Markus Lindqvist

Integrating Unit Testing Processes into Large Legacy Software System Development Process

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

1 April 2014

Tekijä(t) Otsikko Sivumäärä Aika	Markus Lindqvist Yksikkötestauksen käyttöönotto laajan vanhan ohjelmistotuotteen tuotekehitysprosessin osaksi 49 sivua + 6 liitettä 1.4.2014
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikan koulutusohjelma
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Opettaja Sami Sainio Ohjelmistoarkkitehti Mikael Lavi

Tässä esitetyn tutkimuksen tarkoituksena oli arvioida millaisia hyötyjä yksikkötestauskäytäntöjen käyttöönotolla saavutettaisiin vanhan ohjelmistotuotteen tuotekehityksessä.

Yksikkötestauksen ja yksikkötestattavuuden vaikutuksia tuote X:n arvioitiin kuuden vuoden ajalta, vuodesta 2007 vuoteen 2013. Tutkimus osoitti, että yksikkötestaukselle soveltuvan ympäristön luominen ohjelmistokehittäjille voi viedä vuosia kun kyseessä on monimutkainen tuote.

Ensimmäisenä parannuksena yksikkötestaustyökalut liitettiin tuotteeseen ja jatkuvaan integraatioon (continuous integration). Jatkuvan tarkkailun (continuous inspection) raportit luotiin kehityksen seuraamiseksi. Kävi selväksi, että kunnon työkalut ja infrastruktuurin ylläpito ovat oleellinen osa tavoitteisiin pääsemistä. Kehittäjien kouluttamiseksi järjestettiin koulutustilaisuuksia ja järjestelmät dokumentoitiin.

Tuotteen arkkitehtuurin järjestelmällinen kehitys oli oleellinen osa testattavuuden luomista. Monoliittinen tuote pitää modularisoida ja refaktoroida sellaiseksi jossa komponenttien testaaminen on mielekästä. Tutkimuksen tulokset osoittavat että tuotteen modularisointi oleellisesti paransi mahdollisuuksia yksikkötestaukseen.

Kehitystyö yksikkötestauksen osalta tehostuu esimerkiksi virheiden paikallistamisen osalta kun työskennellään pienempien komponenttien kanssa. Pienempien kokonaisuuksien mittaamisesta seuraa näkyviä tuloksia. Kohtuullisen kokoisissa komponenteissa yksikkötestikattavuuden kasvu näkyy ja motivoi ohjelmistokehittäjiä jatkamaan käytäntöä.

Lopputuloksena voidaan todeta, että yksikkötestauskäytännöt on mahdollista sisällyttää haasteellisen tuotteen tuotekehitysprosessiin jos tahtotila siihen löytyy. Hyvä infrastruktuuri mahdollistaa ohjelmistokehittäjien työn, vähitellen lisäten yksikkötestauksen käyttöä ja testien kattavuutta, josta seuraa tuotteen laadun parantuminen.

Avainsanat	yksikkötestaus, arkkitehtuurikehitys, modularisointi

Metropolia

Author(s) Title Number of Pages Date	Markus Lindqvist Integrating unit testing processes into large legacy software system development process 49 pages + 6 appendices 1 April 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Sami Sainio, Lecturer Mikael Lavi, Software Architect

The purpose of this study was to analyse the improvements of software development processes of a legacy software product after introducing unit testing methodologies.

The results and effects on unit testing process and unit testability of Product X were analysed for a period of six years, from late 2007 to 2013. The study showed that setting up a unit-testing-friendly environment for software engineers can take years of time in a challenging product environment.

Improvements started by integrating a unit testing framework into the product and setting up a continuous integration system to utilise it. A continuous inspection dashboard was introduced to follow the progress. It was seen that efficient tooling is a key to success, and that the new infrastructure needs to be well maintained. Educating the developers on all this required organizing trainings and documenting the systems.

Systematic development of the product architecture was a crucial part of making the system unit testable. The monolithic product had to be modularised and refactored to achieve layered architecture where testing of components is meaningful. The results of the study indicate that modularising a monolithic system greatly improved development workflows relevant to unit testing.

Having smaller components increased unit testing efficiency in terms of development speed and pinpointing errors. Measurements of smaller entities for improvements gave visible results. In reasonably-sized modules, the increases in unit testing coverage were real and motivated the software engineers to continue the practice.

In conclusion, the unit testing process can be introduced in a challenging software development environment if the intent of doing it exists. Good infrastructure enables the work of engineers, resulting in incremental use of unit testing, increasing the testing coverage, which can be assumed to affect the product’s quality positively.

Keywords	unit testing, legacy software, architecture improvement, modularisation

Table of contents

1 Introduction 1

2 Unit Testing Process 2

2.1 What Are Unit Tests? 2

2.1.1 Structure of a Test Method 2

2.1.2 Test-Driven Development Process 3

2.1.3 Characteristics of Unit Tests 5

2.1.4 Unit Testing Frameworks 6

2.1.5 Measuring Unit Testing 8

2.1.6 Measuring Individual Unit Tests 9

2.1.7 Unit Testing as a Part of an Agile Process 10

2.2 Characteristics of Legacy Software Systems 12

2.2.1 Complexity 13

2.2.2 Cohesion and Coupling 14

2.2.3 Unit Testing Legacy Software Systems 16

3 Introducing Unit Testing into a Large Legacy Software System 19

3.1 Description of the Development Environment 19

3.2 Initiative to Start Unit Testing in Late 2007 20

3.2.1 Integration of CppUnit Framework 21

3.2.2 TSAR Project 2008 23

3.2.3 Software Engineers' Reactions to Unit Testing Initiatives 24

4 The PAMM Modularization Project 2010 25

4.1 PAMM Vision 25

4.2 Implementation of the PAMM Project 27

4.2.1 Challenges and Improvements in the PAMM Modularization Project 28

5 Systematic Development of Unit Testing Activities 30

5.1 Change to GoogleTest Framework 31

5.2 Continuous Integration System Improvements 32

5.3 Continuous Inspection System - SonarQube 33

6 Results of the Improvement Projects 36

6.1 Architecture Improvements 36

6.2 Increased Use of Unit Tests 37

6.3 Software Engineers' Questionnaire Results 41

7 Conclusion 44

References 46

Appendices 1

Appendix 1: Unit Testing Challenges from Workshop 2008 / Rihtniemi 1

Appendix 2: Unit Testing Challenges 2011 / Lindqvist 1

Appendix 3: Program (Implemented in F#) Used to Extract the Table of Data about Commits Which Contain Changes to Unit Tests 1

Appendix 4: Commits which Contain Changes to Unit Tests 1

Appendix 5: The Unit Testing Questionnaire Form 1

Appendix 6: The Responses to the Unit Testing Questionnaire Form 1

1 Introduction

Company A is a company developing many different enterprise computer software products in various business areas. This thesis focuses on a selected part of the largest product suite of the company, Product X, which is structural engineering software.

The software was originally written using the C programming language, and later partly converted to object-oriented C++, with extensibility interfaces for C (RPC) and .NET platforms (C#, F#, C++/CLI).

The product development unit is responsible for maintaining almost eight million lines of code, which is nearly twice as much as five years earlier. Long history and evolution from a small single-purpose software to complex multipurpose product creates new challenges for the development process. To mitigate these challenges the company has started systematic architecture and process development on many areas.

One way of ensuring a clean architecture and building high-quality software from the bottom-up is to use unit testing or test-driven development from the start. This thesis aims to document and analyse the improvements of the software development process of Company A by integrating unit testing methodologies into the existing workflows.

As a result of this study, the following questions are answered:

- What are the usual challenges specific to implementing an effective, gradually improving, unit testing process in a large legacy software?

- How have these challenges been tackled in Company A?

The research was conducted by analysing Company A’s internal documentation created during the years in various improvement projects, and by interviewing top professionals who have been working with these projects or in related areas in the company.

The effect of change is analysed over six years, from late 2007 to 2013, focusing on the core component of the software product, excluding the external applications, plugins, third-party software or application programming interfaces (API) from the analysis.

2 Unit Testing Process

2.1 What Are Unit Tests?

The Art of Unit Testing (2011) defines unit tests as follows:

> *A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.*

A difference to “ideal unit tests” described before is the fact that commonly in real-life developers test multiple assumptions in single test cases, as opposed to only asserting one condition. Combining multiple test methods into one may be necessary if test cases require expensive setup processes, which need to be reduced for performance reasons.

In real life code bases, the components are often not really independent or usable alone. This leads to unit tests testing the class functionality plus its dependencies too, either from the same module or even from different modules. Whether testing the class with dependencies (as opposed to using mocks for example) is a good or bad depends on the situation. Usually coupling with unrelated classes is considered a negative thing; however, if the functionality is related and dependencies are simple, it may be reasonable to test the classes together.

Google has solved the terminology problem by calling unit tests and unit-test-like tests “small tests” (How Google Tests Software 2011), functional tests that test integration of multiple modules “medium tests”. Integration and end-to-end tests are called “large tests”. The importance has been placed on product quality and relevance of the tests, not on naming conventions or strict ruling.

See chapter 2.1.2 about at which point in the development process the tests are written.

2.1.1 Structure of a Test Method

Structuring tests in a commonly known way is an industry proven method for easing the maintenance and readability.

A unit test method usually consists of three different parts (Kotilainen 2012, 8-10), which are called *arrange*, *act* and *assert* (Grigg 2012). The first part is used to set up test data or objects for the second part, in which the actual program logic is invoked. The third part checks that expected conditions are met, based on the results obtained from a production code call.

```
[Fact]
public void Empty_segment_appears_in_result()
{
    Recipe r = GetRecipe(4, segment);

    var s = scaler.ScaleRecipe(r, 5);

    Assert.Equal(1, s.Count);
    Assert.Equal("Example", s[0].Title);
    Assert.Equal(1, s[0].OrderNumber);
}
```

Listing 1. Example of a unit test with Arrange-Act-Assert structure (Copied from Kotilainen 2012, 8).

Testing the same method, class or function with different parameter combinations often leads to repetitive patterns in the setup code, which prepares test data or sets up the environment. Code duplication is a highly ranked code smell (even called “the number one in the stink parade” (Fowler 1999, 63)), and not good for maintainability. Code duplication is easily fixed by using the “extract method” (Fowler 1999, 89) refactoring, where common code is combined to just one place and invoked from there.

Unit testing frameworks provide a place to put the common code that is run before or after each test method; the setup and teardown methods. The framework itself takes care that these auxiliary methods are invoked at the right time, which again makes the test methods clearer.

2.1.2 Test-Driven Development Process

Normally, unit tests are written by the original developer of the production code, sometime after the production code is finished. This approach has a few problems, some of which are:

- The original developer might not anymore have a fresh idea about the intent of the production code to produce valid tests for it, making it an effort to get back into the code (Astels 2003, 5).
- The project might run out of time, in which case the tests will not be implemented at all.
- Test coverage might not be sufficient to cover all of the production code.

Test-driven development (TDD) is an advanced way of implementing unit testing for all code, before the production code is written.

The TDD process is defined (Poppendieck and Poppendieck 2009) like this

> *“Tests should be created first – before the code is written. Either the tests are derived from the specification, or better yet, the tests* are *the specification. In either case, coding is not done until the tests for that section of code are available, because then developers know what it is they are supposed to code.”*

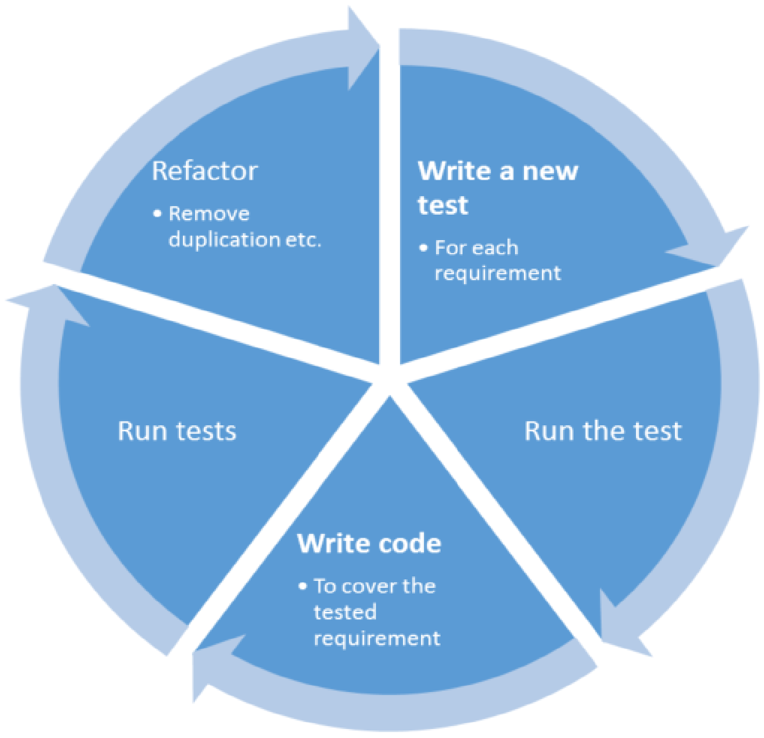


Figure 1. The Test-Driven Development process (adapted from Bit by bit blog 2012).

The challenge of implementing a true TDD process (illustrated in figure 1) is that it requires very high discipline from the developer. The aim is to write every test *before* the production code is written. However, it has been seen that often it is tempting to start experimenting and prototyping with production code, especially when requirements are unclear (Dietrich 2011).

The practice of TDD was popularized in the early 2000s by the rise of the Extreme Programming (XP) development method, which was developed by Kent Beck (Copeland 2001).

Writing tests before code is actually designing the interfaces, therefore resulting in more understandable, simpler code (Poppendieck and Poppendieck 2006, 82). Unit testability follows since the code already has the tests.

2.1.3 Characteristics of Unit Tests

Different sources define unit tests in various different ways, but some characteristics seem to be commonly mentioned by all parties:

- **Fast** (Martin 2008, 132, The Art Of Unit Testing 2011): Unit tests need to run *fast*, that is, they need to provide quick feedback for the software engineer whether a piece of code worked or not.
- **Repeatable** (Martin 2008, 132, The Art Of Unit Testing 2011): The tests need to be runnable in different environments, always producing identical results. The tests may not yield different results when they are executed in different order and they may not depend on random numbers or current time.
- **Small** (Martin 2008, 132): Test code only runs a very limited subset of code from a single module.

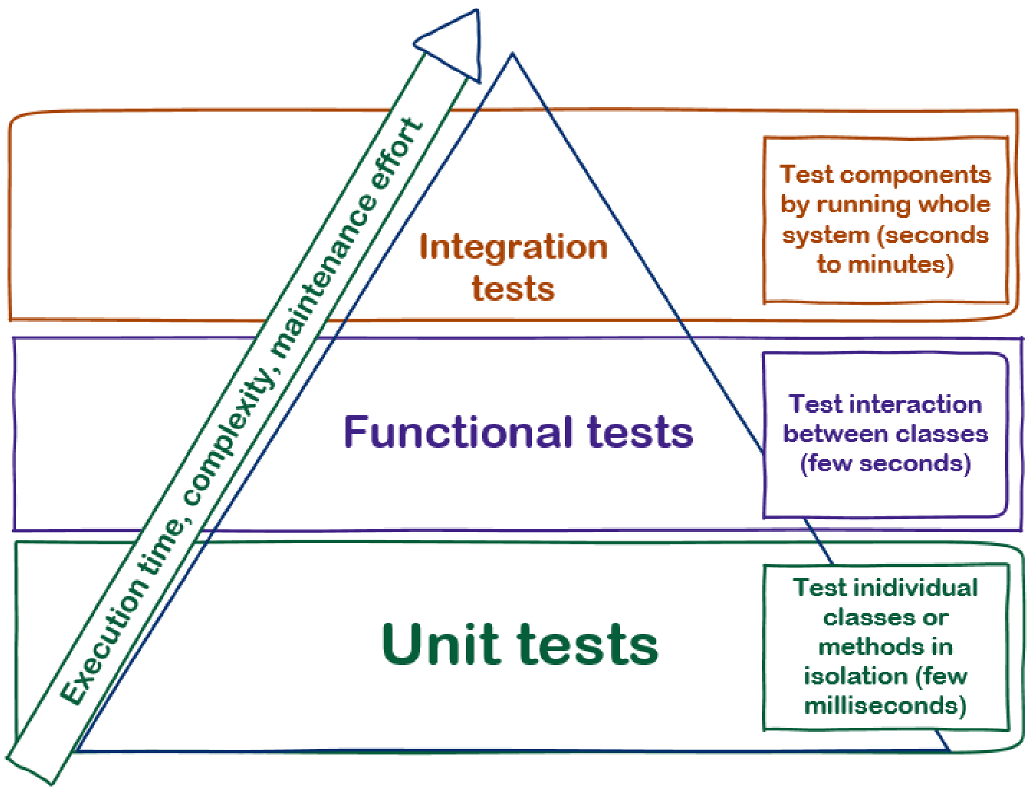

Execution time, complexity, maintenance effort
Integration tests
Test components by running whole system (seconds to minutes)
Functional tests
Test interaction between classes (few seconds)
Unit tests
Test inidividual classes or methods in isolation (few milliseconds)

Figure 2. Different Kinds of Testing (Adapted from Whittaker, Arbon and Carollo 2012, 46).

As seen in figure 2, *How Google Tests Software* (Whittaker, Arbon and Carollo 2012) illustrates different types of automated testing using a stacked pyramid. In Google's environments, software engineers write a large number of unit tests which run fast and provide the basic guarantee of correctness for the software module. Longer-running functional and integration tests, written by quality assurance engineers, only verify that smaller pieces of the puzzle work together. Because of this, the number of high-level tests can be lower. Confidence for the whole system is built by creating quality from the bottom up.

