When Agile Meets OO Testing – A Case Study

Siegfried Goeschl
Senior Software Engineer
Spengergasse 25A/2/16
A-1050 Vienna, Austria
+43-664-19420090
sgoeschl@gmx.at

Matthias Herp
Senior Software Engineer
Wernersdorf 2
A-3110 Wernersdorf/Neidling
+43-676-3581488
matthias.herp@gmx.at

Clemens Wais
Test Manager
Schmalzhofgasse 10/8
A-1060 Vienna, Austria
+43-664-3312838
clemens.wais@gmx.at

ABSTRACT
This paper describes the testing approach for an object-oriented rich client application based on an agile software development process. The paper starts with an overview of project and the testing strategy being used. It then goes on to describe the test tools used in the project and the results achieved. The paper ends with a discussion of the discovered defects, their distribution and improvements for the testing process.

Categories and Subject Descriptors
D.2.9 [**Software Engineering**]: Management – *pair programming, software process models, software quality assurance.*

General Terms
Management, Measurement, Verification

Keywords
Agile Testing, Object-oriented Testing, Unit Testing, Integration Testing, Continuous Integration, Defect Rates, Technical Debts

1. INTRODUCTION
When Scrum [1] was introduced at OOPSLA 1995, it did not only change the way to run a software development project but also had major impact on how to test it. Life under the waterfall required a project test plan, which called for a test phase of a few months and a separate test team. It would come with a detailed capacity planning for the testers, a fine-grained project plan, and a matrix with all the quality attributes and the effort we should take to test every attribute. This traditional approach to software testing looks less promising in the agile world where everything is centered on volatile requirements, sprints, standup meetings, features, retrospectives and shipping software after four weeks. Welcome to world of "agile testing".

To shed some light on how "agile testing" works, the XYZ case study is introduced - a medium size customer information system used by 300 in-house employees. The XYZ front-end is written in Java 1.5 and based on Eclipse RCP (Rich Client Platform). The XYZ front-end communicates with the XYZ server and various other back-office systems using SOAP (Simple Access Object Protocol) over JMS (Java Message Service). From a testing point of view, it is a client/server system with some stringent requirements regarding availability and response time.

The Scrum development approach is used for the project with sprint durations of four weeks and a strong focus on pair programming. The Scrum team is rather large consisting of 8-10 software developers and 2-3 testers all co-located in a single room to foster team communication. As of writing the project is now in its 25th iteration and has lasted for about two years.

2. THE TESTING STRATEGY
Coming from a more traditional testing background, one might ask, "What is the difference between agile and chaotic testing"? The answer lies in the agile testing strategy being applied, so it is time to have a closer look at testing the XYZ project.

2.1 Focus on Individuals and Interactions
Following the "Manifesto for Agile Software Development" [2] and its "individuals and interactions over processes and tools", extensive pair programming and close interaction with the testers are used. Once a software developer pair has finished its work, an informal and face-to-face review takes place together with a tester and/or business analyst to ensure that the required functionality has been implemented. Any problems found during this informal review are fixed immediately without creating a bug report and only then is the code declared ready for QA. Following these rules caters for continuous interaction, which in turn improves team productivity [3].

2.2 Test Automation
The test automation approach for the XYZ project follows the test automation pyramid [4] having three different layers of automated tests. These three levels have different characteristics regarding costs, execution speed and return of investment.

The lowest tier consists of unit and component tests based on JUnit 4 and making extensive usage of Mockito, a mocking framework. The mock testing framework allows running component tests without depending on live back-office systems, thereby making the tests more stable and dependable. The JUnit tests are easily written and maintained by software developers due to the excellent refactoring support of current IDE's. As many tests as possible are pushed to this level of test automation since these tests provide quick feedback and good return of investment. The software developers are required to pass those tests before committing any changes to the source control system.

ECOOP '2010 Maribor, Slovenia EU

The middle tier of the test automation pyramid consists of regression test that exercise the high-level business functionality and the back-office services. The business services and back-office services are tested using JUnit. These tests operate at the API level ("behind the GUI") thereby bypassing the presentation layer. Bypassing the presentation layer makes the tests less expensive to write and maintain than tests that depend directly on the user interface. Running the test of the middle tier is much slower because each test covers more ground than a unit test and accesses the database and/or back-office systems.

The top tier operates on the presentation layer and represents what should be the smallest investment in automation effort. This is because tests, which operate on the presentation layer, are brittle and expensive to maintain. These tests are based on an in-house GUI testing framework using low-level SWT (Standard Widget Toolkit) events to exercise the user interface.

