-up (Section 4) we then tried to confirm or reject these observations using the log messages and inspections of the source code. Furthermore, we have also shown all our observations to the original developers and hence have been subject to external validation. Some of the more relevant remarks of the developers are discussed below.

Table 3 shows an overview of these results (internal *and* external validation), illustrating that for two of the cases all observations are confirmed. Checkstyle is an exception and the rejected observations there are due to a variety of reasons, including unexpected side-effects of anonymous classes in Java (O.7) and changes in the JUnit test framework (O.2). Also in the case of Checkstyle observation O.6 could not be confirmed nor rejected, as the log messages are not explicitly mentioning the

Table 3 Evaluation of the observations made

Case	Confirmed	Rejected	Undecided
Checkstyle	O.1, O.3, O.4, O.5	O.2, O.7	O.6
ArgoUML	O.1, O.2, O.3, O.4, O.5	—	—
Software Improvement Group	O.1, O.2, O.3, O.4	—	—

testing strategy around that time and the developer that we sent the questionnaire to did not respond to our specific remark.

However, as is typical during quality assessments, the relevance of the observation is much more important than the fact whether it is right or wrong, as even wrong observations can lead to interesting insights. The answers to the questionnaires we sent to the developers are particularly interesting in that respect as they give an indication about the relevance of the observations and below we give some examples.

Checkstyle As an example of a wrong observation which is interesting anyway we comment on observation *Checkstyle.O.7*. This observation was rejected because it was not so much a matter of neglecting tests, but rather a matter of the introduction and subsequent removal of anonymous classes. Yet, for a consultant, this is still very interesting information, because it clearly showed that when the anonymous classes were present in the system, they were not tested directly. As a consequence, he could formulate a recommendation that if the developers were to use anonymous classes again, they should ideally be tested in a direct way.

Also, the Checkstyle developer mentions that in principle they apply test-driven development, but a quality engineer making observation *Checkstyle.O.5* (a period of intense development) knows that he will have to warn his team members about decreasing testing vigilance. On the other hand, the *Checkstyle.O.4* (synchronous co-evolution) observation would allow the quality engineer to congratulate the team members.

ArgoUML At a certain point in time, the ArgoUML team was making a conscious decision to adopt JUnit tests. A quality engineer within the ArgoUML team can use the Change History View and observations like *ArgoUML.O.1* (slowly increasing test activity) to demonstrate to the team members the progress that is made. On the other hand, an external consultant making observation *ArgoUML.O.5* (dropping test coverage revealing a component extracted from the core) can use this information nugget to start a discussion on the current architecture of the system and the rationale behind it. In the same vein, an external consultant might question—just as we did— the relatively low test coverage over the project's history and would see that this is again a conscious decision by the project team motivated by the nature of the system (desktop GUI application). Such discussions are important during an external quality assessment, as they expose the tacit knowledge surrounding the development practices within the group.

Software Improvement Group An external consultant making observations *SIG.O.2* (larger increments of development) and *SIG.O.4* (removal of old or redundant code) can use these restructurings to trigger a discussion on the architecture and the rationale behind it. The quality engineer in the SIG development team can use the Growth History View to monitor the test activity and conclude that no extra encouragement is needed for test-driven development, a practice which is (as said during the interviews) "encouraged, yet not enforced". The quality engineer can however congratulate the team, because the team is following test-driven development very well.

Overall The three case studies that we have selected are quite diverse in their take on testing. For instance, ArgoUML introduced developer testing quite late in the

project's history, after which they have followed a phased testing approach. The SIG case on the other hand is an example of a synchronous co-evolution activity right from the start. Checkstyle meanwhile, started off with a single developer test, evolving into a test suite that was maintained in a phased manner, which in turn evolved into a synchronous test/development process.

8.2 The Research Questions Revisited

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 witness:

–   Red or yellow vertical bars indicating periods of intense testing in Checkstyle and ArgoUML. These periods of intense testing were often separated from each other by several months.
–   Test dots on top of production dots as indicators for the simultaneous introduction and modification of production code with corresponding unit tests (e.g., the SIG case and during certain periods for Checkstyle).

By means of the Growth History View, we can see how the production and test artifacts do or do not grow together over time:

–   We observe testing backlogs in ArgoUML and Checkstyle, where development takes the upper hand during the early stages of a system's evolution.
–   We observe a phased testing approach in ArgoUML, as evidenced by the continuous growth in production artifacts and the stepwise growth in testing artifacts.
–   Synchronous co-evolution can be seen in the SIG case, by means of a continuous, steady growth of all artifacts.

RQ2   Can an increased test-writing activity 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 both 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 activity 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.