2.1.4 Unit Testing Frameworks

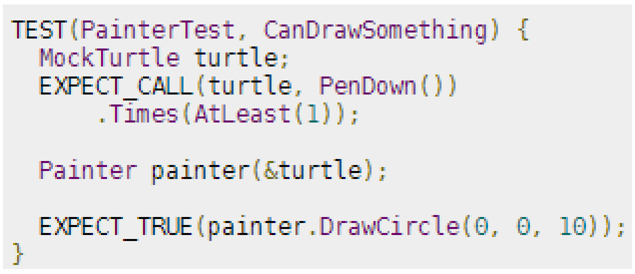
Every programming environment has their own unit testing framework with the purpose of enabling the rapid development of unit tests. Rapid unit test development in turn exists to support development; therefore, unit testing tools are not just testing tools but important development tools (Wells 1999).

The framework implements functionality common to all test methods: handling setup and teardown, execution of the tests, reporting the results, providing a collection of assertions and other helper functions. It also provides support for IDE integration and build system integration.

Usefulness of the framework decreases if usage requires extra effort from the developer. In some frameworks, especially in those for old system programming languages such as C and C++, it is a tedious to task to register test methods to be recognized by the framework. In modern environments, the test method is easily announced to the framework by adding an attribute on top of it.

A wide variety of high-quality *assertions* provided by the framework greatly increase the usefulness of the built-in library. The purpose of an assertion is to check that a certain condition is met when executing the code. Well-equipped frameworks support basic features, comparing all primitive types and many commonly used collection types, as well as advanced features such as mocking, advanced assertions (e.g. “check that a method call crashes the process”) and verbose error logging and diagnostics.

Mocking is related to unit testing frameworks closely. Mocking frameworks provide features for fluently creating small objects or implementations of interfaces to be used in unit tests as replacements to real libraries. Mocks are used to provide minimalistic behaviour that is enough to run tests against some code. The behaviour of mocks is preconfigured and they can be used to ensure that methods are called with certain parameters.



```
TEST(PainterTest, CanDrawSomething) {
  MockTurtle turtle;
  EXPECT_CALL(turtle, PenDown())
      .Times(AtLeast(1));

  Painter painter(&turtle);

  EXPECT_TRUE(painter.DrawCircle(0, 0, 10));
}
```

Listing 2. Example of a mock object usage (copied from Google 2011) in GoogleMock with GoogleTest unit testing framework.

In the above example, the *turtle* interface is mocked and configured to expect at least one call to method *PenDown*. If the *painter* does not reach *PenDown* at least once, the test fails.

2.1.5 Measuring Unit Testing

Measuring unit testing as a development process is difficult. Productivity gains, higher quality software and shorter time-to-market measures might show positive signs in high-level indicators over a longer period of time, but it is challenging to link to a single process which would cause the improvement.

Software projects are always unique and very difficult to repeat (or practically too expensive to repeat); thus, it is challenging to compare how the same project with a similar project team would perform using different processes or tools.

Few studies have been done about unit testing processes and their effects on software development outcome. One study done at Microsoft (Williams, Kudrjavets and Nagappan 2009, 81-89) shows that when unit testing was used on a second version of a software project for one year, a significant drop of defects was seen compared to the previous version, at the cost of increased development time. The study also suggests that the work of testers “changed dramatically” because the “easy” defects were found by the developers or were prevented due to the presence of automated unit tests” (Williams, Kudrjavets and Nagappan 2009, 81-89).

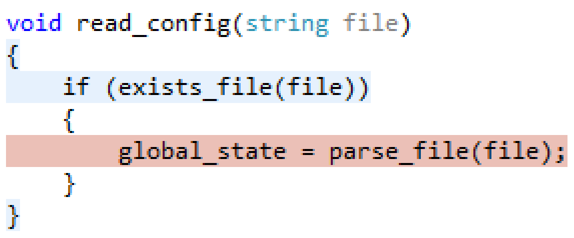
2.1.6 Measuring Individual Unit Tests

Measuring *unit tests* is somewhat easier than measuring the process. The code physically exists, and can be technically analysed in itself, and in terms of actions it performs. The number of test methods or the amount of test code does not indicate how well they cover functionality, but can for example be used to track the ratio of unit test code to production code, and whether the ratio is improving or degrading.

The most relevant measure of unit tests is the *coverage*, which answers the question "how much of the production code was executed by the unit tests".

Common technical measurements include the following:

- Method/function coverage reports the percentage of methods or functions that are covered.
- Statement coverage reports the amount of program statements that are covered.
- Branch coverage measures if different program statements are reached by different control flows from branching execution commands (including if and while). For example, the branching command "`if(a || b)`" offers four different execution paths, of which three would reach inside the if branch, and one ('a' and 'b' both evaluate to false) would not. In this example case, a branch coverage of 25% (¼) could be returned if the control statement never evaluated to true.



```
void read_config(string file)
{
    if (exists_file(file))
    {
        global_state = parse_file(file);
    }
}
```

Listing 3. Unit testing coverage report visualized in Microsoft Visual Studio.

The listing above demonstrates how code coverage can be visualized in a development environment, directly in the source code editor. The light blue background highlights executed statements and light red highlights statements that have not been executed. This visualization does not take into account for branch coverage or function coverage.

The immediate value for the developer in seeing coverage data embedded in the development environment is the ability to add relevant tests to cover missing areas quickly and without task-switching.

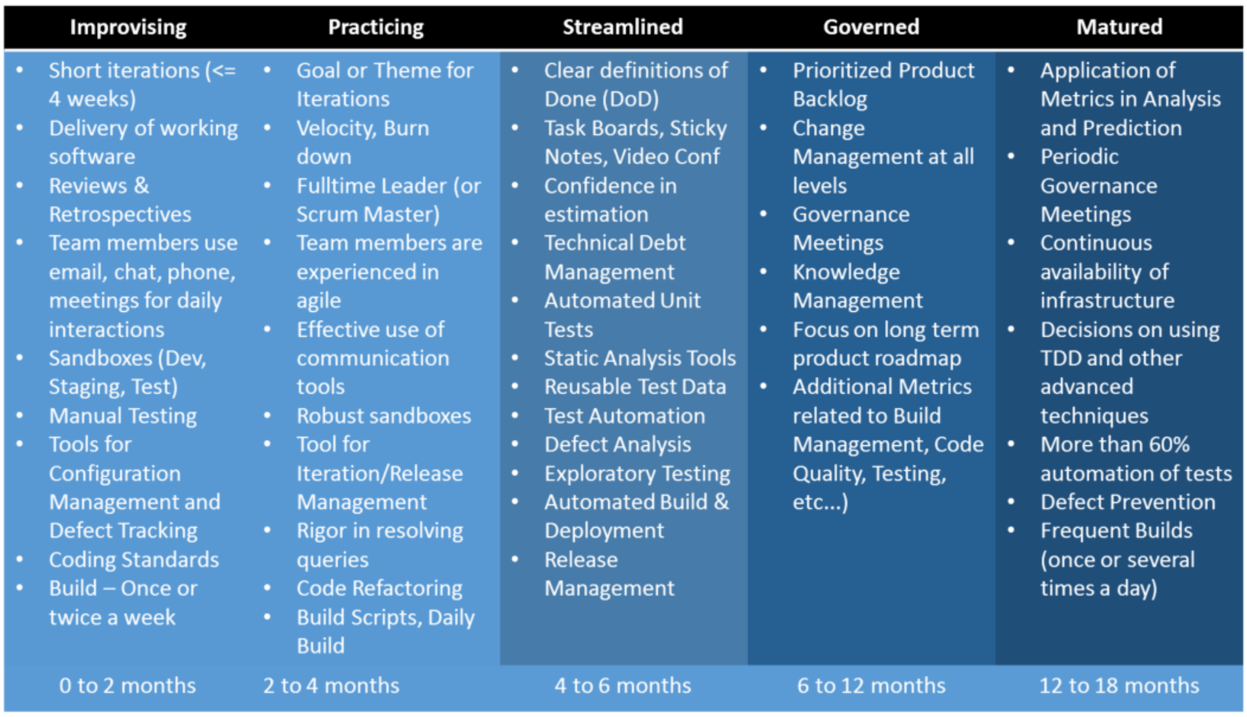
2.1.7 Unit Testing as a Part of an Agile Process

The *Agile Manifesto* (Manifesto for Agile Software Development 2001) is a commonly agreed set of good development principles, aimed to change the way software industry works. The manifesto defines *working software delivered early* as one of the main objectives.

Working software releases, especially in the case of large code base, are achieved by testing early in the process, focusing on automated tests implemented by software engineers who themselves create them as they write production code (Workroom Productions Ltd 2007, 5-6).

Writing tests early, for all bug-fixes, the efficiency throughout the process is increased. Software does not get broken in a similar way twice, and developer confidence in changing code increases. This is in line with the views presented in the Agile Manifesto on technical quality: “Continuous attention to technical excellence and good design enhances agility.” (Principles behind the Agile Manifesto 2001). Also, the initial quality of the software is better and the company’s ability to produce a final product is faster, if covering defects are found during development with unit tests (How Google Tests Software 2011, 219).

Adopting agile practices in an existing organization can be a tedious process and take a considerable amount of time. Raja Bavani has compiled the *Agile Maturity Curve* (Bavani 2012), a framework for implementing an agile process in a distributed development team (see figure 3).



Improvising	Practicing	Streamlined	Governed	Matured
• Short iterations (<= 4 weeks) • Delivery of working software • Reviews & Retrospectives • Team members use email, chat, phone, meetings for daily interactions • Sandboxes (Dev, Staging, Test) • Manual Testing • Tools for Configuration Management and Defect Tracking • Coding Standards • Build – Once or twice a week	• Goal or Theme for Iterations • Velocity, Burn down • Fulltime Leader (or Scrum Master) • Team members are experienced in agile • Effective use of communication tools • Robust sandboxes • Tool for Iteration/Release Management • Rigor in resolving queries • Code Refactoring • Build Scripts, Daily Build	• Clear definitions of Done (DoD) • Task Boards, Sticky Notes, Video Conf • Confidence in estimation • Technical Debt Management • Automated Unit Tests • Static Analysis Tools • Reusable Test Data • Test Automation • Defect Analysis • Exploratory Testing • Automated Build & Deployment • Release Management	• Prioritized Product Backlog • Change Management at all levels • Governance Meetings • Knowledge Management • Focus on long term product roadmap • Additional Metrics related to Build Management, Code Quality, Testing, etc...)	• Application of Metrics in Analysis and Prediction • Periodic Governance Meetings • Continuous availability of infrastructure • Decisions on using TDD and other advanced techniques • More than 60% automation of tests • Defect Prevention • Frequent Builds (once or several times a day)
0 to 2 months	2 to 4 months	4 to 6 months	6 to 12 months	12 to 18 months

Figure 3. Distributed Agile: The Maturity Curve (adopted from Bavani 2012).

According to this framework, it takes about 4 to 6 months for a development team to reach a *streamlined* development process in which automated unit tests are included as a normal practice. Before having unit testing as a daily process, considerable amount of infrastructure work needs to be done, including setting up a continuous integration system (build scripts, daily builds), integrating a unit testing framework, refactoring existing code and writing coding guidelines.

After reaching the *streamlined* level, the development process is supported by static analysis and dynamic analysis tools, guiding metrics and test-driven development methodology. Static analysis creates programmatic feedback for the code the developers have been writing, aiming to find bugs early, as well as finding other non-conformities that differ from agreed development practices. Dynamic analysis is performed by running the unit tests in monitored environment for analysing memory leaks, other memory problems and problematic platform API usage.

Adopting these processes in organizations which have not used them previously can require a notable effort. Practically, in order to implement change successfully, organizational change is required to form new teams and to define responsibility areas for developing and maintaining new systems.

2.2 Characteristics of Legacy Software Systems

Legacy software and degeneration of product architecture is, or at least should be, a concern of any software company. Most software projects start with relatively clean code and well-thought architecture but as features are added quickly to meet deadlines, or changes are made without full understanding of the code, the overall system and parts of it eventually get harder to understand. This effect is described by Feathers in the following quote.

What starts as a clean crystalline design in the minds of the programmers rots, over time, like a piece of bad meat. (Feathers, 2004, Foreword)

A major indicator of a legacy software system is whether the code is considered to be legacy code or not. Different sources define legacy code in different ways, but some developers call any code they do not understand legacy code (Feathers, 2004, Preface xvi). Michael Feathers has a stricter view as he defines legacy code to be any code which does not have tests (Feathers, 2004, Preface xvi). He underlines the fact that changing code without tests is very risky, and it is impossible to tell if the architecture is improving or getting worse if there are no tests to give feedback.

The key is that aging by itself does not make software a legacy system by Feathers' definition. An old system with good tests can be as easy and safe to maintain as a freshly started project.

Other characteristics that indicate a legacy system include the lack of up-to-date documentation, or no documentation at all, complex build and automated testing systems, and tedious installation process.

In many companies the legacy systems reach a point when they need systematic re-engineering efforts. Alternatives vary from rewriting the system to re-engineering parts of it (Sneed 1999, 1) or even leaving it as it is and hoping for the best. Long-term business continuity follows from keeping the software relevant by rearchitecting it but that comes with its risks. It has been found that in a rearchitecting project, testing is easily a major bottleneck, performance degrades (due to an increase of instructions executed, even though code line count would go down) and programming errors happen, diverting the control flow and otherwise altering the behaviour (Sneed 1999, 3-4).

2.2.1 Complexity

Software products by their very nature grow to be complex systems (Brooks 1987). The complexity, which makes the system harder to understand, is a major cost in software development. For example, it is estimated that in the 2000s, the maintenance cost of a software product can be up to four times larger than the initial development cost (Pfleeger and Atlee 2010, 550). The complexity problem is so typical to software products that it has been named as one item in the “laws of software evolution” as pointed out in the following quote.

> *Laws of Software Evolution: As an evolving program is continually changed, its structure deteriorates. Reflecting this, its complexity increases unless work is done to maintain or reduce it (Pfleeger and Atlee 2010, 541).*

Software complexity can be understood in a few different ways, but this definition focuses on how the developer understands the code. As a result of highly complex software, it can become very difficult to work with, which might lead to slow development times and a defective product.

Why the complexity measurement is relevant for legacy systems, is the fact that they have been usually developed for a very long time. When adding new components to software, in most cases the overall system’s complexity “increases much more than linearly” (Brooks 1987). Structuring the programs into classes and modules are ways to make the complexity manageable (Beck and Diehl 2011).

Two commonly used measures for studying are *Halstead*’s method and *McCabe*’s method (Kearney, et al. 1986). In Halstead’s method, the number of the program operators (the keywords and functions) and operands (identifiers and constants) are combined to produce the volume and difficulty metrics for the program (Kearney, et al. 1986). Combining the difficulty with the volume, the metric can be used to estimate the effort and time needed to implement the program.

McCabe’s complexity measurement measures linearly independent execution paths through the code (McCabe 1976). The volume of the code does not reflect how complex it is, but how different control statements (e.g. *for, if*) branch the program execution flow.

2.2.2 Cohesion and Coupling

Cohesion and *coupling* measurements are also related to software complexity. As already mentioned, to make a software system manageable, it is composed of modules which group functionality together and depend on other modules. The usual pattern is to collect related functionalities into a single module. *Cohesion* can then be used to measure the relatedness of code in the module, giving an indication of how well these functionalities really belong together. A modular system is described by Yourdon and Constantine (1979, 106) in the following quote.

> *The most effectively modular system is the one for which the sum of functional relatedness between pairs of elements not in the same module is minimized.*

Well-architected code is highly cohesive and the cohesion holds a module together (Spinellis and Gousios 2009, 39). However, legacy systems often have low cohesion, which may result from a lack of documentation and further worsen the situation when new modifications are applied without sufficient understanding of the system.

As can been seen in the following figure, components of Module 1 do not necessarily belong together because of low cohesion, while Module 3 seems to be highly cohesive except part E, which may belong elsewhere

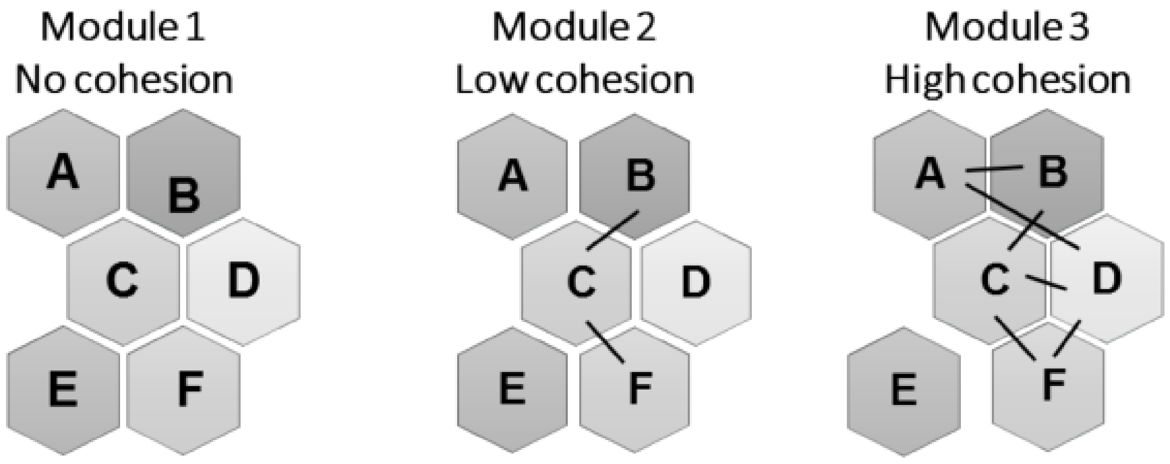


Figure 4. Examples of different levels of cohesion.