No matter how far test automation goes, most systems also need manual testing activities such as manual regression tests, exploratory tests, user acceptance tests and performance tests. The XYZ project is no exception to this rule and most user stories are tested manually.

2.3 Load and Stress Testing

Regular load tests are mandatory to ensure that the project's performance requirements are met. For load and stress testing Apache JMeter (http://jakarta.apache.org/jmeter/) is used since it is open source, can be easily extended and supports testing without a graphical user interface.

3. LOOKING AT NUMBERS

The XYZ project is a real-life software project and looking at such a project raises a few interesting questions

- How much effort was spent for creating automated tests?
- How effective is the testing strategy being used?
- Are there any statistical data regarding defect density and distribution?

Following Tom DeMarco's advice of “You can’t control what you can't measure" some code metrics are measured as shown below in Table-1.

Table 1. Project Metrics

Metric	Value
Overall Code (NCSS[1])	307.992
Production Code (NCSS)	227.795
Test Code (NCSS)	80.197
Overall Nr. of Classes	7.042
Nr. of JUnit Test	3.941
Test Statement Coverage %	43

It can be concluded that 26% of the coding effort went into the regression test suite, which currently consists of 3.941 manually written tests. At first sight the number of regression test cases look impressive but the test coverage needs to be improved to support continuous refactoring. To put it differently - the regression tests are the foundation for fearless refactoring [5].

How effective is the agile testing strategy for the XYZ project? During the project a few qualities of test effectiveness as described in [6] were measured, such as Defect Detection Percentage (DDP), Defect Fix Percentage (DFP) and Defect Density.

The first step of measuring test effectiveness consists of recording the total number of bugs and determining in which testing phase they were detected. The following test phases are defined in the bug-tracking tool

- QA Phase
- Acceptance Test Phase
- Production Phase

The diagram depicted in Figure-1 shows the number of defects per iteration classified by the different testing phases. The introduction of billing-related functionality (consisting of new graphical user interface dialogs and new back-office functionality) caused the distinct spike for the eleventh iteration.

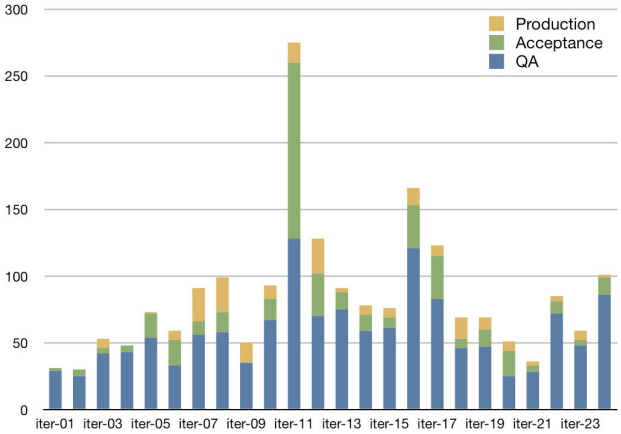

Figure 1. Defects per Iteration

Considering the numbers of Figure 1 the Defect Detection Percentage (DDP) can be calculated as shown in Table-2 below

Table 2. Defect Detection Percentage

	Found	Remaining	DDP(%)
QA	1396	643	69
Acceptance Test	425	218	66
Production	218	N.A.	N.A.

Please note that defects found during the development phase are not tracked - these are the defects found by the developer's unit tests and during the first informal review before declaring a feature ready for testing. Tracking the defect detection over three different phases only has a pragmatic reason. Defects found during the QA Phase can be considered as more or less "good" defects since they are confined to the project room (test and development team share a common office) and are relatively

[1] Non-Comment Source Statements

cheap to fix. Defects found during the acceptance test phase are more expensive and cause management attention, which is to be avoided. Finally, there are the defects encountered during production - these are the really bad defects affecting the customers and causing much management attention. When the number of production defects exceeds the internal estimation, it will affect the next project iteration - developers are busy correcting high priority production defects and attending status meetings. Such pragmatic reasoning leads to next test effectiveness measurement - the Defect Fix Percentage (DFP) per iteration as shown in Figure-2.