In the industrial case study, developers practice continuous integration (a key XP practice) and Scrum sprints to yield frequent releases. As a result, there are no major additional test activities before releases.

However, even though we did not observe any increases in test writing activity before a release, our views are capable of detecting periods of increased testing, e.g., by red or yellow vertical bars in the Change History View and sharp increases in test artifacts in the Growth History View. In Hindle et al.'s work on release patterns, an increase in test-writing and documentation activity is reported before releases of the MySQL database system (Hindle et al. 2007b).

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 production source file alongside its unit test. We found indications of test-driven development in the SIG case and during certain periods of Checkstyle, by means of "test" dots on top of "code" dots in the Change History View, signifying concurrent introduction as well as co-evolution. These observations are backed-up by the results of the developer surveys: later on during Checkstyle's evolution, commits are self-contained, which is also a required practice among the SIG developers.

RQ4   Is there a relation between test-writing activity and test coverage? For the three considered case studies we observe that test coverage grows alongside test code fraction, especially during periods of steady, incremental development. During major restructurings however, e.g., as we see for the 2.x Checkstyle releases, this relation can be disturbed.

We furthermore observe how both coverage and *tRatio* seem to increase as the project matures, except the SIG case where both metrics remain stable from the start.

While we do not want to make a direct comparison between cases, of interest to note is the difference between Checkstyle and the SIG case. In particular, when comparing the test coverage of the later versions of Checkstyle and the SIG case, we observe similar coverage levels of between 80 and 90%. All the while, the amount of test code needed to reach that level of coverage is strikingly different. For Checkstyle, the amount of test code approaches 25% of the total source code, while the *tRatio* observed in the SIG case varies between 58 and 60%. This is surprising as it surmounts the contribution of test code to the overall source code that is described in literature, where numbers between 10 and 50% are mentioned (Yamaura 1998; Beck 2003; Gaelli et al. 2004; Sangwan and Laplante 2006; Moonen et al. 2008; Van Rompaey and Demeyer 2008).

These numbers might suggest that one project is tested more efficiently in terms of amount of test code necessary to reach a certain level of test coverage. However, code inspection and developer interviews revealed that a different test strategy is being used. More specifically, the SIG developers concentrate on writing isolated, self-contained unit tests. Checkstyle developers on the other hand see their tests more as I/O integration tests, yet associate individual test cases with a single production class by name. The former type of tests typically run faster and are better at defect localization (Meszaros 2007), while the latter can also serve as acceptance tests. As a consequence of these different testing strategies, the Test Coverage Evolution View cannot be used to compare across projects. Difference in testing strategy and the consequences on test coverage have also been described by Kanstrén, who proposes to measure test coverage separately for each level of testing (Kanstrén 2008).

During our own study, we found that ArgoUML has, next to the unit test suite, a separate suite of automated GUI tests. These GUI tests are not part of the version control system.

Also, the *quality focus* of the developers of the respective projects can play a major role in the test coverage that is obtained. Developers that actively measure and act upon test coverage are more likely to detect opportunities for increases in test coverage and refactoring of the tests. We have witnessed log messages about such activities in Checkstyle and the SIG case.

Another factor of influence is 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 (Bruntink and van Deursen 2006). This means that the design of the system under test has an influence on the test-writing activity required to reach a certain coverage criterion.

Main research question *"Can we establish the developer testing process that is being followed by mining a version control system (VCS)?"* By answering RQ1 through RQ4, we have shown that our approach is capable of providing a significant level of insight into the developer testing process. In particular, our approach is able to detect whether production and test code are being developed and/or maintained at the same time and whether increased periods of test writing activities are taking place (before a release or at another point in time). Furthermore, we have been able to observe a test-driven development style and we have seen indications that as a software project's development matures, the fraction of test code present in the source code base increases (or stabilizes) alongside test coverage.

8.3 Post-Mortem Analysis of the Visualization Approach

Characteristics of the visualizations used From our experiences using the visualizations that we created for the purposes of studying the co-evolution of production and test code, we made the following observations:

– The visualizations allowed us to quickly gain insight into the testing process of the software system under study, to witness changes in development and testing patterns and to make an initial assessment as to how much we could rely on the existing testing suite. Gathering the same level of insight without the visualizations, e.g., by browsing the log files and/or interviewing the developers/maintainers, is much more time-intensive and would not have allowed us to obtain a broad overview of development and testing practices over a considerable period of time.
– The visualizations scale reasonably well. In particular, the Change History View scales well to large projects, due to its relative lightweightedness – it only needs a dump of VCS activities – and the view itself is easy to navigate and allows for zooming, thus giving it a limited form of interactivity (Lanza 2010). The Growth History View takes quite some time to generate (ranging from several hours to several days, depending on the size of the project and time period under consideration), but once generated is intuitive to use. The current implementation has no zooming capabilities, rendering this view more difficult to use in very large scale projects. The Test Coverage Evolution View is created manually, mainly due to the difficulties that we had in obtaining test coverage data automatically from 2 out of our 3 projects. Due to the limited number of data points that we collected manually, the visualization is easy to interpret, however, it is not so easy to generate.