Interconnections between modules should be minimized. Modules with a low number of dependencies to other modules and little knowledge of other modules are easier to work with and easier to replace with other components. The measure of dependencies between modules is called *coupling* (Spinellis and Gousios 2009, 39). The process of systematically reducing these dependencies is *decoupling* (Yourdon and Constantine 1979, 99). Different levels of coupling are illustrated in figure 5.

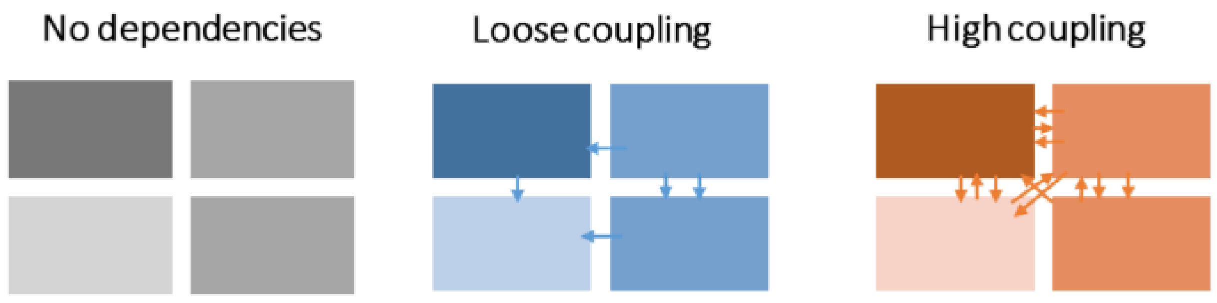


Figure 5. Dependencies between modules increase coupling (adapted from Pfleeger and Atlee 2010, 297).

Types of coupling include (Pfleeger and Atlee 2010, 299):

- Common coupling, where modules share global data.
  - For example usage of global variables instead of function arguments.
  - Global data could be scattered through many modules, making it difficult to analyse program state.
- Control coupling, where a module might know too much about other modules' internals, controlling the logic of the module by passing in parameters (data or control flags).
- Data coupling, where two modules share similar (but different) data structures.
- Subclass coupling (Oregon State University 2004), which happens in object-oriented languages where a child class depends on its parent class.

Unit testing a highly coupled system is close to impossible as the individual components cannot be used alone to perform meaningful work, since they depend deeply on other modules (Spinellis and Gousios 2009, 40).

According to Heikniemi (2012, 20), half of the effort of introducing unit testing to development is refactoring dependencies to make code testable, while the other half includes learning the practice of unit testing, setting up the tools and writing the tests.

Coupling and cohesion are not just matters of maintenance cost or testability, but important factors of realized software quality. Studies have shown that high coupling and low cohesion in software correlates positively with the amount of defects. A study on a C++ telecommunication application concluded that these metrics can be used to predict defective classes in an object-oriented system, and that coupling of classes increases the probability of having a defect in there (Succi, et al. 2003).

In today's world information security is a hot topic. Software faults can be local, and users can find workarounds to accomplish their tasks, but information leakages can have wider impact, as they might interfere with users' privacy or security. With connected software, it is relevant to understand that these previously presented low-level software metrics can have a major effect on software quality. This is shown in an extensive case study done about Mozilla Firefox, which found that the parts of the product, which were non-cohesive, complex and coupled, were more likely to be less secure (Chowdhury and Zulkernine 2010).

2.2.3 Unit Testing Legacy Software Systems

Writing unit tests against existing code is usually challenging because tests can only be written for code that is designed in a certain way. In testable code the dependencies are injectable and other design patterns are properly used.

Commonly appearing problems include the usage of singletons and a global state, and class constructors that contain production code and have non-injectable dependencies. APIs that do not enable mocking objects or interfacing database or network IO for testing are another major problem.

Kenneth Truyers (2012, 1) describes the refactoring problem like this.

> *You can't refactor because you don't have tests and you can't write tests because you can't refactor.*

The "Legacy Code Dilemma" is also identified by Feathers (Feathers 2004, 16). To solve the problem, he proposes a set of refactorings that can be done safely, with minimal risk of breaking the existing code to enable unit testing. For a while, after making the code testable, the overall architecture might become messier since all dependencies cannot be resolved in an ideal way (Feathers 2004, 18).

The following example shows a refactoring of a piece of code to make it testable. The C++ method in Listing 4 reads a file and calculates the length of the contents. However, due to one small implementation detail, it is difficult to test with unit tests.

```
size_t length_of_content(string file)
{
    size_t sum = 0;
    vector<string> contents = IO::read_file(file);
    for(auto line : contents)
        sum += line.length();

    return sum;
}
```

Listing 4. Code for calculating the length of a file.

Unit tests may not access the file system, network or other resources, as they should be standalone. With this method, the problem is that the call to *read_file* cannot be mocked for testing: it cannot be replaced with another function call on runtime, because it is a static method.

A straightforward way to refactor this would be moving the file system dependency higher in the call tree, leaving only the relevant production code in place, which is now easy to test with predefined test data. This is demonstrated in Listing 5.

```
size_t length_of_content(vector<string> contents)
{
    size_t sum = 0;
    for(auto line : contents)
        sum += line.length();
    return sum;
}
```

Listing 5. A refactored version of the code for calculating the length of a file.

Another way (presented in Listing 6) would be implementing an interface, rather than using a static method for reading the file. The interface can be replaced at runtime with a mock implementation producing data for unit testing.

```
size_t length_of_content(IO io_layer, string file)
{
    size_t sum = 0;
    vector<string> contents = io_layer.read_file(file);
    for(auto line : contents)
        sum += line.length();
    return sum;
}
```

Listing 6. Another way of refactoring the code for calculating the length of a file.

The second technique is commonly called “extract interface”, also presented by Truyers (Truyers 2012). However, the first refactoring is more suitable for this problem due its simplicity.

The question remains: How will we know that we did not break anything when we changed the code?

3 Introducing Unit Testing into a Large Legacy Software System

3.1 Description of the Development Environment

Product X has a long development history, starting in late 80's. The originally multiplatform software was converted into a single-platform (Microsoft Windows-only) product in the 2000's.

Product X was originally developed in C, with C++ taken into use in the early 2000's. The product package in total is composed of about 5 million lines of code. This research focuses only on the "Core" part (see figure 6 for high-level architecture), which has grown from 1.5 million lines of code in 2004 to 2 million lines of code in 2008 (Company A 2014).

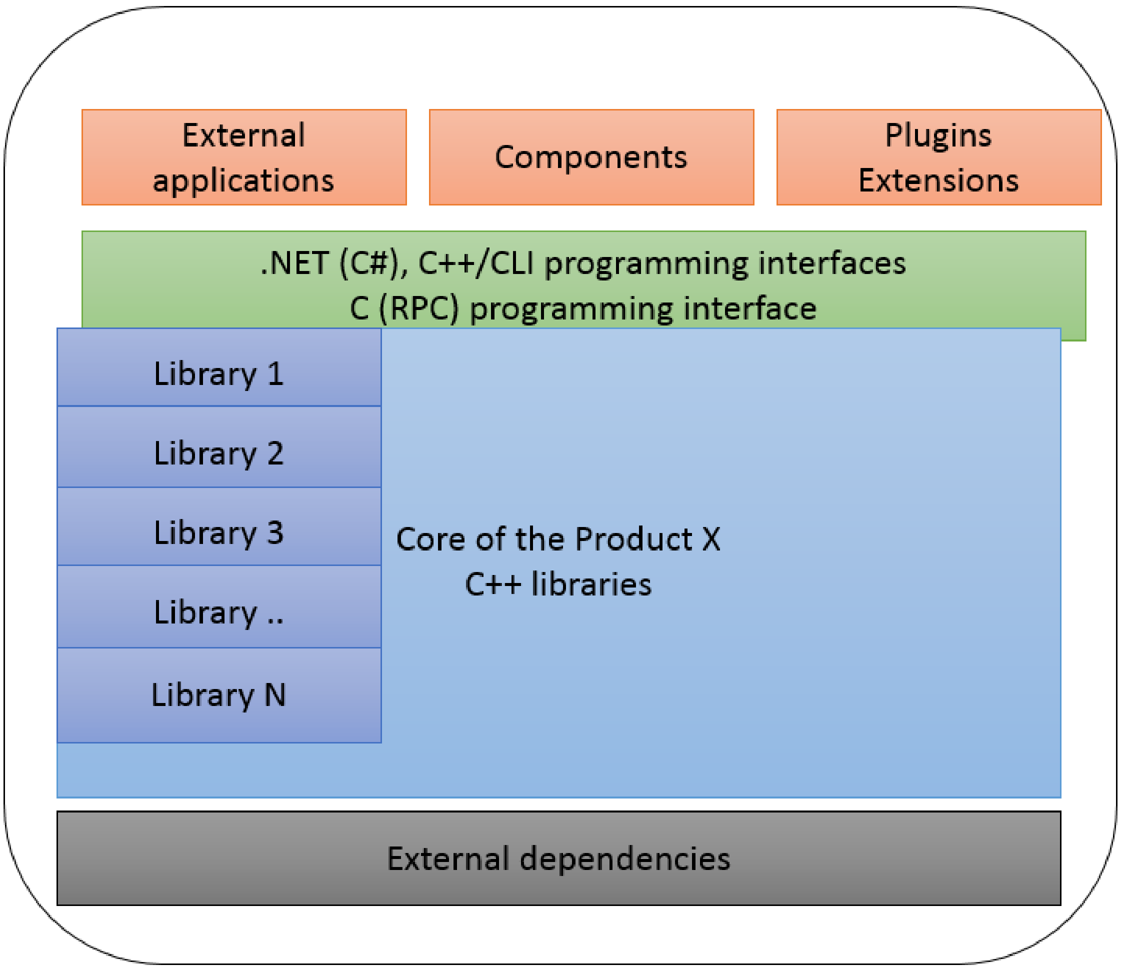


Figure 6. High level architecture of Product X.

While accumulating a considerable amount of code and new product features, the system has also accumulated a lot of technical debt, essentially turning the product into legacy software, according to many definitions of the phrase.

The development tools are the following:

- Microsoft's Visual Studio integrated development environment (IDE) with additional extensions to enhance productivity.
- The software is built using Microsoft's build tools with the Microsoft C++ compiler and linker.
- Integration builds are automated and made periodically of snapshots from the version control system.

In 2008, the Core consisted of 122 code libraries grouped into a single module. This module was used by the software engineers and the build system to produce the final Core executable. Due to the large number of libraries in Core, build times could be as high as one hour when performing a non-incremental build. Smaller changes to code base could result in build times from a few minutes to tens of minutes.

Having to build six different configurations of the product creates a performance challenge for the build system. The product is built for 32-bit and 64-bit platforms with internal debug and release modes, in addition to official release builds.

3.2 Initiative to Start Unit Testing in Late 2007

Working with Product X was found challenging by the developers in 2007, as the product was already quite old. Several decades of work and dozens of different engineers were the main reasons that the complexity had increased and module structure had become hard to work with. Automated programmatic tests did not exist and the high-level integration testing farm was only a few years old.

The lack of architectural documentation as well as other module-level documentation and a clear vision of future development had caused the code to degrade, making it difficult to maintain.

At the time, groups of individuals started to see that the development practices would need to be re-evaluated. These feelings got backed up by new ideas gathered by attending to developer conferences (Heikkonen 2014).

3.2.1 Integration of CppUnit Framework

The first improvement ideas were soon realised in the form of integrating the CppUnit (CppUnit 2009) unit testing framework into the Core module. The build of Core was configured to have two output targets, one for building the main application executable and another one to run the test methods using CppUnit.

The main use-case for the unit test runner soon after first implementation was creating test cases for a specific code library that had been found to be architecturally challenging and was known to be developed further in future (Villa 2014). In addition to that, the unit tester was planned to be used for prototyping and experimentations such as learning unit testing.

The most important features of the unit test runner were:

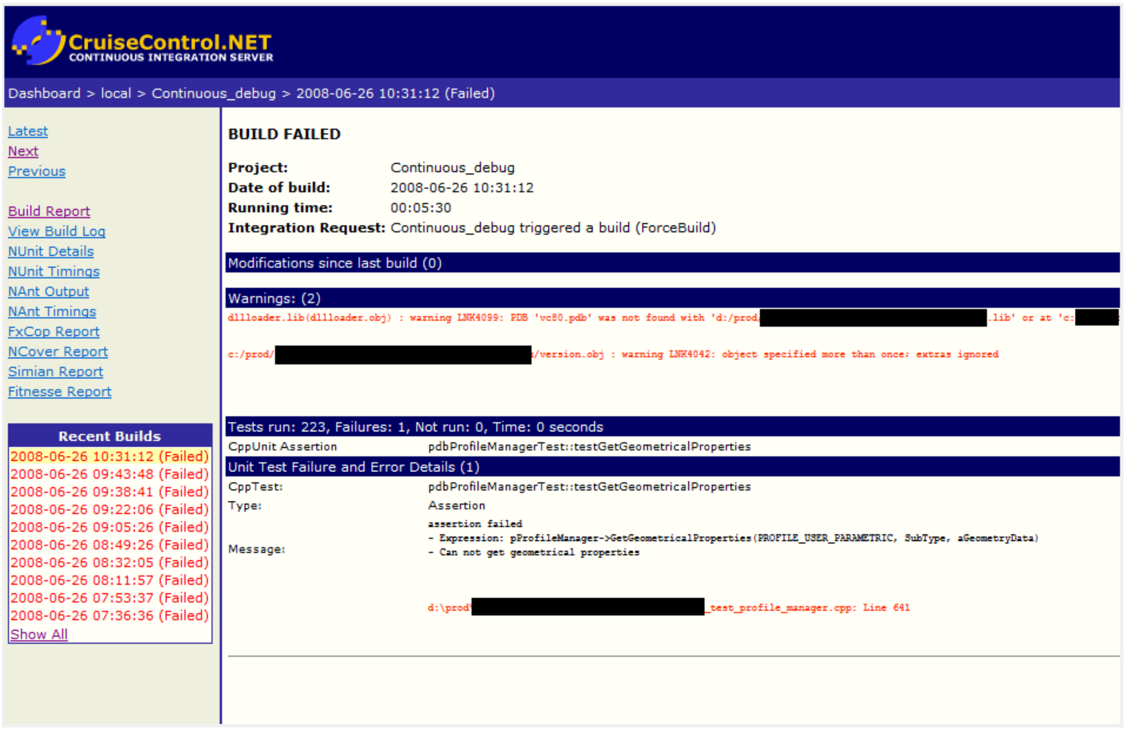
- A possibility to pause the execution in the beginning, which enabled a developer to attach a debugger
- The ability to create XML reports, which allowed integrating the runner into the build system.

From the beginning, the test runner had two categories of tests: the tests were either pure unit tests or tests requiring in-memory database during execution. Most of the tests had a run-time dependency to in-memory database because Product X had been fundamentally implemented in a way that many components access the database directly without any abstraction layer.

Many instructions written for the development of Product X about CppUnit framework usage focused on explaining how to use the Visual Studio debugger with the tests, and how to set tests up correctly to use mocks or the in-memory databases. This resulted from the fact that most of the production code in certain large libraries depended directly on the database (Ahtiainen, Unit testing introduction 2009), in addition to other complex dependencies. Ability to use the debugger was often useful since incorrectly set up tests often caused the unit test runner to crash without giving indication about which test method was the problematic one.

Having the unit test runner integrated into the local builds received mixed responses from developers; some were eager to try the unit testing approach and some were afraid that further increasing the size of the code-base by writing a lot of tests would not be wise (Villa 2014).

In the end, the unit test runner was integrated only into the build system (as opposed to be run locally on developers' machine) to be run after every commit. Running it before committing code changes to the version control system was not technically enforced nor included in the process yet (Villa 2014).



Dashboard > local > Continuous_debug > 2008-06-26 10:31:12 (Failed)

BUILD FAILED

Project:	Continuous_debug
Date of build:	2008-06-26 10:31:12
Running time:	00:05:30
Integration Request:	Continuous_debug triggered a build (ForceBuild)

Modifications since last build (0)

Warnings: (2)

```
dllloader.lib(dllloader.obj) : warning LNK4099: PDB 'vc80.pdb' was not found with 'd:/prod/[redacted].lib' or at 'c:[redacted]

c:/prod/[redacted]/version.obj : warning LNK4042: object specified more than once; extras ignored
```

Tests run: 223, Failures: 1, Not run: 0, Time: 0 seconds

CppUnit Assertion	pdbProfileManagerTest::testGetGeometricalProperties

Unit Test Failure and Error Details (1)

CppTest:	pdbProfileManagerTest::testGetGeometricalProperties
Type:	Assertion
Message:	assertion failed - Expression: pProfileManager->GetGeometricalProperties(PROFILE_USER_PARAMETRIC, SubType, aGeometryData) - Can not get geometrical properties d:\prod\[redacted]_test_profile_manager.cpp: Line 641

Recent Builds

2008-06-26 10:31:12 (Failed)
2008-06-26 09:43:48 (Failed)
2008-06-26 09:38:41 (Failed)
2008-06-26 09:22:06 (Failed)
2008-06-26 09:05:26 (Failed)
2008-06-26 08:49:26 (Failed)
2008-06-26 08:32:05 (Failed)
2008-06-26 08:11:57 (Failed)
2008-06-26 07:53:37 (Failed)
2008-06-26 07:36:36 (Failed)

Figure 7. A screenshot of a unit test runner integrated to CruiseControl.NET hosted build system (Company A 2008).

