Accepted for publication at XP2001, Cagliari, Italy

Diagnosing evolution in test-infected code

Christian Wege
University of Tübingen &
DaimlerChrysler AG
HPC 0516
70546 Stuttgart, Germany
+49 711 17 92952
wege@acm.org

Martin Lippert
University of Hamburg &
Apcon Workplace Solutions Company
Vogt-Kölln-Straße 30
22527 Hamburg, Germany
+49 40 42883 2306
lippert@acm.org

ABSTRACT
In this study we trace the effects of applying the techniques of *refactoring* and *aggressive unit testing* in source code based on historical information. We show how their impact on the evolution of the architecture can be testified. The study comprises the analysis of a large number of individual integration versions of a large framework. The method described here can help development teams find weaknesses in their application of the two traced techniques.

Keywords
Software evolution, architecture evolution, extreme programming, refactoring, aggressive unit testing, software metrics

1 INTRODUCTION

In a world of constantly changing requirements, systems development must ensure that it is able to quickly respond to changed user requirements or technology updates. One major promise of Extreme Programming(XP) [2] is to enable the construction of an evolvable system. Instead of planning all possible future enhancements from the very beginning an extreme programmer relies on its ability to incorporate changes to the system and its architecture at an arbitrary point in the future.

In this study we investigated the artifacts (namely the source code and other historical information) of a project which uses the aggressive unit testing and refactoring techniques extensively for the development. We trace the effects of the application of these two techniques in the developed source code. Our system under investigation is JWAM[1] – a framework for constructing large scale interactive software systems.

In their well-known article Beck and Gamma introduce the testing style of test infection [1]. For every class in the system you write a unit test. New requirements are implemented in the system by refactoring the unit tests first and then the system classes [4]. So when those two techniques are applied strictly we talk about test infected code. Given this definition JWAM is test-infected. Its test suite created with the Java testing framework provided by Beck and Gamma[2].

The JWAM development relies on an integration server [7] which ensures that for every update of the source code all tests still run. The study is based on 254 individual integration versions of the framework which stem from this continuous integration process. In addition to the source code we used the integration log which contains a small description for every update of the source tree.

Lippert et al. state that "With Pair Programming we have improved framework quality, with test cases we maintain it. Without the test cases a lot of the refactoring we did in the past would have been less smooth."[8] With the help of our analysis we validated this rather intuitive statement by observing the history of specific system properties in the produced artifacts. As well with the help of our analysis we can point out some areas of potential improvements of the framework development.

This study concentrates on tracing the effects of agressive unit testing and refactoring directly in the code and in historical information. It doesn't investigate the correlation to requirements changes of defect rates, which would be of high interest as well.

2 THE CASE STUDY

The method

The method used in our study is an adaptation of the approach proposed by Mattsson and Bosch [9] for observing software evolution in object-oriented frameworks. Based on historical information about the subsystems, modules and classes they investigated the size, change rate and growth rate of the system. The work of Mattsson and Bosch is based on a method proposed in [5] which focus was on observing the macro-level software evolution using the version numbering of a system. Mattsson and Bosch adapted this approach for investigating

[1] http://www.jwam.org

[2] http://www.junit.org

object-oriented frameworks. The system was divided into a number of subsystems which were themselves divided in several modules. In the adapted approach each module consisted of several classes (instead of programs as in the original approach).

Size is calculated by the number of classes in each module or subsystem. The calculations of change and growth rate are made in terms of changed classes as units. Class change is measured in terms of the change in the number of public methods for each class. The focus on public methods stems from the fact that a change in the public methods reflects a better understanding of the boundary of the system. Changes of private methods however mainly reflect refinements of implementation details and are thus of minor interest.

The method steps in the original approach are[3]:

1. calculate, for all releases, the change and growth rate for the whole system,
2. calculate, for all releases, the change and growth rate for each of the subsystems,
3. for those subsystems that exhibit high growth and change rates calculate, for all releases, the change and growth rates for the modules,
4. those modules that exhibit high change and growth rates are identified as likely candidates for restructuring [9].