Complementarity of the three evolution views When performing our case studies, we experienced that our three views are complementary when trying to assess the

"test health" of a software process. Firstly, when the Change History View indicates a change to a large number of artifacts, we need the Growth History View to determine whether the change actually entails the addition of new production or test code, or whether the changes are more shallow, of which typical examples are code beautification or import-statement optimization. Secondly, while the Change History View and Growth History View tell something about the test-writing activity of the developers, they tell little to nothing about the potential of the test suite to detect defects. This is where the Test Coverage Evolution View comes in and relates the test writing activity to the test coverage.

9 Threats to Validity

In this section we identify factors that may jeopardize the validity of our results and the actions we took to reduce or alleviate the risk. Consistent with the guidelines for case studies research (see Runeson and Höst 2009, Yin 2002) we organize them into four categories.

Construct validity: does the case study investigate what was intended?   For the external evaluation we have either sent a survey to the developers or we have interviewed the developers ourselves. In order to mitigate the *leading question bias*—i.e., the pressure participants might feel to answer positively, we have explicitly asked the developers who responded to our survey or participated in our interview to have an open and honest discussion. We also let them see and approve our text before publication; nobody made any objections to the text.

We have identified two variation factors of the development process with regard to the use of the version control system (namely the commit style and the use of branching) which may affect the visualizations. The individual *commit style*—short cycles, one commit per day, ...—has an effect on the granularity of the commits, hence on the number of dots in the visualizations. Moreover, the Checkstyle developers have 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. Secondly, developers can use *branching*. In ArgoUML, developers use branches to fix certain bugs. In the SIG case however, branching is not a common practice. In the Change and Growth History Views, merging a branch back into the main development line gives a similar result as a large commit. If a large part of a project's development effort happens in branches, it can be useful to specifically apply the views to these branches. Since the views are mainly used to look for general trends, we expect this effect of the commit style and branching to be minimal. However, this is certainly something that needs to be considered when replicating the case study.

Finally, we note that when a production class is moved within the repository, e.g., when the class changes package, a new horizontal line is introduced in the Change History View. The associated test class—if any—however, is now associated with both the old and the new instance of the class in the Change History View. This is because the strategy that we follow cannot determine with certainty that the old instance is in fact deleted (this is related to a technicality that originates from a sometimes faulty cvs2svn conversion script). As a consequence we also cannot

determine whether the old instance will never be changed again. As a result, the Change History View can sometimes feature more yellow dots for a particular revision than would be expected. We have investigated this phenomenon with regard to our case studies and the number of times this has happened seems low, thus limiting its impact on the results.

Internal validity: are there unknown factors which might affect the causal relationship? Since this is an exploratory case study, we did not seek for causal relationships, hence this category of risks is less important. However, the way we set up our cross-validation—with an internal (inspection of the log-messages) and external evaluation (interviews with the developers)—might miss certain context information that may provide a better explanation of some of the observations.

Concretely, for the internal evaluation we use the versioning system's log messages to confirm or reject our observations. As no strict conventions are in place for what should be specified in such messages, there are large differences in the content and quality of log messages across projects, tasks and developers. Hence there is a risk that the developers did not bother to record important context information into their log message and that we consequently misclassified an observation into confirmed/rejected/undecided. In response to findings by Bachmann and Bernstein, who report that 20% of the log messages in the Eclipse version control system remained empty (Bachmann and Bernstein 2010), we performed a similar analysis for our three case studies. The results can be found in Table 4. Our readings suggest that the developers filled in the log messages rather conciously. Although, these readings provide no guarantees as to the quality of the log messages, they do provide an indication that the log messages can be used for validation purposes.

Concerning the external evaluation, the number of persons that cooperated in the survey or interview was limited to one or two individuals per case study. As such, we concede that their view on the project's history and crucial events is limited and that they might miss crucial context information.

External validity: to what extent it is possible to generalize the findings? First of all, we must point out the inherent limitations of studying the testing process by analyzing the contents of a VCS. The focus of our approach is on testing activities that are performed by the developers themselves, i.e., unit testing and integration testing, as these tests are typically codified and stored into the VCS alongside the production code. We acknowledge that the testing process is much more than only unit and integration testing. However, when the tests are not code and not stored in the VCS, we have no means of involving these tests in our VCS-based approach.