The screenshot from 2008 shows the fact that at the time it was not a top priority to immediately fix unit tests. The *recent builds* table lists failing builds from several hours, but it did not prevent any development work from continuing, since local builds would succeed. At the time there was no management enforcement to prioritize fixing failing tests, since the testing process was not official yet.

Individuals promoted and maintained the system in addition to their other assignments as there was no named responsible person for the system (Salonen 2014).

3.2.2 TSAR Project 2008

A multi-year roadmap for the incremental renewal of the architecture was approved in 2007. Improvements would start in a project called TSAR, using a cross-functional team, combining knowledge from several line-organization teams, for doing actual modularisation work towards higher-level goals (Ahtiainen, TSAR Project Plan 2007).

The TSAR vision identified three main points for long-term architecture development targets (Heikkonen, TSAR Vision 2008):

- More features quicker (scaling development, faster development)
- Long term sustainability (maintainability, competitiveness)
- Better quality (product efficiency, reliability in number of defects)

It was seen that unit testing would help delivering higher-quality software more efficiently. In the TSAR Vision, it was noted in a list of threats that if the architecture improvement project is not started, “Unit testing is largely [...] impossible due to high coupling” (Heikkonen, TSAR Vision 2008).

During 10 development sprints, among other work, the TSAR project accomplished modularisation of four Product X libraries as independent, reusable entities, which were covered with unit tests (Ahtiainen, Accomplished in TSAR 2010).

Extracting the libraries was relatively simple, because the libraries chosen were tool-like components with limited dependencies. For this reason, it was easy to cover the functionality with unit tests (Ahtiainen 2014).

Components were tested by covering most of the public interface functions, which ensured enough overall coverage of the targets. Automatic tools for measuring the coverage were not used in the project; thus, no coverage information is available.

Learning about unit testing was a part of the project since the process was new to all developers participating. Architecturally, the project helped to realise how much work it would be to improve the rest of the system (Ahtiainen 2014).

3.2.3 Software Engineers' Reactions to Unit Testing Initiatives

After TSAR and other initiatives for unit testing, a workshop was held to gather improvement ideas for the unit testing process (Rihtniemi 2014). The virtual architect team was organizing the workshop and lead developers were invited to bring in more viewpoints.

The main findings of the workshop were (see Appendix 1):

- Refactoring is needed to write unit tests.
- Developers are not aware of how to write good unit tests.
- Some fear the process change.
- People do not see the benefits of unit testing.

All the comments are natural from developers who have never worked with a unit-tested, unit-testable and modularised code base. The “I'll believe it when I see it” attitude is visible in the answers of the workshop. Some developers were not even ready to try unit testing because they felt the effort was too high.

To improve the situation, several actions were implemented. A guideline for unit testing was written to match the need for information sharing and documentation. People were encouraged to continue improving the system by educating other developers face-to-face, using pair-programming for example.

4 The PAMM Modularization Project 2010

4.1 PAMM Vision

The fact that computer software grows in complexity as it evolves, as presented in chapter 2.2.1, was also visible in Product X. Effects of complexity could be seen in employees' motivation and productivity, as well as implementation efficiency for creating new features and maintaining existing ones.

Project PAMM started in 2010, as a long-term initiative to ensure business continuity and personnel satisfaction with the working environment. Profit growth and business continuity would result from the capability of launching new products more easily, based on well architectured existing products. Product development would work more efficiently by introducing fewer defects in existing products when the architecture is sound, making unit testing and other modern ways of development possible.

Product X was, until 2009, compiled together so that all of the core code libraries were linked into a single monolithic executable. In general, when more libraries are added into a project, the linking part soon becomes a bottleneck in the build process of a large C++ project.

The following figure displays the increase in the number of libraries in Core, compared to the number of executable modules produced from those libraries.

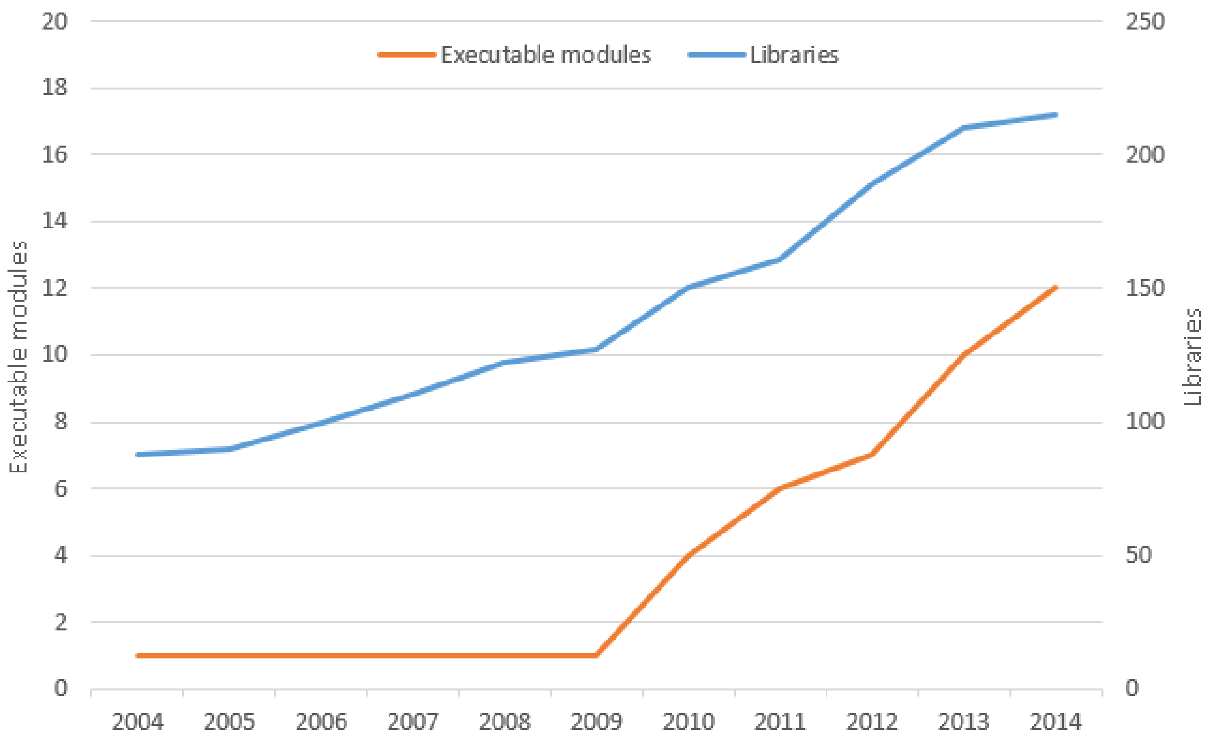


Figure 8. Number of executable modules and code libraries 2004-2013.

The architecture of combining all libraries into a single module allowed all of them freely interconnect with any other libraries in the system, which during the years resulted in low cohesion and high coupling. Because of the large size of the code base and the lack of clearly defined architectural boundaries, all types of coupling presented in 2.2.2 and other sources can be found.

Based on the logical architecture of the existing system, the practical goal of the PAMM project was to split the large monolithic system into four smaller modules.

Two large main subsystems were identified in the product and modularised as their own entities. Both of these depended on common functionality, which was also separated into a new module. Higher level application logic and the user interface logic were left in the remaining module.

Architecturally, this change means more visibility of the high-level product structure to the developers, by making the module structure clear already in the development environment (Ahtiainen 2014).

In their daily work, the developers spent quite much time compiling the monolith; even incremental builds of the system were slow. For example, linking times were measured in minutes rather than seconds.

4.2 Implementation of the PAMM Project

The following figure demonstrates the high-level monolithic architecture the modularisation started from, and the resulting situation after the PAMM project was implemented.

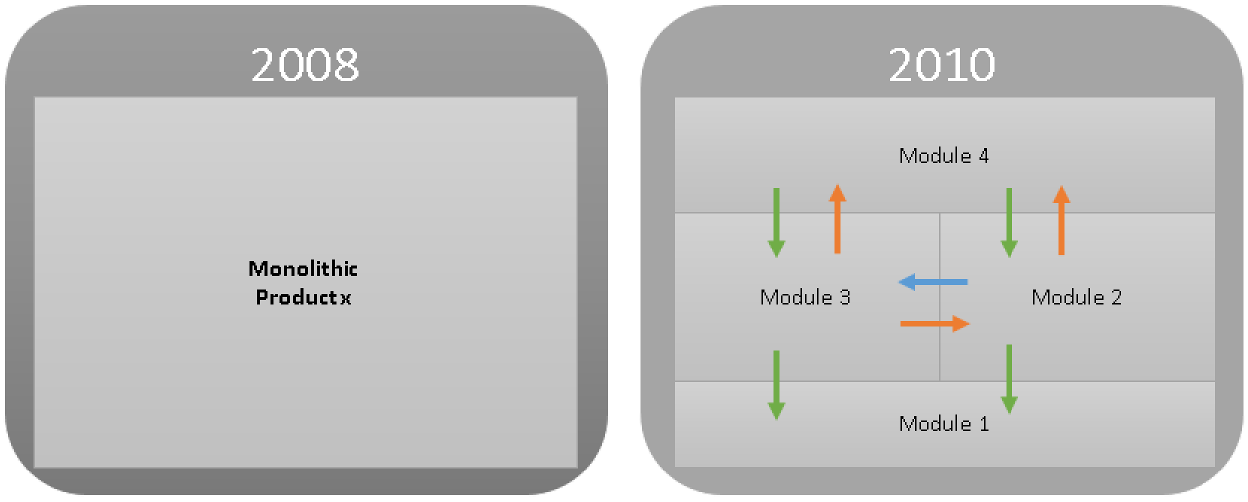


Figure 9. The starting point and the goal of PAMM project.

In the figure, the green arrows indicate compile-time dependencies and the blue and red arrows indicate run-time dependencies. As the modules have been developed to be highly interconnected, the interfaces between them could not be completely cleaned during the refactoring project, leaving the unwanted dependencies in the system.

The red-arrow (run-time) dependencies cannot be present at build-time, because there would be circular dependencies between the modules. Rather, the dependencies are connected when the actual application is initialized.

The blue arrow from module 2 to module 3 indicates a logical and architecturally allowed dependency which is also hooked in place at runtime only. This allows compiling the two subsystems in parallel.

4.2.1 Challenges and Improvements in the PAMM Modularization Project

The PAMM project was estimated to last from 6 to 12 months. Simultaneously to the refactoring project the product feature development was ongoing as usual. This created a major challenge for the automated testing system and the build system since they had to support two ways of delivering binaries, which had not been a requirement earlier.

A new way of building the modular software was implemented while most developers still worked on the old way of building a monolithic application. After PAMM was carried out successfully, each new module and the main executable module had their own unit test runners. Developers working with only improvements to module *2* were able to run only the subset of tests included in that module when performing incremental builds.

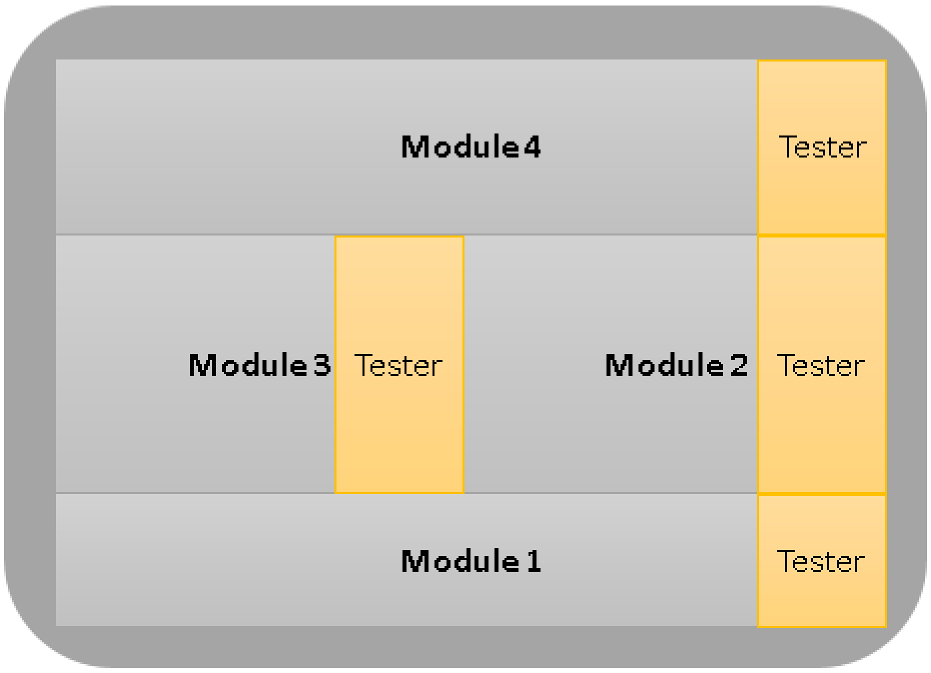


Figure 10. Unit test runner projects attached to each module.

As the build system was overhauled to support building modularized software, the unit test runner also received an important update. Each module was configured to include a new project for creating an executable runner to run unit tests from libraries included in the module (see figure 10).

The efficiency of the edit-compile-test cycle was greatly improved. The ability to narrow down regressions was increased when only a relevant subset of the tests was run. This was a major improvement over the previous situation, in which all the unit tests were run every time a build was made. Improvements made it possible to make unit tests run as part of every developer build when they built any of the modules.

Failing unit tests would fail the build and the developer would have to fix the issue immediately instead of carrying on with possibly broken code. Before this change was made, defective code was sometimes checked into the build system, hindering other developers when they got the latest code.

The dependencies between modules, both allowed and not allowed, were combined into large C++ interfaces and exported from the modules. This way they could be used as interface documentation, clearly indicating inter-module dependencies, therefore making the current state of the architecture visible. In addition to architecture documentation, the interfaces could be used in unit testing for creating mock implementations of them.

5 Systematic Development of Unit Testing Activities

In the beginning of 2012 two full-time employees were assigned to develop unit testing practices in Company A’s product development unit. A vision for the improvement work was formed, based on previous experiences and feedback from development teams.

> *“Unit testing is feasible and considered a normal part of development work.” (Lindqvist, Unit testing vision 2012).*

The vision acted as longer term target, but practical work started from creating an improvement roadmap. For the roadmap, the developers were interviewed about the issues related to unit testing. Comments from interviews included (see Appendix 2):

- Legacy code is impossible or difficult to test.
- Team leads/product owners are not interested in technical quality.
- Some of the existing tests are badly implemented.
- Too much setup is needed because of dependencies.
- Direct dependencies to database do not allow creating pure unit tests.
- Feedback from the continuous integration system is slow.

The first two items are organizational development items, not something that was addressed immediately. The ideas were communicated further to relevant parties. The product architecture unit, with the help of the lead developers, would start the necessary refactoring projects, as well as educate personnel about legacy code.

Other concrete action items included holding a test-driven development study group for interested developers, scheduling a project to move away from CppUnit and a decision to start collecting metrics about improvements.

The study group about test-driven development and unit testing in general was a success and had a high number of participants. The company provided time and facilities for trying out newly learned techniques about writing good unit tests in a real-life code base. Since all code is reviewed in the development teams, also those not participating in the group were exposed to unit testing and were encouraged to learn and give feedback.

To make technical quality everyone’s business in product development, new guidelines were agreed upon with line managers (Lindqvist, Unit testing guideline 2012):

- It is not allowed to introduce new coding guideline violations during projects.
- Unit testing coverage may not decrease.
- All new code must be unit tested to meet a certain coverage level.

The new guidelines, like the existing ones such as that code reviews are mandatory, would be followed up periodically during feature development.

5.1 Change to GoogleTest Framework

The increased volume of unit tests and users of unit tests also meant greater demand for better tools and a higher-quality unit testing framework. The status of the CppUnit framework was evaluated with the conclusion that its usage would not support future needs. The last release was in 2009 with no forthcoming updates.

Missing support for continuous integration systems and development environment integration are the main drawbacks of using CppUnit. Test development suffers from the lack of missing features of CppUnit, many of which are found as standard features in other unit testing frameworks.

Asserts in unit tests stop execution of a test method when the first problem is detected even though developers would often like to see the results of the following assertions as well. This is a problem especially in C++ environments with long edit-build-link-test cycles.

GoogleTest provides an improvement to conventional assertions called *non-fatal assertions*. Using non-fatal assertions allows the test to run through multiple checks without stopping at the first failure, and reporting them all at the end. GoogleTest also provides a very rich set of customizable assertions, which report the errors in much finer granularity than CppUnit, thus speeding up error detection.

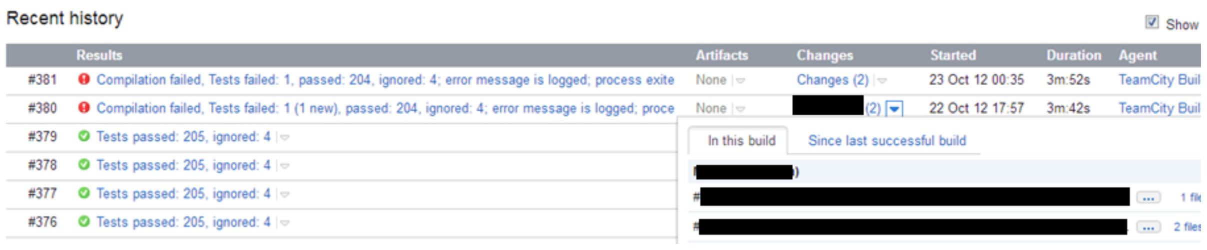
Many times tests result in a failure for a known reason, or are under development or planned to be fixed in the future. For this reason, developers need to be able to suspend the failing test in a more sophisticated way than just by removing the code or disabling it using comments. Marking a method with a *disabled* attribute allows the continuous integration system to keep track of the tests so they are not forgotten.