For our study *we based our calculations for system, subsystems and modules on the package/subpackage structure of Java*. The packaging feature of Java is a natural structuring mechanism provided by the language. In JWAM this mechanism is used to distinguish between the core and several non-core part of the whole system and inside the core to distinguish between the framework layers. We will discuss this in more detail later. Java interfaces are treated exactly the same way as Java classes.

The second important adaptation is *that we changed the top-down approach to a bottom-up approach*. Instead of starting with the top level system, we calculate the values for every class and subsystem and go up to the top. We try to trace the development method in the code, therefor we are interested in all developed artifacts. For being able to give advice on possible restructuring candidates (like in the approach by Mattsson and Bosch) we have to widen the empirical base first.

The third and most important adaptation is the *introduction of the test coverage rate*. If aggressive unit testing is one central part of test-infected programming then the results should be dependent on the number of system classes covered by unit tests.

[3] see section 3 for our modifications

The investigated system

JWAM is a Java framework supporting the development of large scale interactive software systems according to the tools & materials approach [11]. The foundation of the JWAM framework was laid in 1997 by research assistants and students of the Software Engineering Group at the University of Hamburg [8]. In 1998 the commercialization of the framework began. In 1999 the team started to use XP techniques. Our study covers 254 individual integration versions of the whole system from April 2000 to December 2000 with roughly one version per day.

The top-level package structure of JWAM 1.5.0 differentiates between the framework core and several collections of other components:

- `de.jwam`: The framework core contains the interfaces and classes which are necessary to create a simple application according to the tools and materials approach.
- `de.jwamx`: JWAM components which provide technical or domain oriented services.
- `de.jwamy`: Third party components which provide technical or domain oriented services.
- `de.jwamdev`: Tools used for the work with the framework.
- `de.jwamalpha`: New JWAM components and new JWAM tools[4].

The framework core in `de.jwam` is divided into several layers to separate different concerns. This is the most fundamental part for building new applications on top of JWAM and is the ground work for the architecture of applications based on JWAM.

3 DIAGNOSING EVOLUTION AND TEST INFECTION

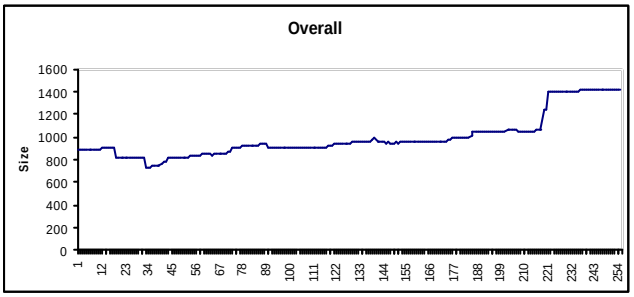
Based on the structural observations derived from the 254 integration version we are able make statements about the system's evolution and about the influence of the used techniques during development on this evolution. We have extracted information about the following subset of system properties, which is an adaptation of the descriptions found in [9]:

- The *size* of each package and subpackage is the number of classes it contains. Only top-level classes (no inner or nested classes) are used because they reflect the behavior of the system for the outside world.
- The *change rate* is the percentage of classes in a particular package that changed from one version to the next. To compute the change rate two versions of a class are needed. The relative number of the changed classes represents the change rate.

[4] Adapted from program documentation of JWAM 1.5.0

- The *growth rate* is defined as the percentage of classes in a particular package, which have been added (or deleted) from one version to the next. To compute the growth rate, two versions are compared and the numbers of the added and deleted classes are computed. The relative number of the new classes (i.e. the difference between added and removed classes) represent the growth rate.
- The *test coverage rate* is the percentage of classes that are covered by test classes. Given the convention to name a test class with the appendix "Test" we can count the number of test classes for a given package. The test coverage rate is the number of test classes divided by the number of system classes in the package subtree (i.e. the number of classes without the test classes).

One important detail for calculating the system properties is the way we deal with package restructurings. Given a class in version *n* of the system we first look for that class in version *n-1* with exactly the same package qualifier. If this class is not found we look for the class in version *n-1* without the package qualifier. Due to the nature of the development method of making small iterations and increments we are likely to find those classes that are only moved to another package but not renamed. Now we can identify the predecessor of a given package in version *n-1* by looking for the packages from which the classes in package of version *n* originate.