We have selected the cases as being representative for software systems produced in smaller teams (5 to 10 core developers) where team members take up varying roles (analyst-designer, programmer, tester, debugger) and were quality assurance is

Table 4 Number of empty log messages

	Checkstyle	ArgoUML	SIG
Total log messages	2,258	7,477	2,838
Empty log messages	2	162	98
Percentage empty log messages	0.09%	2.17%	3.45%

mostly integrated into the daily activities of the team members. We cannot make any claims about the use of the views in larger teams with a separate quality assurance department. Also, the three cases cover three quite distinct testing cultures and the tool has been able to detect symptoms of an XP/Scrum-like development process. Nevertheless, for the moment we do not make any claims regarding the capability of detecting other development processes.

Reliability: is the result dependent on the researchers and tools?   The observations made on each of the views were collected and analyzed by the developers of the co-evolution views themselves. The same holds for the cross-validation against an internal and external perspective. Therefore, there is the risk that other researchers would make other observations or find other supporting or contradicting evidence. We reduced this risk by having a systematic procedure for both the interpreting of the diagrams (described in Section 2) and the cross-validation (described in Section 4).

The TeMo toolchain might contain faults which explain the results of the case studies. As a countermeasure, we thoroughly tested the tool and we relied as much as possible on widely used components with a reputation of reliability (JFreeChart, Emma, Clover, XQuery, etc.).

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`. This naming convention is promoted in literature and tutorials (Gamma and Beck 1998; Fewster and Graham 1999; Feathers 2005) and all of our case studies adhered to this naming convention quite strictly. Nevertheless, to ensure that the matching is indeed correct, we have investigated the generated production/test code pairs and we found two false positives (over all of the three case studies), which we have removed with an exception list in our configuration.

10 Related Work

D'Ambros et al. provide an overview of the research performed in the area of *Analyzing Software Repositories to Understand Software Evolution* (D'Ambros et al. 2008). Their work gives a good overview of software evolution related work that is being done in the *Mining Software Repositories* community (Hassan et al. 2005).

Within this area, we have identified three research subdomains that are relevant in the context of this work: (i) software visualization (targeting software evolution) and (ii) research on traceability and co-changes and (iii) empirical studies that investigate the efficacy of testing strategies. This section highlights some of the contributions in these areas that lie closest to our research.

Visualizing software evolution   Visualizing the revision history of a set of source code entities has been used to study how these entities co-evolve. Van Rysselberghe and Demeyer (2004) use a very simple, yet effective visualization of the history of a software system in order to identify the most valuable and problematic parts of a software system. Their approach allows them to identify unstable components, design evolutions and fluctuations in team productivity.

Wu et al. (2004) use the concept of *spectographs* to visualize how metric-values for software entities evolve over time. Spectrographs allow to visualize the decay

of a particular property over time, e.g., an entity that has just been changed can be colored red, while a component that has been changed a little while longer ago, can be colored orange.

Lanza (2001) and Lanza and Ducasse (2002) extend on the previous idea of poly-metric views (Lanza and Ducasse 2003) by using a visualization they call the *evolution matrix*. This visualization depicts time on the X axis, entities of the software system on the Y axis and they use polymetric views, i.e., boxes of different colors and shapes to express properties of the entities over time.

The aforementioned visualizations are similar through the fact that they all use one axis to represent time, while the other axis represents the source code entities. This visualization-approach has been used to detect logical coupling between files, determine the stability of classes over time, developer expertise and interaction, 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 *Evolution Radar* by D'Ambros et al. (2009) on the other hand does not use the typical XY-chart visualization. Rather, it uses the idea of a radar to visualize the historical information of a software system. The evolution radar can be used to analyze change couplings to detect architectural decay and coupled components.

Evolens is a graph-based visualization technique developed by Ratzinger et al. (2005). It represents the directory structure and source files as nested graphs. Change coupling dependencies between source files are visualized as arcs between nodes, where the width of the arcs is determined by the number of times two source files are committed together. A focal point, a user-selectable entity of interest, is used to focus the analysis on the change coupling relationships of a specific entity.

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 (Godfrey and Tu 2000). To a certain degree, our research interests are similar as we investigate whether produc-tion code and test code grow at similar or different points in time during a project's history in our Growth History View.

Telea and Auber present *Code Flows* (Telea and Auber 2008), a visualization technique that allows to analyze the structural evolution of source-code at a level higher than the line level, but below the file level.

Voinea and Telea developed a framework for visual data mining of version control repositories (Voinea and Telea 2006). Their framework delivers a basic toolbox for data acquisition, analysis and visualization of historical data. Of particular interest, is that they also propose a new technique for identifying cluster of files with similar evolution, which can help users to perform a logical decomposition of the system and to predict future changes.