Useful unit tests fail sometimes, and by failing they alert developers that their code has regressed. However, not all programmatic errors lead to failing assertions. Some lead to memory leaks, or to different types of *undefined behaviour* (LLVM Project 2011), in which case the operating system usually terminates the program. During unit test runs, it is not useful to blindly terminate execution. It is more useful to catch the problem and report it in the test results. GoogleTest uses the *Structured Exception Handling* (Microsoft n.d.) of the Microsoft Windows operating system to catch undefined behaviour in test methods. It reports these failures and continues with the next test.

5.2 Continuous Integration System Improvements

In the early 2013 the *TeamCity* (JetBrains 2014) continuous integration system replaced CruiseControl.NET. For the developers, this meant greater visibility of technical quality: the number of unit test methods, both passed and failed, were displayed for each build (as seen in figure 11).

Changes from each commit are now linked to each build, which makes strict process enforcement possible, if wanted. Builds can be configured to automatically fail or send notifications if the number of tests or test coverage decreases over the defined threshold.


Recent history

☑ Show

	Results	Artifacts	Changes	Started	Duration	Agent
#381	Compilation failed, Tests failed: 1, passed: 204, ignored: 4; error message is logged; process exite	None	Changes (2)	23 Oct 12 00:35	3m:52s	TeamCity Buil
#380	Compilation failed, Tests failed: 1 (1 new), passed: 204, ignored: 4; error message is logged; proce	None	[redacted] (2)	22 Oct 12 17:57	3m:42s	TeamCity Buil
#379	Tests passed: 205, ignored: 4					
#378	Tests passed: 205, ignored: 4					
#377	Tests passed: 205, ignored: 4					
#376	Tests passed: 205, ignored: 4					

In this build | Since last successful build

[redacted])

#[redacted] ... 1 fil

#[redacted] ... 2 files

Figure 11. Continuous integration system in use.

Having a modern continuous integration system came with a lot of possibilities for other improvements as well. The tools were set up to catch new types of defects from the unit tests.

Dynamic analysis of unit tests was set up using the Intel Inspector XE toolset. Dynamic analysis reveals memory leaks, memory corruption and uninitialized memory access (Intel 2014), all common problems for native code.

Traditionally, Intel tools have been used occasionally when problems were reported or suspected, but now, with the tools continuously running, a subset of code is always tested automatically, providing invaluable feedback. If memory leaks are found from unit tests, it is very likely that the problems are visible when running the full application stack (Tiensuu 2014). Good tooling has helped to reveal several otherwise difficult-to-diagnose memory-management issues which have been fixed before the code was ever shipped to customers or the testing department.

The ability to run the tests in a system which collects the data about them has helped to improve the reliability and speed of the tests. As presented in *Characteristics of unit tests*, common traits of unit tests include *repeatability* and *fast execution speed*. The repeatability of tests is ensured by the build system running the tests in shuffled order, to automatically test them in a changing environment. This has helped to catch quite many initialization and clean-up issues from the tests, as well as other false assumptions about the running environment.

According to Bavani’s (2012) framework, the system can now be considered *streamlined*; the most relevant tools and processes are now automated, and the infrastructure is ready and utilised by development teams to deliver results.

5.3 Continuous Inspection System - SonarQube

Unit testing coverage builds, dynamic analysis builds, several static analysis tools, unit test runs, integration test builds and other automated tools created a lot of data every day. The need for a dashboard for all of this data became obvious.

The *SonarQube* Continuous Inspection system was introduced in product development. SonarQube is designed to drive towards better software by providing tools for managing code quality, as well as other services, as seen in figure 12.

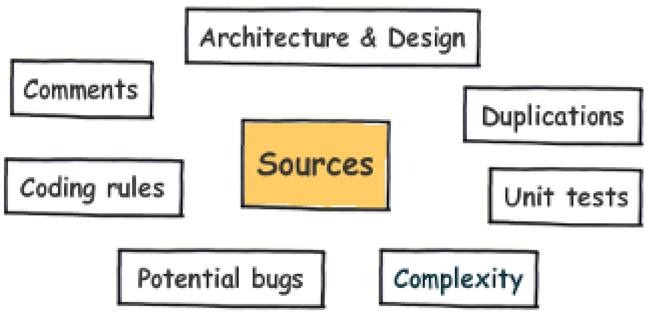


Figure 12. SonarQube framework (copied from SonarQube 2014).

The implementation work done in unit tests is also reflected elsewhere in the system; development in code coverage might also be visible in complexity or the number of new bugs.

Displaying the data does not help by itself, but drilling down in the historical development of metrics gives the possibility to analyse the current situation, predict the future state of the system and, most importantly, enables steering future work based on historical data.

Tools integrated into the SonarQube system during 2012-2013 were:

- CppCheck, static analysis tool for C++, for finding different kind of programming errors.
- BullseyeCoverage, code coverage analysis for C++.
- GoogleTest, unit test run statistics.
- Rough Auditing Tool for Security (RATS), static analysis scanner for security vulnerabilities in C and C++.
- cpplint, C++ style checking.
- Vera++, static analysis tool for C++, for following coding conventions and style.

With respect to the process, the introduction of such an extensive toolset for a large, already existing code base turned out to be challenging in the beginning. The amount of non-conforming code was huge and it was difficult to get developers to believe in the system, as fixing a hundred items from several hundred thousand does not make a noticeable difference in the graphs.

In the long run, the visibility of metrics has shown its value: the developers have been more motivated to improve code quality and unit test coverage.

6 Results of the Improvement Projects

In this chapter, the results of the change are discussed. The data which is discussed was collected from the available documentation about Product X and from a questionnaire sent to all developers who have worked with Product X during the time from 2008 to 2013.

In 2013, many of the actions implemented in 2012 really started to pay off. The systems allowed monitoring status and quality of the code on the level of individual projects and developers. The continuous inspection dashboard allowed monitoring the product's quality in terms of testing coverage, automatically found defects, architecture issues and style violations

Personal dashboards in the continuous inspection system made it easy for developers to track technical debt in the areas they worked on. The changes in metrics also allowed targeting supporting actions, when needed, to relevant teams. In legacy systems, the pace of development often slows down as complexity grows, as discussed already in previous chapters. With metric management tools and practices in place, the growth of complexity can be managed.

6.1 Architecture Improvements

Architectural improvement projects were running simultaneously to feature development for almost all the time from 2008 to 2013 and have greatly improved the modularity of Product X, as seen in figure 13.

The seven modules allow optimal workflows and a fine granularity of metrics. Clear borders between the modules have provided a way of decoupling many functionalities by just reorganising the code.

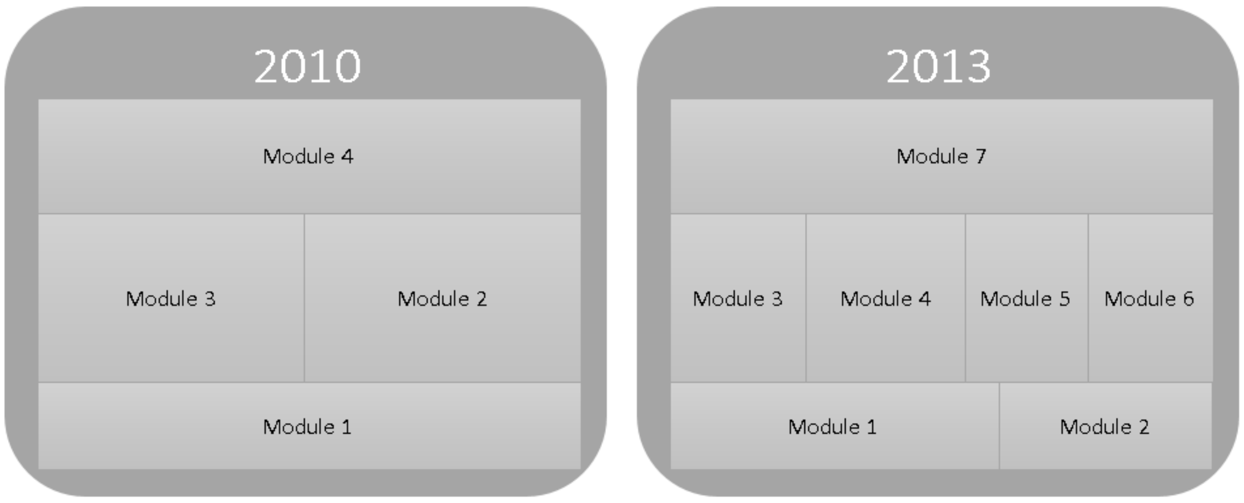


Figure 13. Product architecture development, modularisation into independent components.

Interfaces between modules act as documentation and guide further development. Implemented features are of higher quality in the current system, both from a technical perspective and feature-wise (Salonen 2014). This results in more modularised code, which is easier to maintain, and leads to increased efficiency in the development process.

6.2 Increased Use of Unit Tests

The target was to *introduce unit testing to product development*, which practically means software engineers should be changing or adding unit tests while they work on production code.

A program (see Appendix 3) was implemented to analyse the commits to unit tests in the version control system. All changes to files under “test” subdirectories in the source code repository[1] were counted as changes to unit tests. These commits were then grouped by person and year, for years 2010 to 2013[2] and analysed for trends.

---

[1] In Product X Core, the unit test code exists under a “test” subdirectory in each library

[2] Note: Linking unit tests to an issue or defect fixed would perhaps yield better results, but the limitations in the version control system currently do not make this possible. In the development it is not unusual to use multiple commits to fix a single issue, where only one of the commits implement or update the tests, and the others might fix the bug or review comments.

Persons who had a non-zero unit testing coverage from 2010 to 2013 were selected for analysis of the trend.

In the following graph, active unit testers of 2013 are presented in descending order from left to right. Different bars can be used to compare each individual to their results in previous years. In general, many of these employees have increased their changes to unit tests during the follow-up period. On average, 30% of the commits in 2013 contained changes to unit tests.

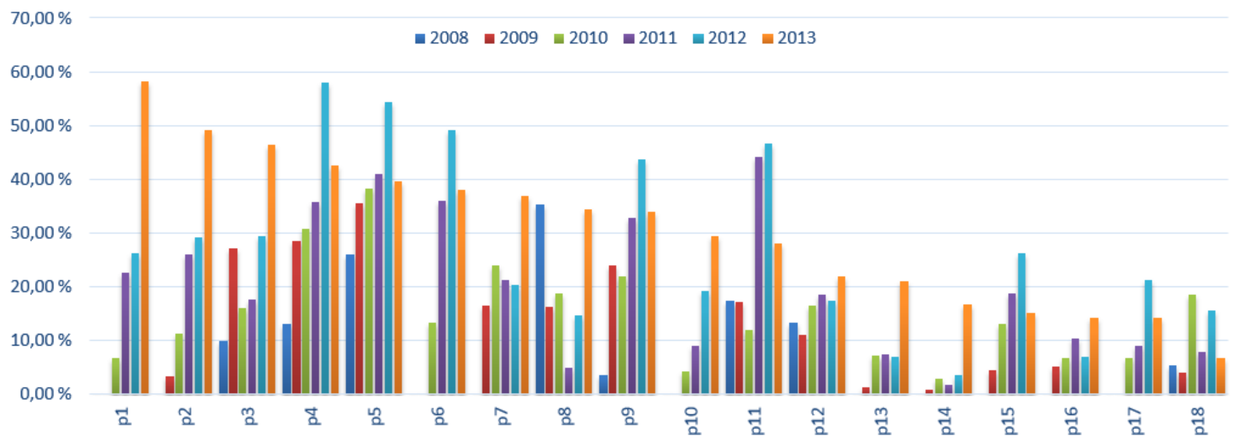


Figure 14. Average percentages of commits which included changes to unit testing-related files.

It is clear from the averages of each year's commits (see figure 15) that the percentage of changes to unit tests steadily increased every year. This increase is a logical consequence of making the process easier and the tools more available. The motivators for these improvements are discussed in chapter 6.3.

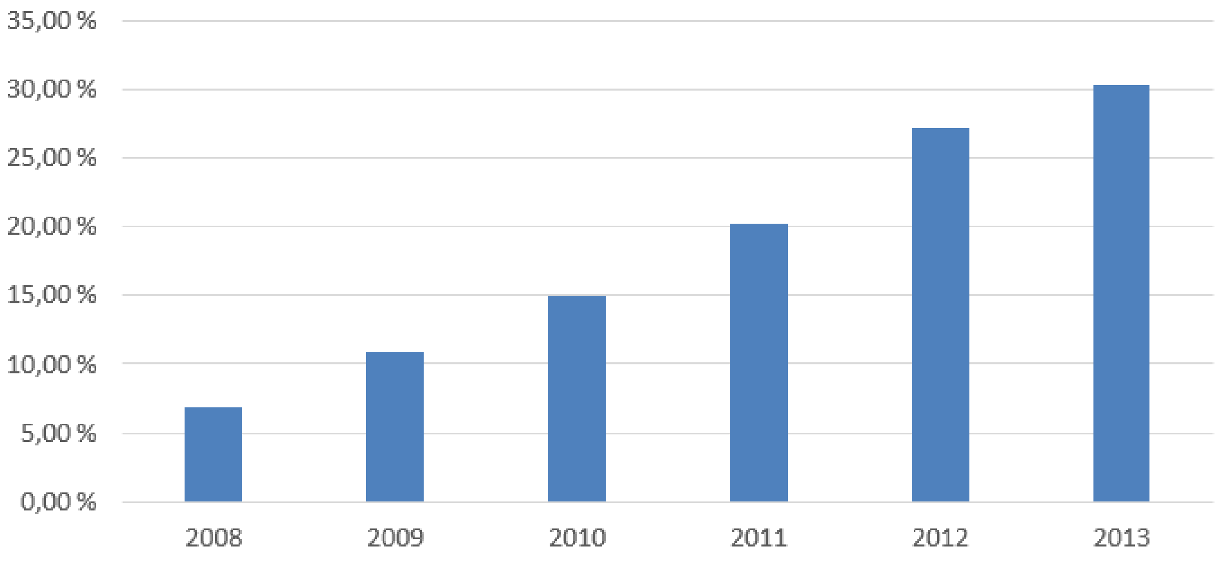


Figure 15. Average percentages of commits which included changes to unit testing-related files.

While the *changes* to unit tests soared, the number of actual tests increased too. The adoption of unit testing is visualised in the following figure. Over the years, the number of test methods added to the code base has grown steadily. The graph also highlights the modularisation of Product X by displaying the counts of tests in each module.

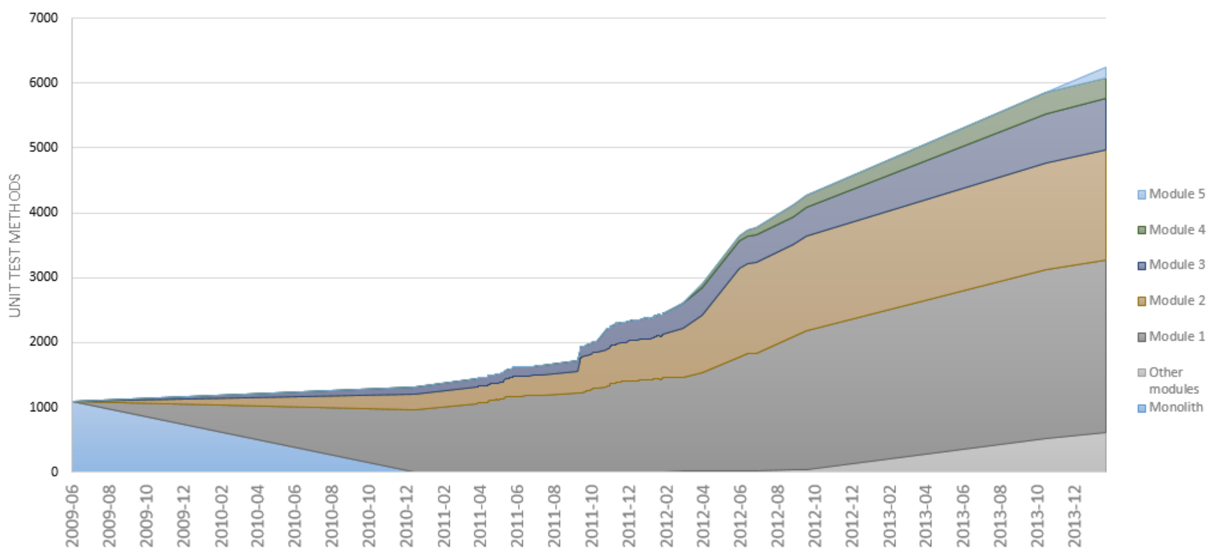


Figure 16. The number of unit tests.

It is possible to observe a similar growth in terms of coverage data as well. At the end of 2013, 35% of all functions in Product X Core were covered by unit tests and 12% of condition/decision coverage[3] was achieved.