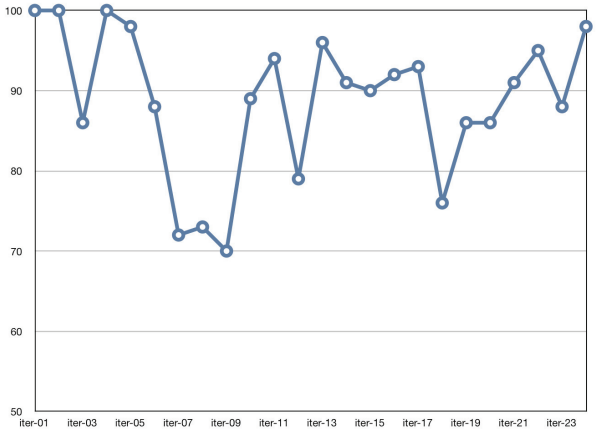

Figure 2. Defect Fix Percentage

Looking at Figure 2 it can be concluded that 11% of the defects are discovered during the production phase. Not all production bugs have the same impact therefore the project team is most concerned about the high priority production bugs and their historical trend over the project iterations as shown in Figure-3

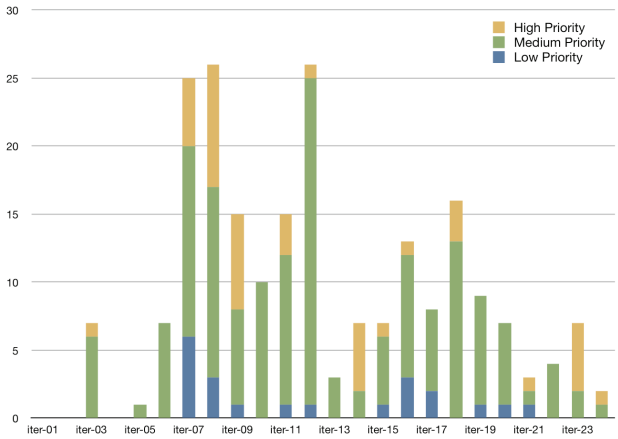

Figure 3. Production Defects

The numbers in Figure-3 show that 20% of production defects were critical. For the production defects of the last twelve months (spanning nine iterations and eight bug fix releases) a root cause analysis was performed as shown in Table-3.

Table 3. Delivered Defects Root Cause Analysis

	Low	Medium	High	All
Requirements	1	19	4	24
Design	0	3	1	4
Implementation	4	17	7	28
Other	1	12	0	13
All	6	51	12	69

The category "Other" consists of root causes such as configuration, rollout, operational and production data quality issues.

4. GOOD ENOUGH TESTING

An agile project is centered on short iterations in which each iteration can be viewed as a small project of its own. A software developer usually knows the requirements for the current iteration when working on a component. This component will be extended or rewritten to fulfill the requirements of a future iteration. Therefore, one characteristic of an agile project is the high degree of rework that mandates a comprehensive set of regression tests. This leads to next question, "Is the testing good enough for XYZ?"

The following quote clarifies "good enough testing": "Good enough testing is the process of developing a sufficient assessment of quality, at a reasonable cost, to enable wise and timely decisions to be made concerning the product." [7].

One particular quality assessment was centered on the question of whether high quality software can be delivered despite using an agile development and testing approach. This question can be answered by comparing the defect rate with published case studies [8][9] as shown in Table-4

Table 4. Overview of Delivered Defect Rates per Function Point (for projects nominally 1.000 FPs in size)

SEI CMM Levels	Delivered Defects per FP	Delivered Defects per 1.000 NCSS
SEI CMM 1	1,00	18,9
SEI CMM 2	0,40	7,5
SEI CMM 3	0,15	2,8
SEI CMM 4	0,08	1,5
SEI CMM 5	0,01	0,2

The production code consists of approximately 228.000 NCSS (Non-Comment Source Statements) and 218 defects in the released product. The comparison of defect rates across different project size metrics, application domains and software engineering tools is problematic and only allows for a ballpark estimate of software quality.

Assuming that 228.000 source statements represent roughly 4.000 IPFUG function points (using a conversion rate of 53 Java statements per function point as described in [8]) the following defect density can be estimated

- 0,06 delivered defects per FP
- 0,96 delivered defects per 1.000 NCSS

It can be concluded that the XYZ project delivers an excellent software quality comparable to SEI CMM Level 4, which is considered good enough by Product Owner and customer.

5. THE ROAD AHEAD

Every real world project has encountered problems and will face challenges, and again the XYZ project is no exception to this rule. During the retrospectives a few improvement opportunities were identified

- Decreasing the technical debts
- Increase of the declining test coverage
- Evaluating a replacement for the in-house SWT testing framework
- Evaluating a commercial GUI testing tool

Looking at the root causes of delivered defects as shown in Table-3, it can be concluded that implementation issues are responsible for more than 40% of the defects. An increasing proportion of these implementation issues are caused by technical debts [10] such as code inspection warnings or "TODO" sprinkled all over the source code. For the last three weeks additional effort has been made to decrease these technical debts as depicted in Figure-4.