Other work by Voinea et al. is CVSScan (Voinea et al. 2005). CVSScan is an interactive tool that makes use of dense pixel displays to show the overall evolution of code structure, semantics and attributes. A user study with CVSScan revealed that users were able to easily spot issues at a high-level and then drill down to get more information. In Voinea and Telea (2007) continued and refined their work.

Co-changes   Another branch of research in the same area does not rely on visu-alization but identify logical coupling, i.e., coupling that might not be immediately noticeable from analyzing the source code, but coupling that is visible through the

fact that certain entities are frequently committed together. Seminal work in this area is the work of Gall et al. (1998) and Ball et al. (1997).

In the domain of co-changes, Beyer and Noack visualize the software history by displaying sequences of cluster layouts based upon co-change graphs (Beyer and Noack 2005). These graphs consist of files as nodes and the level of co-change as weighted edges. Beyer and Hassan continued on this idea and added animation to follow the evolution of software throughout time (Beyer and Hassan 2006).

To identify co-changing lines, Zimmermann et al. (2006) build an annotation graph based upon the identification of lines across several versions of a file.

Kagdi et al. (2007) 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. (2007a) studied the release-time activities for a number of software artifacts: source code files, test files, build files and documentation. In particular, they examine four open source systems by counting and comparing the number of revisions in the period before and after a release. They summarize the observed behavior in a condensed notation. In contrast to our own study, Hindle et al. (2007b) report an increased test writing activity just before releases of the MySQL database system. Even though we did not observe any increases in test writing activity before a release in our three case studies, our views are capable of detecting periods of increased testing, e.g., by red or yellow vertical bars in the Change History View and sharp increases in test artifacts in the Growth History View.

Fluri et al. examine whether source code and associated comments are changed together alongside the evolutionary history of a software system (Fluri et al. 2007). 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. Conceptually, the main difference is of course that Fluri et al. concentrate on a different kind of co-evolution, namely that of source code and documentation instead of production and test code.

Bouktif et al. propose an approach based on dynamic time-warping to answer the question: given a software system and one file under modification, what other files must be changed (Bouktif et al. 2006)? Their approach, called Synchrony, obtains high recall and precision levels when compared to the opinion of experts.

There have also been efforts to use co-change information for aspect mining. In particular, Breu and Zimmermann (2006), Adams et al. (2010) and Mulder and Zaidman (2010) have made efforts in this area.

Empirical studies Bhat and Nagappan investigate whether applying test-driven development improves the code quality of software (Bhat and Nagappan 2006). They studied two projects at Microsoft and found that while applying test-driven development takes more time up front, there is a significant increase in code quality. As a side note, they also note that one of the additional benefits of early testing is the availability of documentation in the form of tests.

Siniaalto and Abrahamsson report on a comparative case study in which they measure the effect of test-driven development with regard to program design and test coverage (Siniaalto and Abrahamsson 2007). While they did not find a conclusive effect of test-driven development on the design of the software systems, their

research does indicate that the test coverage obtained with test-driven development is superior to iterative test-last development.

Relation to our own work   The work described in this paper builds upon many of the aforementioned ideas, e.g., it uses simple XY-chart visualizations to depict the evolution of a software system. While the visualizations we use are quite similar to those previously presented, the actual subject of investigation, namely the co-evolution of production and test code is a unique new angle of research in the domain.

11 Conclusion & Future Work

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

1. We introduce three views: (i) the Change History; (ii) the Growth History; and (iii) the Test Coverage Evolution View; and combine them to study how test code co-evolves over time.
2. We demonstrate and validate the use of these views on two open source cases and one industrial case by making several relevant observations about the testing processes used in the development.

In particular, across our cases we witness phased testing approaches and more synchronous co-evolution, corresponding to what is to be expected from the development style that is being followed by the developers (RQ1). Our case studies do not show an increase in testing activity before major releases, but we did recognize periods of intense testing in the development's history (RQ2). We found evidence of test-driven development in two of our case studies, which showed up as tests being committed alongside production code (RQ3). The fraction of test code in the source code tends to increase with increasing coverage, yet, we have to be careful to compare the Test Coverage Evolution View across projects due to differences in testing strategies. Coming back to our central research question, we find that mining a version control system provides us with a significant level of insight into the testing process.

Some future directions that we have identified for this line of work are: (i) to apply our analysis to additional (larger) cases to characterize other development methods, (ii) to refine our analysis so that we can discern between artifacts of different testing levels, e.g., unit tests, integration tests, etc., (iii) to automatically filter out shallow changes to code in the Change History View, (iv) to perform a user study with the TeMo tool (similar to Cornelissen et al. 2009) (v) to continue investigating statistical techniques to the change history data so that phenomena like test-driven development or phased testing no longer have to be visually recognized (Lubsen et al. 2009) and (vi) to continue investigating the relationship between the testing strategy followed and the efficiency of the issue handling process (Luijten et al. 2010).