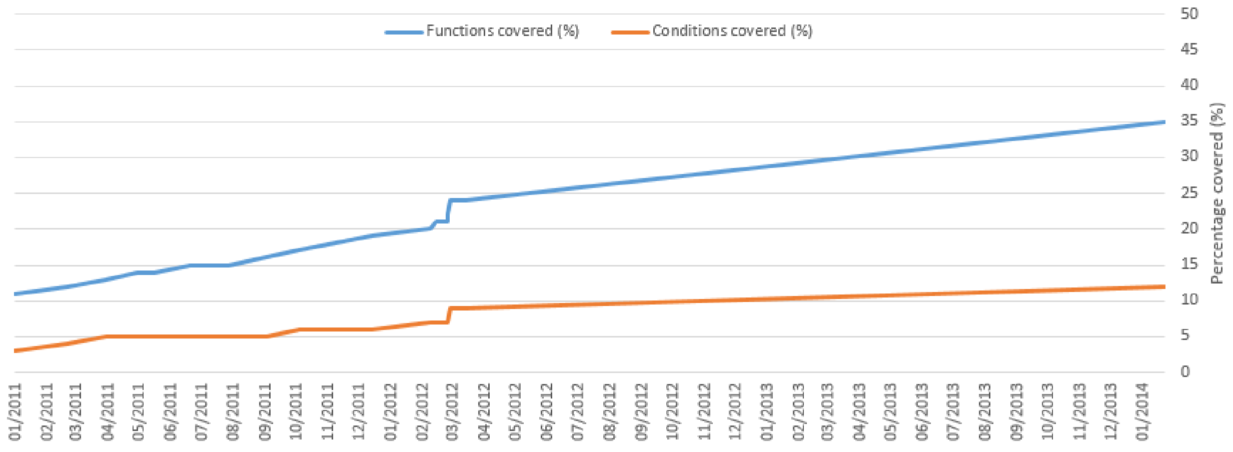


Figure 17. Code coverage development.

When looking at the overall system, it can be seen that while new features were developed, the pace of code growth was somewhat lowered and the number of libraries had increased. Having a larger number of libraries with smaller amount of code indicates that the software is more modular than before, making maintenance work easier. Architectural work has been an important part of making unit testing feasible by producing smaller, meaningful libraries that can be tested more independently or with fewer dependencies.

[3] BullsEye Testing Technology refers to decision coverage, which is also known as branch coverage presented in chapter 2.1.6. Condition/decision coverage is “composed by the union of condition coverage and decision coverage” (Bullseye Testing Technology 2011).

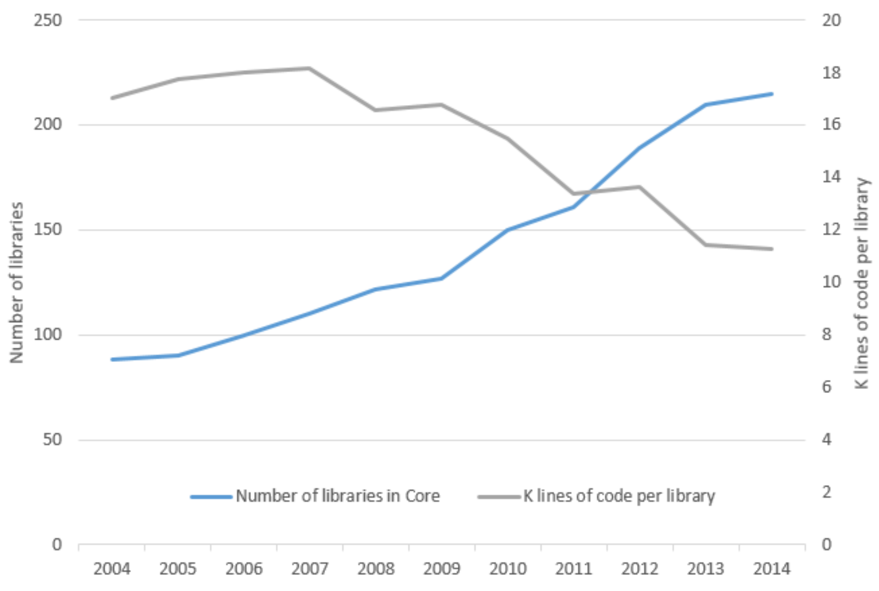
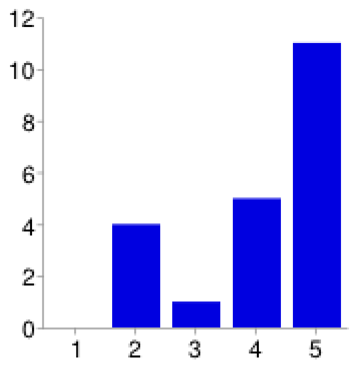


Figure 18. The development in number of libraries and average amount of code per library.

6.3 Software Engineers' Questionnaire Results

A questionnaire about unit testing related improvements was sent to developers who had been working with Product X Core from 2007 to 2013. Developers were asked to rate whether certain implemented actions to improve unit testing actually improved it in their opinion. The full questionnaire form is in Appendix 5 and the responses are in Appendix 6.

Based on the answers, most of the developers found the many architectural improvements *highly useful* (on a scale of *Not at all useful (1)* to *highly useful (5)*).



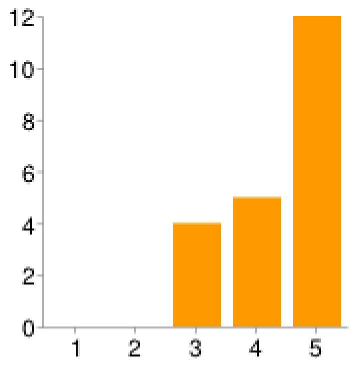
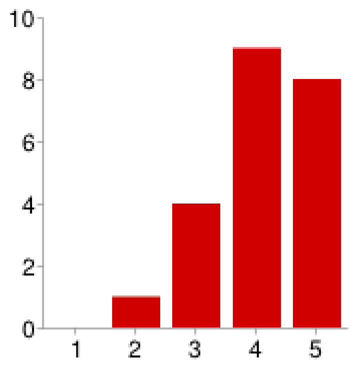


Figure 19. Left: Usefulness of high-level modularization of product.

Figure 20. Right: Usefulness of “other modularisation work” (refactoring of components, specifying interfaces, modernizing code)

The usefulness of unit testing education, including study groups, guidelines, other instructions and code reviews, was also measured in the questionnaire. Another question asked about improvements for the unit testing framework, specifically whether the change from CppUnit to GoogleTest was useful.



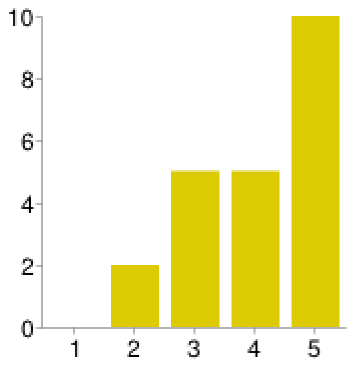
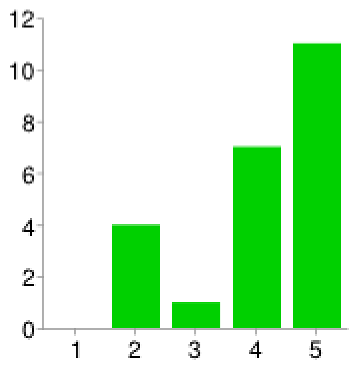


Figure 21. Left: Usefulness of unit testing education.

Figure 22. Right: Unit testing framework improvements.

Personal motivation of the employee is often the key to the success. When developers were asked what motivates them to use unit testing, the answers clearly suggest that the process is seen as a useful method for improving the product’s quality.

The answers for these questions are on a scale of *Do not agree (1)* to *Strongly agree (5)*.



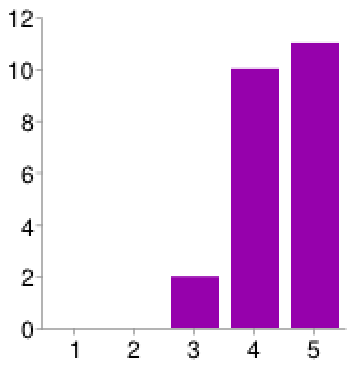


Figure 23. Left: Unit tests have helped you to find real bugs.

Figure 24. Right: Unit testing process has helped to increase the overall quality of the product.

7 Conclusion

In this thesis, six years of advancements in product development processes were analysed, mainly concentrating on the unit-testing viewpoint. A wide range of changes have been implemented across the product development department, improving the organisation, processes, tooling and the knowledge of the software engineers.

The change towards unit testing began from the bottom up, driven by motivated individuals, before any official enforcement existed. After the encouraging results from the early adopters, these practices became officially required for all developers. Developers were motivated to change their ways of working when they saw the progress and the benefits. Other positive factors that affected people were study groups, supporting guidance and peer pressure.

It has been observed during the years that for unit testing high quality tooling must be available, and a clearly defined process needs to exist, to start reaching the majority of developers.

In the case of Company A, according to all the data presented in the previous chapters, the continuous and systematic development of the product architecture greatly advanced the possibilities for unit testing. In general, smaller modules allow effective development workflows in terms of the edit-compile-test -cycle.

Based on the previous insights, the components of successful environment for unit testing are gathered in the following figure.

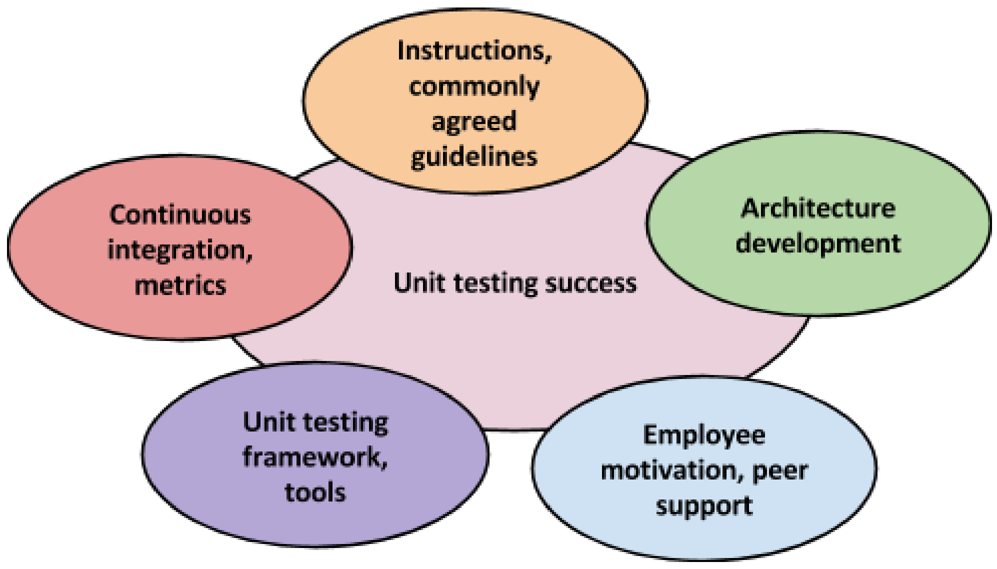

Instructions, commonly agreed guidelines
Continuous integration, metrics
Unit testing success
Architecture development
Unit testing framework, tools
Employee motivation, peer support

Figure 25. Components for achieving successful unit testing practices in a legacy software.

As a result of all the improvement work, the whole infrastructure for unit testing was taken into use and the code coverage of unit tests rose to an acceptable level. The vision of *making unit testing feasible and a normal part of development work* was achieved.

8 References

Ahtiainen, Aleksi (Software Architect). 2014. Interviewed by Markus Lindqvist on 14 January.

Ahtiainen, Aleksi. 2010. “Accomplished in TSAR.” Unpublished presentation.

Ahtiainen, Aleksi. 2007. “TSAR Project Plan.” Unpublished document.

Ahtiainen, Aleksi. 2009. “Unit testing introduction.” Unpublished presentation.

Astels, David. 2003. Test-driven development: a practical guide. New Jersey, Prentice Hall.

Bavani, Raja. 2012. “Distributed agile: The maturity curve.” Agile Record 2012:9: 26-28.

Beck, Fabian, and Stephan Diehl. 2011. “On the congruence of modularity and code coupling.” Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering 5 Sep. 2011: 354-364.

Bit by bit blog. 2012. Introduction to test driven development [online]. URL: http://bitbybitblog.com/introduction-to-test-driven-development/. Accessed April 09, 2014.

Brooks, Frederick. 1987. “No silver bullets: Essence and accidents of software engineering.” University of North Carolina.

Bullseye Testing Technology. 2011. Code coverage analysis - condition/decision coverage [online]. URL: http://www.bullseye.com/coverage.html#basic_conditionDecision. Accessed April 09, 2014.

Chowdhury, Istehad, and Mohammad Zulkernine. 2010. “Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?” Proceedings of the 2010 ACM Symposium on Applied Computing 22 Mar. 2010: 1963-1969.

Company A. 2008. CruiseControl.NET build system. Unpublished document.

Company A. 2014. Lines of Code, History. January. Unpublished document.

Copeland, Lee. 2001. Extreme programming [online]. 3 December URL: http://www.computerworld.com/s/article/66192/Extreme_Programming. Accessed January 15, 2014.

CppUnit. 2009. CppUnit [online]. URL: http://cppunit.sourceforge.net. Accessed January 16, 2014.

Dietrich, Erik. 2011. Test driven development (TDD) [online]. 22 June. URL: http://www.daedtech.com/test-driven-development. Accessed March 14, 2014.

Feathers, Michael. 2004. Working effectively with legacy code. Prentice Hall Professional.

Fowler, Martin. 1999. Refactoring: improving the design of existing code. Addison-Wesley Professional.

Google. 2011. Using mocks in tests [online]. URL: https://code.google.com/p/googlemock/wiki/ForDummies#Using_Mocks_in_Tests. Accessed January 15, 2014.

Google. 2011. How google tests software [online]. 23 March. URL: http://googletesting.blogspot.fi/2011/03/how-google-tests-software-part-five.html. Accessed January 17, 2014.

Grigg, Jeff. 2012. Arrange-Act-Assert [online]. 12 July. URL: http://c2.com/cgi/wiki?ArrangeActAssert. Accessed January 10, 2014.

Heikkonen, Teemu (Chief Software Architect). 2014. Interviewed by Markus Lindqvist on 14 January.

Heikkonen, Teemu. 2008. “TSAR vision”. Unpublished document.

Heikniemi, Jouni. 2012. Yksikkötestauksen käyttöönotto. Microsoft TechDays 2012, Helsinki.

Intel. 2014. Intel® Inspector XE 2013 [online]. URL: http://software.intel.com/en-us/intel-inspector-xe. Accessed February 12, 2014.

JetBrains. 2014. TeamCity [online]. URL: http://www.jetbrains.com/teamcity/. Accessed February 12, 2014.

Kearney, Joseph K., Robert L. Sedlmeyer, William B. Thompson, Michael A. Gray, and Michael A. Adler. 1986. “Software complexity measurement.” Communications of the ACM 29.11 (1986): 1044-1050.

Kotilainen, Lauri. 2012. Yksikkötestauksen haasteita kentältä. Microsoft TechDays 2012, Helsinki.

Lindqvist, Markus. 2012. “Unit testing guideline .” Unpublished document.

Lindqvist, Markus. 2012. “Unit testing vision.” Unpublished document.

LLVM Project. 2011. What every C programmer should know about undefined behavior #1/3 [online]. 13 May. URL: http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html. Accessed February 12, 2014.

Manifesto for agile software development [online]. 2001. URL: http://agilemanifesto.org/. Accessed December 19, 2013.

Martin, Robert C. 2008. Clean code: A handbook of agile software craftsmanship. Prentice Hall.

McCabe, T.J. 1976. “A complexity measure.” Software Engineering, IEEE Transactions on 4 (1976): 308-320.

Microsoft. n.d. structured exception handling [online]. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/ms680657.aspx. Accessed February 12, 2014.

Oregon State University. 2004. Introduction to OOP chapter 23 [online]. URL: http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/slides/chap23/slide12.htm. Accessed March 12, 2014.

Pfleeger, Shari Lawrence, and Joanne M. Atlee. 2009. Software engineering: Theory and practice. Prentice Hall.

Poppendieck, Mary, and Tom Poppendieck. 2006. Implementing lean software development: From concept to cash. Addison-Wesley Professional.

Poppendieck, Mary, and Tom Poppendieck. 2009. Leading lean software development: Results are not the point. Addison-Wesley Professional.

Principles behind the Agile Manifesto [online]. 2001. URL: http://agilemanifesto.org/principles.html. Accessed December 19, 2013.

Rihtniemi, Timo (Manager, Product Architecture). 2014. Interviewed by Markus Lindqvist on 27 January.

Salonen, Mika (Software Architect). 2014. Interviewed by Markus Lindqvist on 28 January.

Sneed, Harry M. 1999. “Risks involved in reengineering projects.” Reverse Engineering, 1999. Proceedings. Sixth Working Conference on 6 Oct. 1999: 204-211.

SonarQube. 2014. SonarQube [online]. URL: http://www.sonarqube.org/. Accessed on 15 March.

Spinellis, Diomidis, and Georgios Gousios. 2009. Beautiful architecture, 1st Edition. O’Reilly Media.

Succi, Giancarlo, Witold Pedrycz, Milorad Stefanovic, and James Miller. 2003. “Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics.” Journal of Systems and Software 65.1 (2003): 1-12.

The art of unit testing. 2011. Unit test – definition [online]. URL: http://artofunittesting.com/definition-of-a-unit-test/. Accessed December 13, 2013.

Tiensuu, Jeppe (Software Engineer). 2014. Interviewed by Markus Lindqvist on 22 January.