System observations

Diagram 1 shows the historical development of the size of JWAM for the 254 observed integration versions. Around version 19 you see an irregularity of shrinking framework size. Here a library which once has been part of the framework was deleted completely. At integration version 33 another outdated library was deleted. The other exceptionally high change in the system size is at integration version 219 where a large number of old test cases and old examples are integrated at once. Except for those singularities a more or less linear growth of the framework's size can be testified over the observed period.

Mattsson and Bosch see a linearly growing size as a sign for the maturity of a framework. The change rate and growth rate of such a system should more or less linearly fall. They explain non-linear behavior of the change rate of the overall system with a major architectural change (i.e. the introduction of online capabilities into a batch-oriented system). In case of JWAM the non-linear change rate and growth rate curves stem from the fact that we observed every integration version of the system – not only the released versions. Thus in the context of investigating test-infected code we are more interested in the frequency of the change and growth rate peeks rather than in their absolute height.

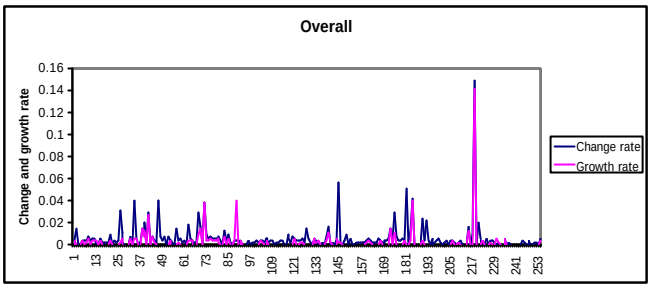
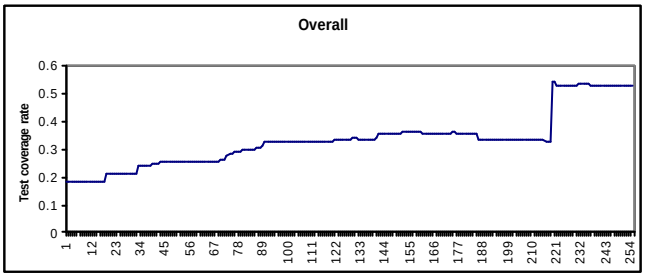


Diagram 2 shows the change and growth rate history for the whole system. Compared to the size history this diagram shows more the individual development steps (the dynamics of the development process). A change rate average of 0.005[5] testifies the iterative development in small steps. A growth rate average of 0.002[6] testifies the incremental[7] development in small steps. A general observation is that a change occurs more often than a growth of the system. This indicates that after or before the addition or deletion of new methods or new classes some refactoring steps are performed. This matches perfectly the test-infected development style.

In this study we introduced the system property *test coverage rate*. This can be seen in diagram 3 for the overall



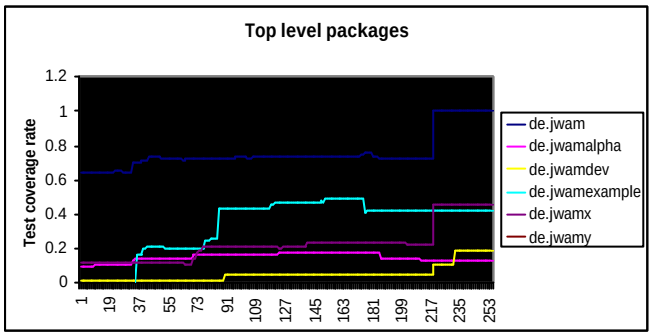
[5] A change rate of 0.005 means that 5 classes are changed given a average overall size of the system of 1000 classes.

[6] A growth rate of 0.002 means that 2 out of 1000 methods are added or deleted.