---

[8]http://www.sig.nl, Amsterdam, The Netherlands.

References

Adams B, Jiang ZM, Hassan AE (2010) Identifying crosscutting concerns using historical code changes. In: Proceedings of the international conference on software engineering (ICSE). ACM, New York, pp 305–314

Ant (2010) http://ant.apache.org. Website last visited: 4 August 2010

Bachmann A, Bernstein A (2010) When process data quality affects the number of bugs: correlations in software engineering datasets. In: Proceedings of the 7th international working conference on mining software repositories (MSR). IEEE Computer Society, Washington, pp 62–71

Ball T, Eick SG (1996) Software visualization in the large. IEEE Comput 29(4):33–43

Ball T, Kim J, Porter A, Siy H (1997) If your version control system could talk. In: ICSE workshop on process modelling and empirical studies of software engineering. Boston, USA

Beck K (1999) Extreme programming explained: embrace change. Addison Wesley, Reading, MA

Beck K (2003) Test-driven development: by example. Addison-Wesley, Reading, MA

Berner S, Weber R, Keller RK (2007) Enhancing software testing by judicious use of code coverage information. In: Proceedings of the international conference on software engineering (ICSE). IEEE Computer Society, Los Alamitos, pp 612–620

Beyer D, Hassan AE (2006) Animated visualization of software history using evolution storyboards. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE Computer Society, Washington, pp 199–210

Beyer D, Noack A (2005) Clustering software artifacts based on frequent common changes. In: Proceedings of the international workshop on program comprehension (IWPC). IEEE Computer Society, Washington, pp 259–268

Bhat T, Nagappan N (2006) Evaluating the efficacy of test-driven development: industrial case studies. In: Proceedings of the international symposium on empirical software engineering (ISESE). ACM, New York, pp 356–363

Bible J, Rothermel G, Rosenblum D (2001) A comparative study of coarse- and fine-grained safe regression test selection. ACM Trans Softw Eng Methodol 10(2):149–183

Binder R (2000) Testing object-oriented systems: models, patterns, and tools. Addison-Wesley, Reading, MA

Bouktif S, Guéhéneuc YG, Antoniol G (2006) Extracting change-patterns from cvs repositories. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE Computer Society, Washington, pp 221–230

Breu S, Zimmermann T (2006) Mining aspects from version history. In: Proceedings of the international conference on automated software engineering (ASE). IEEE Computer Society, Washington, pp 221–230

Bruntink M, van Deursen A (2006) An empirical study into class testability. J Syst Softw 79(9):1219–1232

Clover (2010) http://www.atlassian.com/software/clover/. Website last visited: 4 August 2010

Cornelissen B, Zaidman A, van Deursen A, Van Rompaey B (2009) Trace visualization for program comprehension: a controlled experiment. In: Proceedings of the international conference on program comprehension (ICPC). IEEE Computer Society, Washington, pp 100–109

D'Ambros M, Gall H, Lanza M, Pinzger M (2008) Analyzing software repositories to understand software evolution. In: Mens T, Demeyer S (eds) Software evolution. Springer, Berlin, pp 39–70

D'Ambros M, Lanza M, Lungu M (2009) Visualizing co-change information with the evolution radar. IEEE Trans Softw Eng 35(5):720–735

Demeyer S, Ducasse S, Nierstrasz O (2002) Object-oriented reengineering patterns. Morgan Kaufmann, San Francisco

Elbaum S, Gable D, Rothermel G (2001) The impact of software evolution on code coverage information. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 170–179

Emma (2010) http://emma.sourceforge.net/. Website last visited: 4 August 2010

Feathers M (2005) Working effectively with legacy code. Prentice Hall

Fewster M, Graham D (1999) Software Test Automation: effective use of test execution tools. ACM Press/Addison-Wesley Publishing Co, New York

Fluri B, Würsch M, Gall H (2007) Do code and comments co-evolve? On the relation between source code and comment changes. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE Computer Society, Washington, pp 70–79

Gaelli M, Lanza M, Nierstrasz O, Wuyts R (2004) Ordering broken unit tests for focused debugging. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 114–123

Gall H, Hajek K, Jazayeri M (1998) Detection of logical coupling based on product release history. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 190–197

Gamma E, Beck K (1998) Test infected: programmers love writing tests. Java Rep 3(7):37–50

Gîrba T, Ducasse S (2006) Modeling history to analyze software evolution. J Softw Maint 18(3):207–236