Truyers, Kenneth. 2012. How to unit test and refactor legacy code? [online] 15 December. URL: http://www.kenneth-truyers.net/2012/12/15/how-to-unit-test-and-refactor-legacy-code/. Accessed December 18, 2013.

Wells, Don. 1999. Unit testing framework [online]. URL: http://www.extremeprogramming.org/rules/unittestframework.html. Accessed January 24, 2014.

Whittaker, James A, Jason Arbon, and Jeff Carollo. 2012. How Google tests software. Addison-Wesley Professional, 2012. Westford, Massachusetts, United States.

Villa, Petteri (Manager, Product Offering). 2014. Interviewed by Markus Lindqvist on 22 January.

Williams, Laurie, Gunnar Kudrjavets, and Nachiappan Nagappan. 2009. On the effectiveness of unit test automation at Microsoft. Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on 16 Nov. 2009: 81-89.

Workroom Productions Ltd. 2007. Testing in an agile environment [online]. URL: http://www.workroom-productions.com/papers/Testing%20in%20an%20agile%20environment.pdf. Accessed December 19, 2013.

Yourdon, Edward, and Larry L. Constantine. 1979. Structured Design: Fundamentals of a discipline of computer program and system design. Prentice Hall.

9 Appendices

Appendix 1: Unit Testing Challenges from Workshop 2008 / Rihtniemi

The following information is from a mind map produced by Rihtniemi in 2008 for the collection of information about Product X's unit testing challenges.

- Knowledge
  - **How to write good unit tests? +++++**
  - Who is able to help?
  - No good examples available ++
  - People do not see the benefits of unit testing
  - Guidelines are outdated +
  - Unit testing has not been used much
  - Regression testing system
- Test maintenance
  - Tests need refactoring in the future
  - Code changes require test changes
  - Slows down new development in development phase
  - Fixing test defects requires time
  - Fakes have to be maintained
  - Slows down defect fixing
- Change in way of working
  - More effort needed during implementation -> tests are not done
  - Easier to continue the same way as before
  - Test implementation & planning need to be included in project items
- Framework
  - **We don't have it yet +++**
  - Building it is hard
  - Easy to use tools are missing
  - Hard to test [data] visualization
- Coverage
  - Only those parts should be tested which are touched
  - What is sufficient level [of coverage]?
  - Tools for measuring [missing?]
- Software architecture
  - Some parts of the code are difficult to test +
  - Message system [of the product] hides execution paths
  - **Refactoring is needed to enable unit testing +++**
  - Dependency of different components and UI

  - Several small changes are needed during implementation
  - Running tests is slow
  - A lot of big classes
  - A lot of different level tests are needed
  - Some tests have execution paths down in the layer structure and some up in the layer structure
- [Product] requirements
  - Unclear for old code
  - Hard to test
  - Unclear for new code

Appendix 2: Unit Testing Challenges 2011 / Lindqvist

The following information is from a mind map produced by Lindqvist in the first quarter of 2012. It was used to collect information about the status of Product X unit testing challenges relevant for developers during the past year.

- **Legacy code cannot be tested**
- Technical difficulties
  - CppUnit tests badly implemented
  - Too much work needed, could take even 80% defect fix
  - Higher level testing not possible
  - Hard to “unit test” when database faking is needed
  - Knowledge of current code base is low
  - Trial-and-error coding style makes it impossible to do test-first development
  - Fixing unit tests is not fast enough, debugger is often needed
  - Refactoring needed before a developer can unit test existing code
  - Need to fake too much: data, settings, callbacks, UI
  - Cannot test against interfaces if they do not exist
  - Basic primitive types should exist before testing (math utils, containers)
- Organizational
  - No habit of doing unit testing
  - Product/business owners not interested in “technical quality”
  - Product/business owner or team leader/scrum master doesn’t require testing
  - Feeling that by using unit testing customer defects cannot be prevented
  - Too much high-priority defects to be fixed => no time to testing
  - Testing simple code areas not seen as important
  - Good practices not shared between teams
  - Documentation and training are missing
  - Coordination: How to know if something is already tested? Where to add tests?
  - Projects are ended before features are finished, no time to do “additional things”
  - Testing of subcontracted components challenging
  - Feedback from systems is slow (automated testing, build system)
  - Projects are missing lower level test plans
- Positive findings (*these were also collected in the interviews*)
  - Success stories from unit testing
  - Positive attitude towards unit testing
  - TDD approach used in .NET API development
  - CppUnit database fakes have progressed

- Tests are integrated into builds
- Testing new code is possible and straightforward
- Tests help to understand the functionality when doing code reviews
- Seeing the progress in code coverage is motivating
- Unit testing prevents and reveals architectural issues

Appendix 3: Program (Implemented in F#) Used to Extract the Table of Data about Commits Which Contain Changes to Unit Tests

```
open System
open System.IO
open System.Reflection
open System.Text.RegularExpressions
open Microsoft.FSharp.Core.CompilerServices
open ScmHistory

let path = @"AllCheckins.txt"
let allCommits = (new ScmHistory(path)).Data

// Check if filename is a unit test file
let isTestFile (filename:string, rev) = filename.Contains("test/")
// Check if this changeset contributes to tests
let hasCheckedAnythingToTests changeset = changeset.changedfiles |> List.exists 
isTestFile

// List of individual developers
let allpersons = allCommits |> Seq.map(fun f -> f.person) |> Seq.distinct

printf "Person,2008,2009,2010,2011,2012,2013\n"
for currentPerson in allpersons do
    // For each year, fetch the ratio between contributions for testing / total 
contributions
    let data = [|2008 .. 2013|] |> Array.Parallel.map(fun currentYear ->
        let ifCurrentYearAndCurrentPerson commit =
            commit.date.Year = currentYear &&
            commit.person.Equals(currentPerson)

        let allContributions = allCommits |> Seq.filter ifCurrentYearAndCur-
rentPerson
        let testContributions = allContributions |> Seq.filter hasCheckedAny-
thingToTests

        let allContributionCount = allContributions |> Seq.length |> float32
        let testContributionCount = testContributions |> Seq.length |> float32

        (testContributionCount / allContributionCount))

    // Print data for user if it has been more than zero 2010-1013
    // This is relevant so that we can inspect the /change/ in the number of con-
tributions
    if data.[2] > 0.f && data.[3] > 0.f && data.[4] > 0.f && data.[5] > 0.f then
        printfn "%s,%f,%f,%f,%f,%f,%f" currentPerson data.[0] data.[1] data.[2] 
data.[3] data.[4] data.[5]

Console.ReadKey()
```

Appendix 4: Commits which Contain Changes to Unit Tests

Person	2008	2009	2010	2011	2012	2013
p1	0.00 %	0.00 %	6.73 %	22.53 %	26.32 %	58.14 %
p2	0.00 %	3.42 %	11.18 %	25.93 %	29.13 %	49.04 %
p3	10.00 %	27.12 %	15.93 %	17.61 %	29.31 %	46.38 %
p4	13.09 %	28.49 %	30.78 %	35.64 %	57.89 %	42.52 %
p5	26.06 %	35.58 %	38.27 %	40.87 %	54.32 %	39.50 %
p6	0.00 %	0.00 %	13.33 %	36.08 %	49.04 %	38.05 %
p7	0.00 %	16.43 %	24.04 %	21.21 %	20.27 %	36.78 %
p8	35.29 %	16.19 %	18.66 %	4.95 %	14.58 %	34.38 %
p9	3.64 %	24.06 %	22.04 %	32.88 %	43.78 %	34.03 %
p10	0.00 %	0.00 %	4.29 %	8.99 %	19.10 %	29.37 %
p11	17.44 %	17.19 %	12.02 %	44.12 %	46.67 %	28.10 %
p12	13.33 %	11.11 %	16.39 %	18.47 %	17.36 %	21.85 %
p13	0.00 %	1.27 %	7.11 %	7.31 %	6.93 %	21.03 %
p14	0.00 %	0.84 %	2.78 %	1.67 %	3.57 %	16.67 %
p15	0.00 %	4.51 %	13.18 %	18.82 %	26.32 %	15.20 %
p16	0.00 %	5.13 %	6.80 %	10.43 %	6.94 %	14.21 %
p17	0.00 %	0.00 %	6.79 %	9.09 %	21.25 %	14.16 %
p18	5.37 %	3.99 %	18.53 %	7.80 %	15.60 %	6.66 %
Average	**6.90 %**	**10.85 %**	**14.94 %**	**20.24 %**	**27.13 %**	**30.34 %**

Appendix 5: The Unit Testing Questionnaire Form

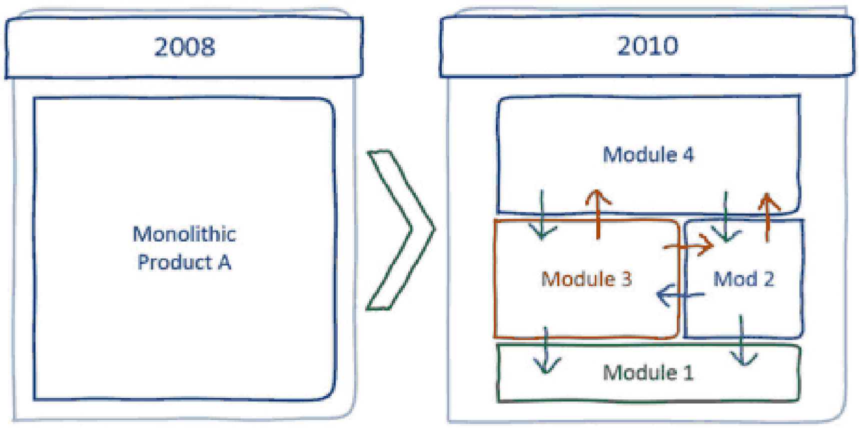
Unit testing questionnaire

What kind of improvements have you found useful for unit testing

What kind of improvements have you found useful for using unit testing process to create higher quality products since it was first taken into use late 2007.

If the question does not apply for you, you can leave it unanswered.

Example of the modularization work done

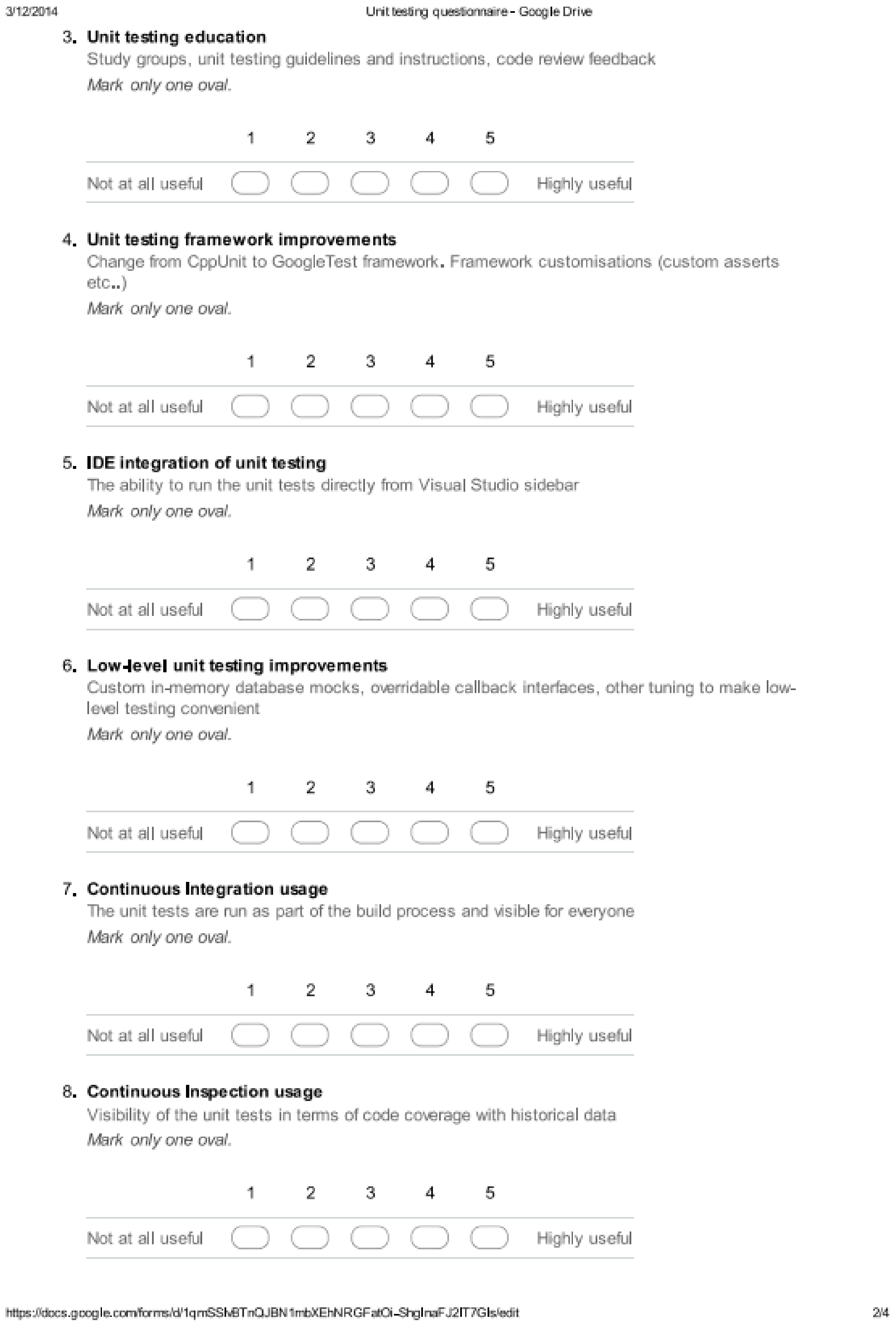


1. **Product architecture improvement - high level modularization**
   Modularization from monolithic product to several high-level modules
   Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

2. **Other modularization work**
   Refactorings of components, specifying interfaces, modernizing code
   Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful



3. **Unit testing education**
Study groups, unit testing guidelines and instructions, code review feedback
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

4. **Unit testing framework improvements**
Change from CppUnit to GoogleTest framework. Framework customisations (custom asserts etc..)
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

5. **IDE integration of unit testing**
The ability to run the unit tests directly from Visual Studio sidebar
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

6. **Low-level unit testing improvements**
Custom in-memory database mocks, overridable callback interfaces, other tuning to make low-level testing convenient
Mark only one oval.

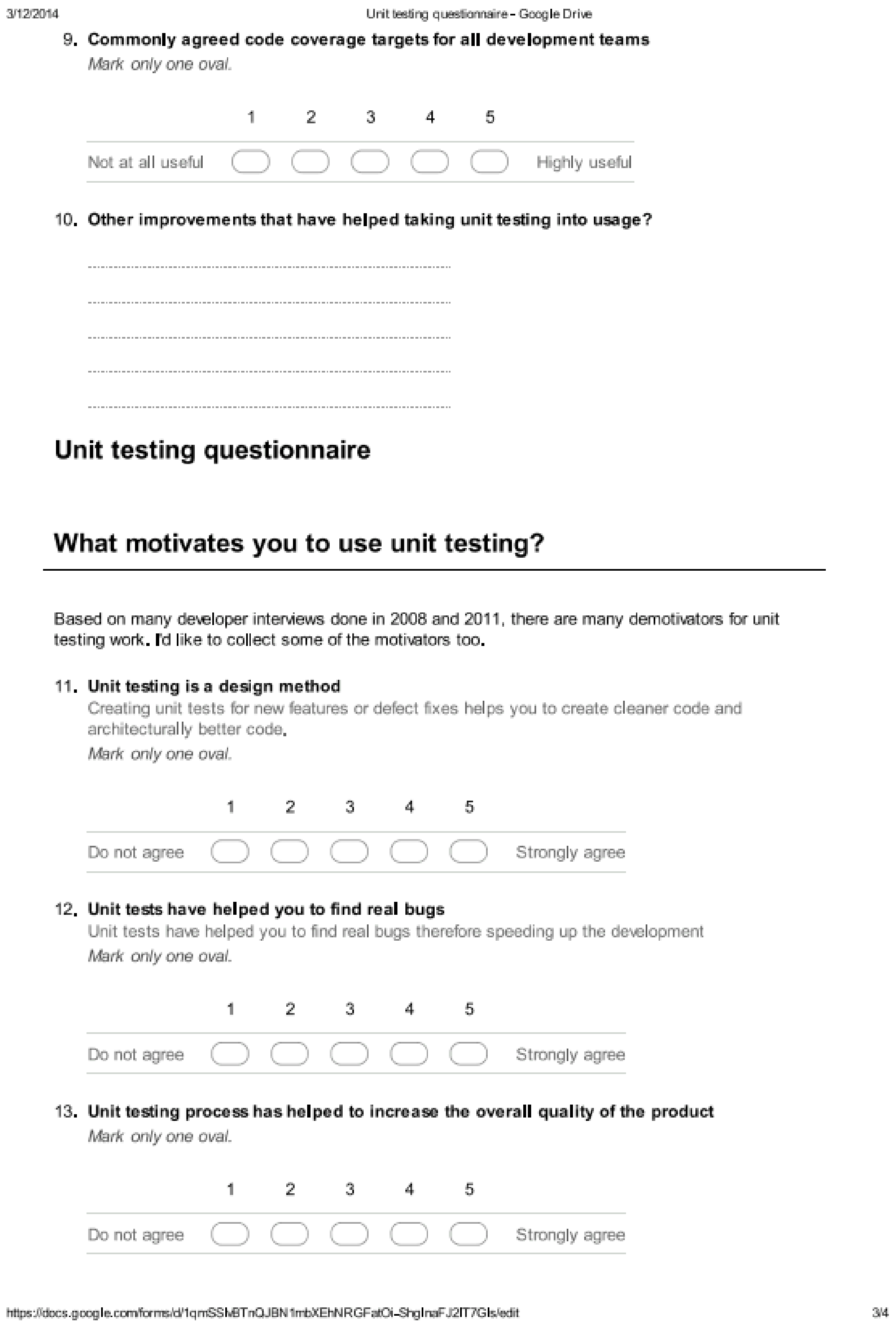
	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

7. **Continuous Integration usage**
The unit tests are run as part of the build process and visible for everyone
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

8. **Continuous Inspection usage**
Visibility of the unit tests in terms of code coverage with historical data
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful



9. **Commonly agreed code coverage targets for all development teams**
Mark only one oval.