[7] Cockburn distinguishes between incremental and iterative development. "Incremental development is a staging strategy in which portions of the system are developed at different times or rates, and integrated as they are ready. [...] Iterative development is a rework scheduling strategy in which time is set aside to revise and improve parts of a system." [3]

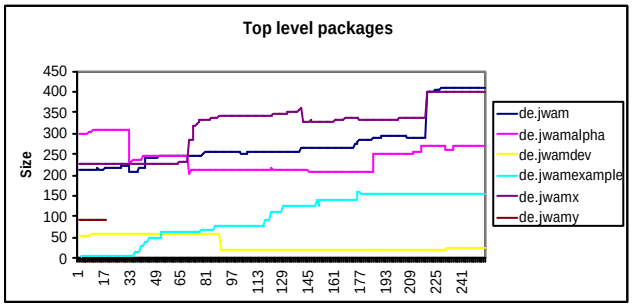
system. Except from the singularities explained in the discussion of the size history for the overall system the test coverage rate history exhibits a more or less constant growth. For example starting with a test coverage rate of less than 0.2[8] it reaches more that 0.5 at the end of the observed period. This is an indication for the growing matureness of the application of the techniques for the framework development.

A test coverage rate of about 0.5 doesn't seem to be very sophisticated for a development process which states to be a form of XP as XP requires a very high test coverage rate to be successful. As can be seen in diagram 4 we have to differentiate between the different top-level packages in order to make a more qualified diagnosis. This diagram shows the test coverage rate for the individual top-level packages.

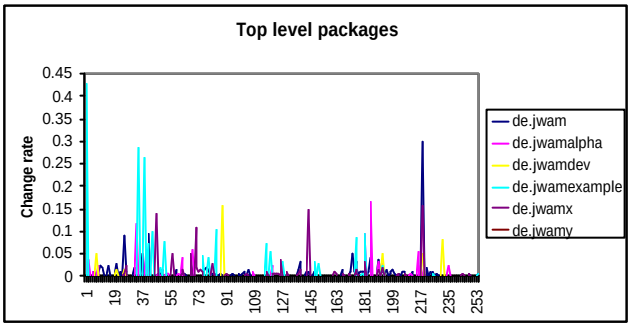


The framework core in `de.jwam` exhibits a more or less healthy growth of the test coverage rate from below 0.7 to 1. This fits to the close to ideal size history for the framework core in diagram 5. For the `de.jwamexample` package the development team seems to have realized the importance of test cases for the examples during the framework development.

Starting from a test coverage rate of 0 it ends at 0.4. At the same time the package `de.jwamexample` is the only other top-level package which exposes a close to linear growing size in diagram 5. The development team seems to have understood the value of a suite of examples and the importance to ensure their high quality. An application developer could base its own development on the provided examples together with the associated test cases.



The developers seem to have realized the importance of a test suite for the `de.jwamx` package (containing additional components on top of the framework core) which exhibits a growing number of test cases. But the number of test cases still didn't reach a level which could ensure a healthy behavior of the size history curve for this package as can be seen in diagram 4. The packages `de.jwamalpha` and `de.jwamdev` have by far the worst test coverage rate history. The developers obviously don't see the need to put the same amount of effort in the evolvability of their development tools as they did for the rest of the framework. The package `de.jwamalpha` is planned to be a test area for new ideas. In XP terms these new ideas are spike solutions which do not have to be developed with the same care as the rest of the system. A development of a proper test suite for the new ideas is deferred to the point in time when those ideas are incorporated into the base system. This is completely valid for XP and does not exhibit a fallacy in the development process. Diagram 5 confirms this behavior for the size history of the two packages.



The differences between the top-level packages can also be traced in the change rate history in diagram 6. For the framework core the development seems to be very close to the ideal: many small peeks with a high frequency indicate many iterative steps. Whereas in the case of the other top-level packages the change rate peeks are less regular. Here the development is performed in fewer and bigger steps.

Diagnosis and Recommendations

Generally the development team seems to be on the right track for applying the two techniques. The analysis of the code shows the positive effects of building and maintaining a good test suite on the evolution of the system. The iterative and incremental development steps are most clearly seen in the parts of the system which are the most mature ones. The effort distribution of the development resources seems to be effective in the sense that a complete test suite is only maintained for those parts that have to be of high quality (i.e. the framework core). The framework

[8] A test coverage rate of 0.2 means that only 2 out of 10 system classes are covered by a test class.

parts which are in an experimental state (i.e. the alpha package) have a poor test coverage rate, a non-linear size history and a few big change and growth rate peeks (as opposed to many, equally distributed small peeks in the case of the framework core).