Glover A (2006) In pursuit of code quality: don't be fooled by the coverage report. http://www.ibm.com/developerworks/java/library/j-cq01316/. Last visited: 4 August 2010

gnuplot (2010) http://www.gnuplot.info. Website last visited: 4 August 2010

Godfrey M, Tu Q (2000) Evolution in open source software: a case study. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 131–142

Hassan AE, Mockus A, Holt RC, Johnson PM (2005) Special issue on mining software repositories. IEEE Trans Softw Eng 31(6):426–428

Hindle A, Godfrey M, Holt R (2007a) Release pattern discovery: a case study of database systems. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 285–294

Hindle A, Godfrey M, Holt R (2007b) Release pattern discovery via partitioning: methodology and case study. In: Proceedings of the international workshop on mining software repositories (MSR). IEEE Computer Society, Washington, pp 19–26

JFreeChart (2010) http://www.jfree.org/jfreechart/. Website last visited: 4 August 2010

Kagdi H, Collard ML, Maletic JI (2007) A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J Softw Maint 19(2):77–131

Kagdi H, Maletic J, Sharif B (2007) Mining software repositories for traceability links. In: Proceedings of the international conference on program comprehension (ICPC). IEEE Computer Society, Washington, pp 145–154

Kanstrén T (2008) Towards a deeper understanding of test coverage. J Softw Maint 20(1):59–76

Lanza M (2001) The evolution matrix: recovering software evolution using software visualization techniques. In: Proceedings of the international workshop on principles of software evolution (IWPSE). ACM, pp 37–42

Lanza M (2010) The visual terminator. In: Proceedings of the 7th international working conference on mining software repositories (MSR). IEEE Computer Society, http://www.slideshare.net/michele.lanza/the-visual-terminator

Lanza M, Ducasse S (2002) Understanding software evolution using a combination of software visualization and software metrics. In: Proceedings of LMO 2002 (Languages et Modèles à Objets). Hermes Publications, pp 135–149

Lanza M, Ducasse S (2003) Polymetric views—a lightweight visual approach to reverse engineering. IEEE Trans Softw Eng 29(9):782–795

Lehman M (1980) On understanding laws, evolution and conservation in the large program life cycle. J Syst Softw 1(3):213–221

LOCC (2010) http://csdl.ics.hawaii.edu/Plone/research/locc/. Website last visited: 4 August 2010

Lubsen Z, Zaidman A, Pinzger M (2009) Using association rules to study the co-evolution of production & test code. In: Proceedings of the 6th international working conference on mining software repositories (MSR). IEEE Computer Society, Washington, pp 151–154

Luijten B, Visser J, Zaidman A (2010) Assessment of issue handling efficiency. In: Proceedings of the international working conference on mining software repositories (MSR). IEEE Computer Society, Washington, pp 94–97

Mathur AP (2008) Foundations of software testing. Pearson Education

Maven (2010) http://maven.apache.org. Website last visited: 4 August 2010

Maximilien E, Williams L (2003) Assessing test-driven development at IBM. In: Proceedings of the international conference on software engineering (ICSE). IEEE Computer Society, Washington, pp 564–569

Mens T, Wermelinger M, Ducasse S, Demeyer S, Hirschfeld R, Jazayeri M (2005) Challenges in software evolution. In: Proceedings of the international workshop on principles of software evolution (IWPSE). IEEE Computer Society, Washington, pp 13–22

Meszaros G (2007) xUnit test patterns: refactoring test code. Addison-Wesley, Reading, MA

Moonen L, van Deursen A, Zaidman A, Bruntink M (2008) The interplay between software testing and software evolution. In: Mens T, Demeyer S (eds) Software evolution. Springer, Berlin, pp 173–202

Mulder F, Zaidman A (2010) Identifying cross-cutting concerns using software repository mining. In: Proceedings of the 4th international joint ERCIM/IWPSE symposium on software evolution (IWPSE-EVOL). ACM (To appear)

Prevalence and incidence of color blindness (2010) http://www.wrongdiagnosis.com/c/color_blindness/prevalence.htm. Website last visited: 4 August 2010

Ratzinger J, Fischer M, Gall H (2005) Evolens: lens-view visualizations of evolution data. In: Proceedings of the international workshop on principles of software evolution (IWPSE). IEEE Computer Society, Washington, pp 103–112

Runeson P (2006) A survey of unit testing practices. IEEE Softw 25(4):22–29

Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empir Software Eng 14(2):131–164

Sangwan R, Laplante P (2006) Test-driven development in large projects. IT Pro 8(5):25–29

Schwaber K, Beedle M (2002) Agile software development with Scrum. Prentice Hall, Upper Saddle River