	1	2	3	4	5	
Not at all useful	◯	◯	◯	◯	◯	Highly useful

10. **Other improvements that have helped taking unit testing into usage?**

....
....
....
....
....

Unit testing questionnaire

What motivates you to use unit testing?

Based on many developer interviews done in 2008 and 2011, there are many demotivators for unit testing work. I'd like to collect some of the motivators too.

11. **Unit testing is a design method**
Creating unit tests for new features or defect fixes helps you to create cleaner code and architecturally better code.
Mark only one oval.

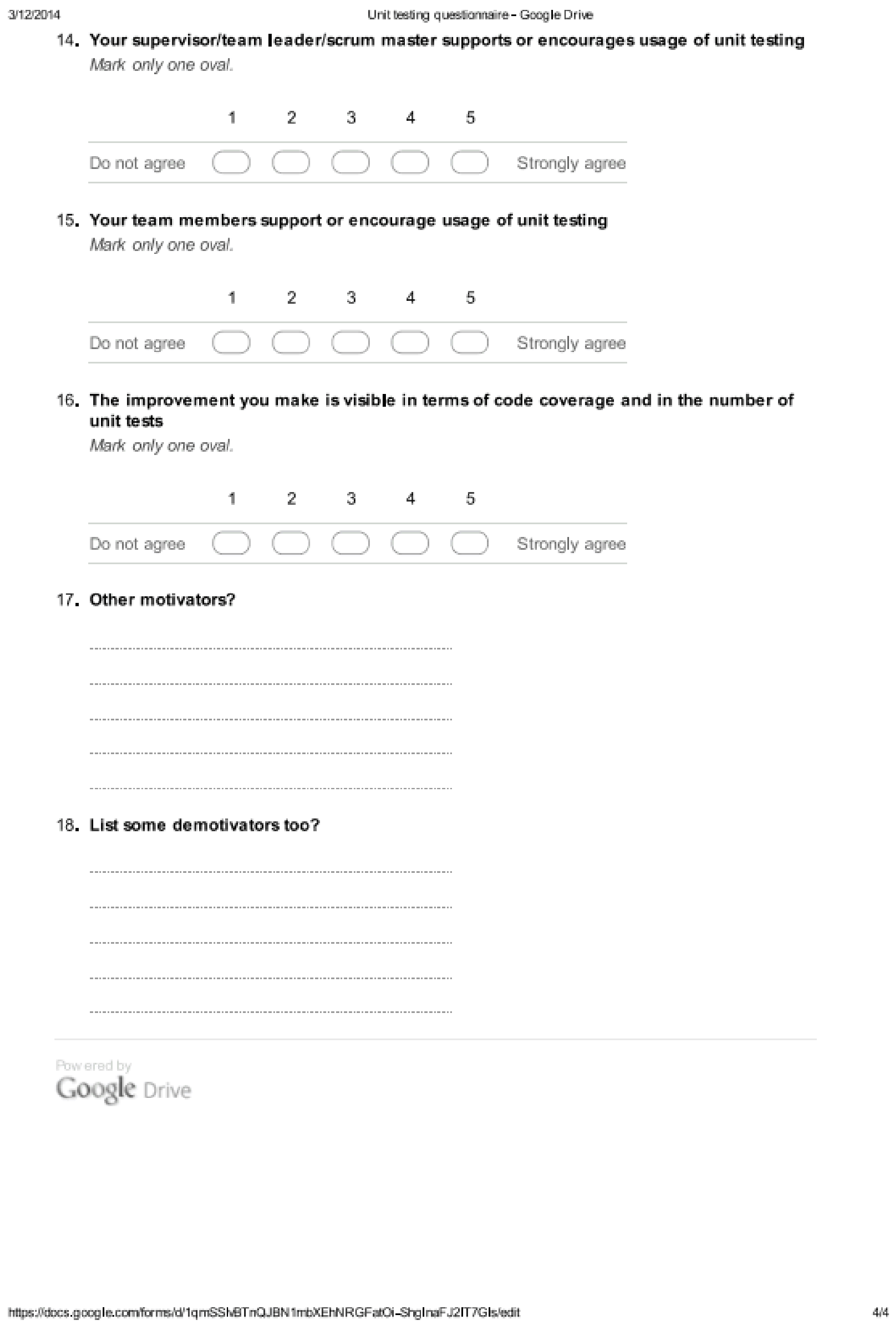
	1	2	3	4	5	
Do not agree	◯	◯	◯	◯	◯	Strongly agree

12. **Unit tests have helped you to find real bugs**
Unit tests have helped you to find real bugs therefore speeding up the development
Mark only one oval.

	1	2	3	4	5	
Do not agree	◯	◯	◯	◯	◯	Strongly agree

13. **Unit testing process has helped to increase the overall quality of the product**
Mark only one oval.

	1	2	3	4	5	
Do not agree	◯	◯	◯	◯	◯	Strongly agree



14. **Your supervisor/team leader/scrum master supports or encourages usage of unit testing**
Mark only one oval.

	1	2	3	4	5	
Do not agree	⬭	⬭	⬭	⬭	⬭	Strongly agree

15. **Your team members support or encourage usage of unit testing**
Mark only one oval.

	1	2	3	4	5	
Do not agree	⬭	⬭	⬭	⬭	⬭	Strongly agree

16. **The improvement you make is visible in terms of code coverage and in the number of unit tests**
Mark only one oval.

	1	2	3	4	5	
Do not agree	⬭	⬭	⬭	⬭	⬭	Strongly agree

17. **Other motivators?**

18. **List some demotivators too?**

Powered by
Google Drive

Appendix 6: The Responses to the Unit Testing Questionnaire Form

Markus Lindqvist Edit this form

24 responses

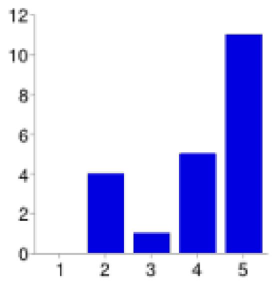
Publish analytics

Summary

What kind of improvements have you found useful for unit testing

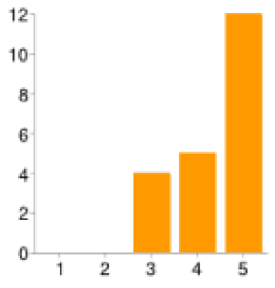
Example of the modularization work done

Product architecture improvement - high level modularization



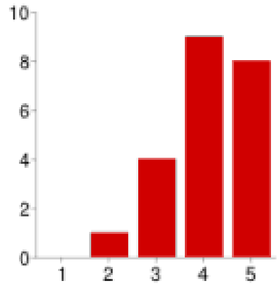
1	**0**	0%
2	**4**	19%
3	**1**	5%
4	**5**	24%
5	**11**	52%

Other modularization work



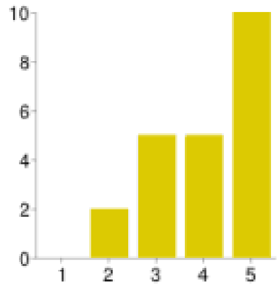
1	**0**	0%
2	**0**	0%
3	**4**	19%
4	**5**	24%
5	**12**	57%

Unit testing education



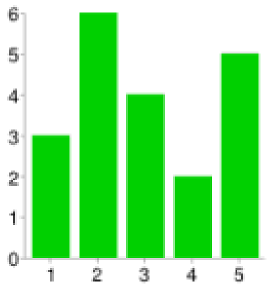
1	**0**	0%
2	**1**	5%
3	**4**	18%
4	**9**	41%
5	**8**	36%

Unit testing framework improvements



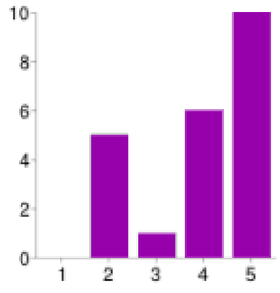
1	**0**	0%
2	**2**	9%
3	**5**	23%
4	**5**	23%
5	**10**	45%

IDE integration of unit testing



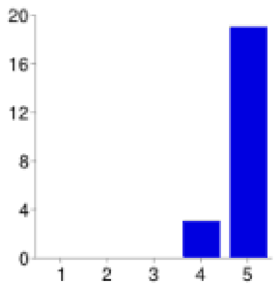
1	**3**	15%
2	**6**	30%
3	**4**	20%
4	**2**	10%
5	**5**	25%

Low-level unit testing improvements



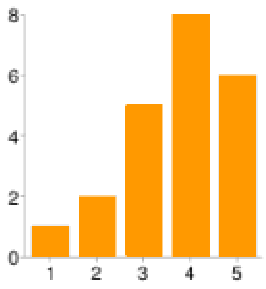
1	**0**	0%
2	**5**	23%
3	**1**	5%
4	**6**	27%
5	**10**	45%

Continuous Integration usage



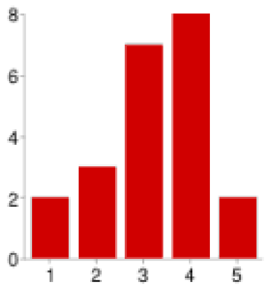
1	**0**	0%
2	**0**	0%
3	**0**	0%
4	**3**	14%
5	**19**	86%

Continuous Inspection usage



1	**1**	5%
2	**2**	9%
3	**5**	23%
4	**8**	36%
5	**6**	27%

Commonly agreed code coverage targets for all development teams



1	**2**	9%
2	**3**	14%
3	**7**	32%
4	**8**	36%
5	**2**	9%

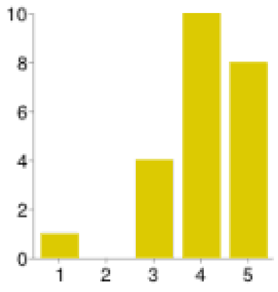
Other improvements that have helped taking unit testing into usage?

Unit tests as part of compilation process. This might be considered to be part of the "Continuous Integration usage" above. However, I think the distinction between local build scripts used by single developer and larger integration systems is valid. At least for me running Unit tests as part of daily development work is more useful than unit tests that are run after submitting code changes for integration. Generally applying TDD practices (whenever applicable).

Unit testing questionnaire

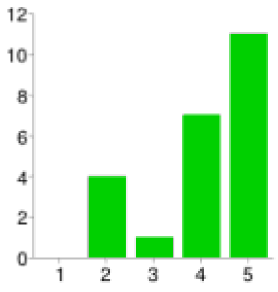
What motivates you to use unit testing?

Unit testing is a design method



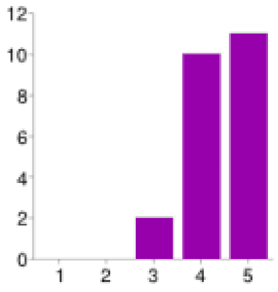
1	**1**	4%
2	**0**	0%
3	**4**	17%
4	**10**	43%
5	**8**	35%

Unit tests have helped you to find real bugs



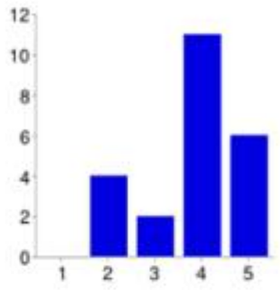
1	**0**	0%
2	**4**	17%
3	**1**	4%
4	**7**	30%
5	**11**	48%

Unit testing process has helped to increase the overall quality of the product



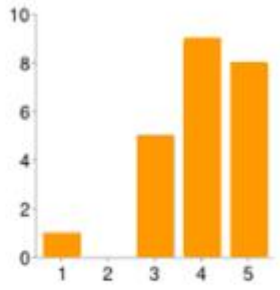
1	**0**	0%
2	**0**	0%
3	**2**	9%
4	**10**	43%
5	**11**	48%

Your supervisor/team leader/scrum master supports or encourages usage of unit testing



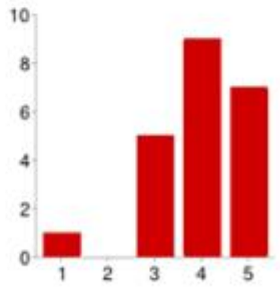
1	**0**	0%
2	**4**	17%
3	**2**	9%
4	**11**	48%
5	**6**	26%

Your team members support or encourage usage of unit testing



1	**1**	4%
2	**0**	0%
3	**5**	22%
4	**9**	39%
5	**8**	35%

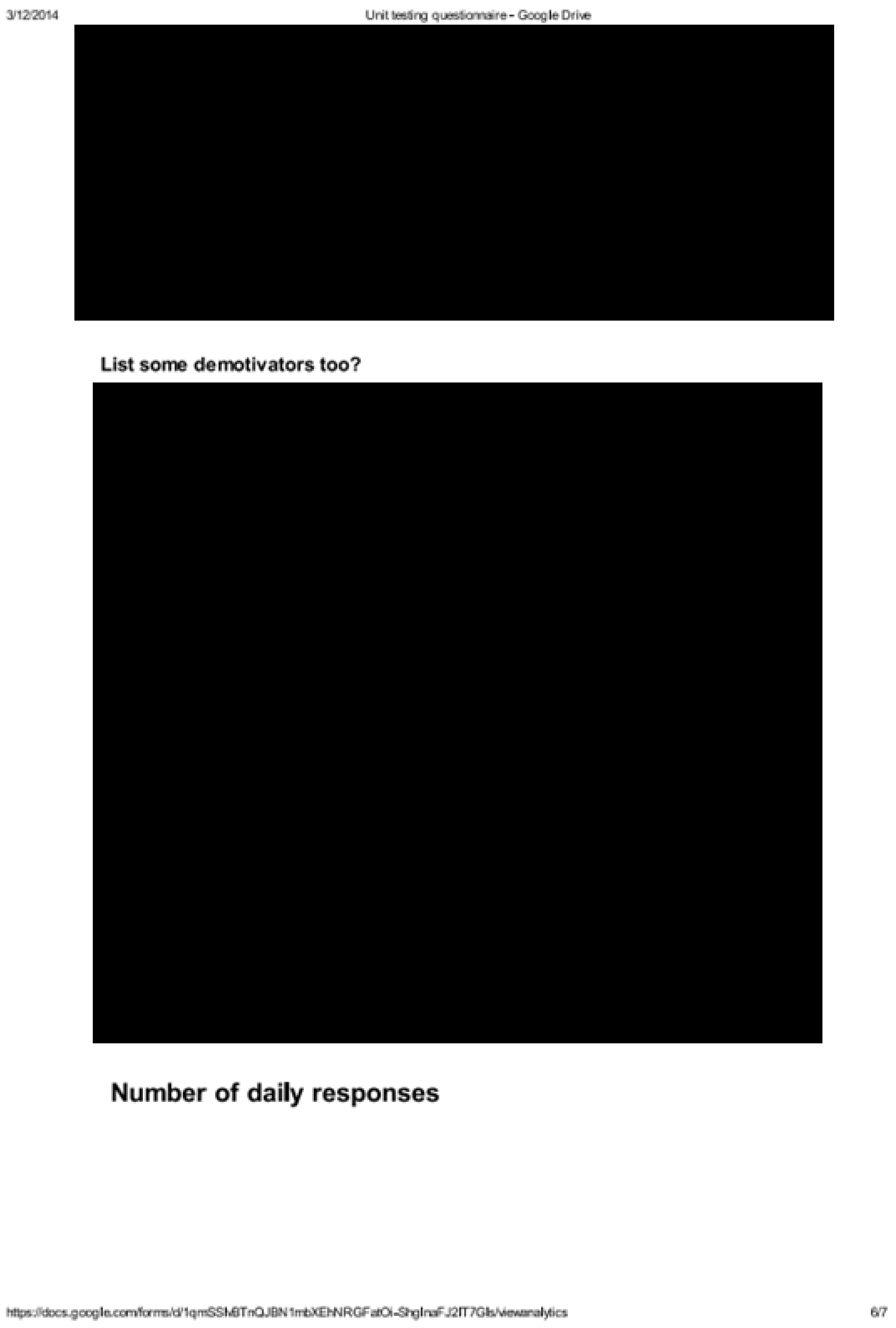
The improvement you make is visible in terms of code coverage and in the number of unit tests



1	**1**	5%
2	**0**	0%
3	**5**	23%
4	**9**	41%
5	**7**	32%

Other motivators?

[redacted]

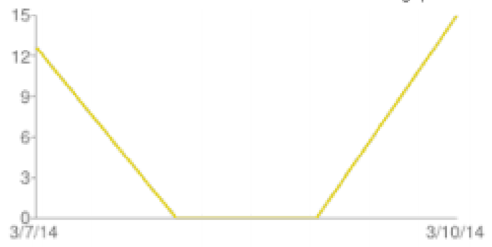


[redacted]

List some demotivators too?

[redacted]

Number of daily responses


15
12
9
6
3
0
3/7/14
3/10/14