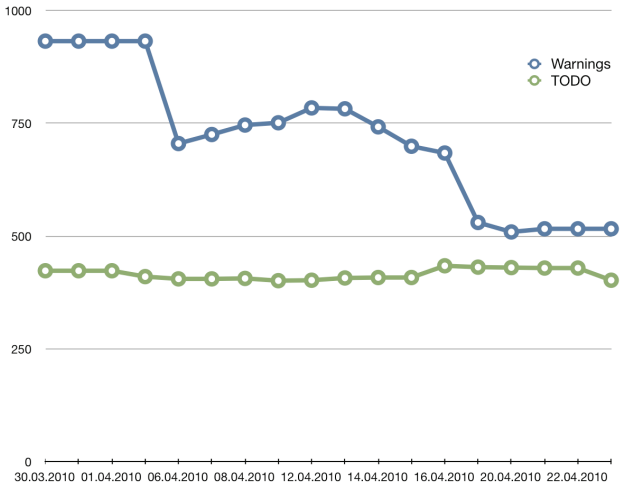

Figure 4. Decreasing Technical Debts

The code warnings can be fixed quickly, but on the average it takes two to three hours to remove a "TODO". Therefore it can be estimated that removing all these issues will take somewhere between 800 to 1200 hours of development time.

Over the last iterations the number of unit/component regression tests remained constant and the test coverage declined slowly, thereby causing quality problems. On the one hand, the test coverage was not actively tracked, on the other hand writing GUI tests using the in-house SWT testing framework was mostly discontinued. The open source tool "SWTBot" (see http://www.eclipse.org/swtbot) will be evaluated as a replacement for the in-house SWT testing framework.

The majority of GUI regression tests are currently executed manually, which will upset the time-boxed release plans when the code base continues to grow. The test team is currently evaluating commercial products but most of the commercial offerings seem to integrate poorly into an agile environment for the following reasons:

- Vendor-specific languages make coding difficult, are hard to learn and interfere with the collaboration between testers and programmers [11]
- Central test repositories make integration with source control difficult
- The tools are hard to use with continuous integration due to the lack of APIs
- Some tools are confined to a particular operating system, e.g. Microsoft Windows
- Restrictive and expensive licensing prohibits every developer on the team from being able to run the test scripts

6. CONCLUSION

While agile software development and testing is not a "one size fits all" approach [12], it can deliver excellent software quality for small to medium sized projects. For the XYZ project this is achieved by following a set of agile practices such as

- Extensive pair programming serving as an informal review process
- Test automation to allow refactoring and addition of new software features
- Continuous integration to keep the code base working
- Retrospectives to improve the development and testing process after each iteration
- Individuals and interactions over processes and tools to minimize bureaucracy between testers and developers

While none of these practices are unknown or difficult to follow (apart from extensive pair programming) they work best in combination. Following these agile practices, the XYZ project achieved a defect density of 0,96 delivered defects per 1.000 source statements thereby achieving a software quality comparable to SEI CMM Level 4. All in all, the agile software development approach and the XYZ project are considered a success story within the company, and therefore XYZ is acting as a blueprint for future projects.

7. REFERENCES

[1] Schwaber, K. and Beedle, M. 2001. Agile Software Development with Scrum. Prentice Hall.

[2] Beck, K., et al. 2001. Manifesto for Agile Software Development, http://agilemanifesto.org/.

[3] Talby, D., Keren, A., Hazzan, O. and Dubinsky, Y. 2006. Agile Software Testing in a Large-Scale Project. IEEE Software. 30-37.

[4] Crispin, L. and Gregory, J. 2009. Agile Testing - A Practical Guide for Testers and Agile Teams. Addison-Wesley. 276-279.

[5] Jeffries, R. and Melnik, G. 2007. 'Guest Editors' Introduction: TDD - The Art of Fearless Programming. IEEE Software. Vol. 24. No. 3. 24-30.

[6] Fewster, M. and Graham, D. 1999. Software Test Automation. Addison-Wesley. 211-219.

[7] Bach, J. 1998. A Framework for Good Enough Testing. Computer 31(10). 124-126.

[8] Jones, C. 2010. Software Engineering Best Practices. McGraw-Hill, 558-585.

[9] Jones, C. 2008. Software Quality in 2008: A Survey of the State of the Art, http://issuu.com/meade/docs/qualitysurvey2008

[10] Cunningham, W. 1992. The WyCash Portfolio Management System. ACM SIGPLAN OOPS Messenger. Volume 4. Issue 2. 29-30

[11] Kaner, C., Bach, J. and Pettichord, B. 2002. Lessons learned in Software Testing. John Wiley & Sons. 117-118.

[12] Boehm, B. and Turner, R. 2003. Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley Longman Publishing. 55–57