Siniaalto M, Abrahamsson P (2007) A comparative case study on the impact of test-driven development on program design and test coverage. In: Proceedings of the first international symposium on empirical software engineering and measurement (ESEM). IEEE Computer Society, Washington, pp 275–284

Slaughter SA, Harter DE, Krishnan MS (1998) Evaluating the cost of software quality. Commun ACM 41(8):67–73

Storey MA, Čubranić D, German D (2005) On the use of visualization to support awareness of human activities in software development: a survey and a framework. In: Proceedings of the symposium on software visualization. ACM, New York, pp 193–202

SVNKit (2010) http://svnkit.com/. Website last visited: 4 August 2010

Tassey G (2002) Economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology (NIST)

Telea A, Auber D (2008) Code flows: visualizing structural evolution of source code. Comput Graph Forum 27(3):831–838

Van Rompaey B, Demeyer S (2008) Estimation of test code changes using historical release data. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE Computer Society, Washington, pp 269–278

Van Rompaey B, Du Bois B, Demeyer S, Rieger M (2007) On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans Softw Eng 33(12): 800–817

Van Rysselberghe F, Demeyer S (2004) Studying software evolution information by visualizing the change history. In: Proceedings of the international conference on software maintenance (ICSM). IEEE Computer Society, Washington, pp 328–337

Voinea L, Telea A (2006) An open framework for cvs repository querying, analysis and visualization. In: Proceedings of the international workshop on mining software repositories (MSR). ACM, New York, pp 33–39

Voinea L, Telea A (2007) Visual data mining and analysis of software repositories. Comput Graph 31(3):410–428

Voinea L, Telea A, van Wijk JJ (2005) Cvsscan: visualization of code evolution. In: Proceedings of the ACM symposium on software visualization (SOFTVIS). ACM, pp 47–56

Wu J, Holt RC, Hassan AE (2004) Exploring software evolution using spectographs. In: Proceedings of the working conference on reverse engineering (WCRE). IEEE Computer Society, Washington, pp 80–89

XQuery (2010) http://www.w3.org/TR/xquery/. Website last visited: 4 August 2010

Yamaura T (1998) How to design practical test cases. IEEE Softw 15(6):30–36

Yin RK (2002) Case study research: design and methods, 3rd edn. Sage Publications

Zaidman A, Van Rompaey B, Demeyer S, van Deursen A (2008) Mining software repositories to study co-evolution of production & test code. In: Proceedings of the international conference on software testing, verification, and validation (ICST). IEEE Computer Society, Washington, pp 220–229

Zhu H, Hall PA, May J (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427

Zimmermann T, Kim S, Whitehead J, Zeller A (2006) Mining version archives for co-changed lines. In: Proceedings of the international workshop on mining software repositories (MSR). ACM, New York, pp 72–75

Andy Zaidman is an assistant professor at the Delft University of Technology, The Netherlands. He obtained his M.Sc. (2002) and Ph.D. degree (2006) in Computer Science from the University of Antwerp, Belgium, after which he became a post-doctoral researcher at the Delft University of Technology. His main research interests are reverse engineering with the help of dynamic analysis, program comprehension, software repository mining and software testing. He is the founder and organizer of the International Workshop on Program Comprehension through Dynamic Analysis (PCODA) series which started in 2005. He was the general chair of the 15th Working Conference on Reverse Engineering (WCRE 2008) held in Antwerp, Belgium, and the program co-chair of WCRE 2009 held in Lille, France.

Bart Van Rompaey received the MSc degree in computer science from the University of Antwerp, Belgium, in 2004 and the MSc degree in industrial management from the Catholic University of Leuven, Belgium, in 2005. He then became a PhD student in the Laboratory on Reengineering (LORE) at the University of Antwerp, Belgium. His research interests include reverse engineering and reengineering of large evolving software systems, with a particular focus on software tests. He obtained his Ph.D. in Spring 2009 and is currently working for LoQutus.

Arie van Deursen is a full professor at Delft University of Technology, where he is heading the Software Engineering Research Group. He obtained and his PhD degree from the University of Amsterdam in 1994. His research interests include software evolution, program comprehension, and software testing. He is general chair of the Mining Software Repositories 2011 (MSR'11) conference, and program chair of the International Conference on Program Comprehension 2012 (ICPC'12).

Serge Demeyer is a professor at the University of Antwerp, where he leads the Lab on Reengineering (LORE) part of the Antwerp Systems and software Modelling research group (AnSyMo). Serge Demeyer investigating themes such as refactoring, reverse engineering, and mining software archives. His main research interest concerns reengineering in an object-oriented context, but for historical reasons he maintains a heavy interest in hypermedia systems as well. He wrote a book titled "Object-Oriented Reengineering Patterns" and edited a book on "Software Evolution".