It is a good idea to provide framework users with a set of examples which come with a whole suite of tests specific to those examples. This shows the users how to write tests for typical uses of the framework and helps improve the overall quality.

However the analysis also exhibits some possible weaknesses of the system development. The size history of the overall system shows some non-linearity. This seems to indicate large steps and big changes in the development. That would conclude that these steps did not happened in an XP like style. But the current used system demonstrates that the used method in this article has also its weaknesses. The mentioned changes influenced a lot of classes but they were no "big" changes in the sense of XP. The changes were done in a few minutes, maybe one hour, and did not influence many other parts of the system. So we would say they were not big or complicated changes. This "quality of the change" is not measured by the used measurement method.

4 RELATED AND FUTURE WORK

Clearly the work of Mattsson and Bosch is the basis for our approach of identifying the software evolution through examination of historical information. We extended their approach in some ways to fit the specific needs of our research question. Mattsson and Bosch extended the original approach of Gall et al. in the sense of smaller granularity of the examined entities [5]. Our method extends the approach of Mattsson and Bosch in the sense that we examine the software evolution over smaller periods of time to better fit the incremental and iterative development in small steps.

An empirical study by Lindvall and Sandahl show that software developers are not so good at predicting from the requirements specification how many and which classes will be changed [6]. In the context of XP the idea of a requirements document is omitted completely in favor of user stories which contain only the next most important requirement for the evolution of the system [2].

Simon and Steinbrückner have analyzed JWAM 1.5 with their high quality metrics tool. They are working on an analysis of a more recent version of JWAM to see how their first recommendations on the quality of the framework found their way in the version [10].

The analysis concentrated on public methods given by the method we base our work on. Experience however shows that refactoring is applied to private methods in many cases. On the other hand our analysis didn't take into account the correlation to requirements changes and defect rates, which could reveal other insights in the development process.

5 CONCLUSIONS

In this study we presented an approach for tracing the effects of the techniques refactoring and aggressive unit testing in the code. We examined 254 integration versions of a large Java framework. The integration versions stem from a continuous integration process which enables an XP-like development of the framework.

We showed the usefulness of our approach and discussed how the effects of "test-infected development" can be seen in the history of specific system properties. As well we were able to highlight some areas of potential improvements in the development process.

REFERENCES

1. Kent Beck and Erich Gamma, Test-infected: Programmers love writing tests. *Java Report*, Vol 3, No. 7, 1998.
2. Kent Beck: *Extreme Programming Explained*. Addison-Wesley, 2000.
3. Alistair Cockburn: *Surviving Object-Oriented Projects: A Manager's Guide.* Addison-Wesley, 1998.
4. Martin Fowler: *Refactoring: Improving The Design Of Existing Code*. Addison-Wesley, 1999.
5. H. Gall, M. Jazayeri, R.G. Klösch, G. Trausmuth, Software Evolution Observations Based on Product Release History, *Proceedings of the Conference on Software Maintenance - 1997*, 1997.
6. M. Lindvall, K. Sandahl: How Well do Experienced SoftwareDevelopers Predict Software Change? *Journal of Systems and Software*, 43(1), 1998.
7. M. Lippert, S. Roock, R. Tunkel, H. Wolf: Stabilizing the XP Process Using Specialized Tools. To appear in *Proceedings of XP 2001 conference*, Cagliari, Sardinia, Italy, 2001.
8. M. Lippert, S. Roock, H. Wolf, H. Züllighoven: JWAM and XP - Using XP for framework development. *Proceedings of the XP2000 conference*. Cagliari, Sardinia, Italy, 2000.
9. Michael Mattsson and Jan Bosch: Observations on the Evolution of an Industrial OO Framework. In *Proceedings of the ICSM'99*, International Conference on Software Maintenance, Oxford, UK, 1999.
10. Simon, Steinbrückner: Analysis of JWAM 1.5 with the metrics tool Crocodile. Online at http://www.jwam.de/home/quality_assessment_jwam15.pdf.
11. Heinz Züllighoven: Das objektorientierte Konstruktionshandbuch. dpunkt.verlag, 1998.