Timo Jakku

Increasing Multi-tier Computer System Service Quality through Software Testing

Helsinki Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Master's Thesis

11.04.2016

Metropolia

Tekijä	Timo Jakku
Otsikko	Palvelun laadun parantaminen testauksen keinoin monikerroksisessa tietokonejärjestelmässä
Sivumäärä	54 sivua
Aika	11.04.2016
Tutkinto	Insinööri, YAMK
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Mobiiliohjelmointi
Ohjaaja(t)	Teemu Arvonen, Section Manager Auvo Häkkinen, Yliopettaja

Työn tarkoituksena oli tutkia monikerroksisen Internetpalvelun tuotantokäytössä raportoituja vikoja ja luoda analyysin pohjalta kuva kuinka palvelun laatua voitaisiin kehittää testauksen keinoin.

Tieto vioista on saatu virheraporttien muodossa raportointijärjestelmästä. Virheiden joukosta on karsittu kaikki muut kuin ohjelmistovirheet pois (viat laitteistoissa, infrastruktuurissa jne.). Virheet on karsinnan jälkeen luokiteltu järjestelmän kerroksen, prioriteetin, testaustason ja ISO 25010 -laatumallin mukaan. Näitä luokitteluita on verrattu toisiinsa mahdollisen yhteyden löytämiseksi.

Saaduista tuloksista voitiin nähdä selviä parannustarpeita useissa testaukseen liittyvissä toiminnoissa. Tutkielman tuloksena annettiin suosituksia manuaalisen testauksen lisäämiseksi ja automaattisen testauksen kattavuuden ja vaihtelevuuden lisäämiseksi. Vaikka tuloksista oli nähtävissä, että valtaosa vioista sijoittui järjestelmän keskikerrokseen, ei testauksen keskittämistä tuohon kerrokseen erityisesti suositella, sillä siitä ei uskota olevan vastaavaa hyötyä.

Tulokset antavat käyttökelpoisen tilannekatsauksen kyseisen järjestelmän ongelmista ja kehitystarpeista. Tuloksia ei voida yleistää kattamaan muiden vastaavalla tavalla rakennettujen järjestelmien toimintaa, sillä jokaisessa järjestelmässä on liikaa muuttujia vertailtavuuden esteenä. Tässä työssä hyödynnettyjä metodeja ja prosesseja voi hyödyntää vastaavan tutkimuksen tekemiseen toisesta järjestelmästä.

Avainsanat	Ohjelmistotestauksen laatu, tietokonepalvelun laatu, Internetpalvelu

Author	Timo Jakku
Title	Increasing Multi-tier Computer System Service Quality through Software Testing
Number of Pages	54 pages
Date	Monday 11th April, 2016
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	Mobile Programming
Instructor(s)	Teemu Arvonen, Section Manager Auvo Häkkinen, Principal Lecturer

The goal of this thesis was to study one multi-tier system from the perspective of faults found in production usage. These faults were analyzed in order to find out if any particular level of testing and any particular tier of the system had any statistical connection and thus would reveal the need for an improvement on a certain test activity or level on a certain tier of the system.

The reported faults were extracted in the form of trouble reports from a tracking system. Only software fault related issues were kept for analysis, disregarding hardware, infrastructure and other miscellaneous faults. The faults were categorized based on the system tier, priority, testing level and ISO 25010 quality model. These categorizations were then mapped against each other in order to see if any logical connection could be found.

The analysis shows that there is room for improvement in several areas of testing. The study makes recommendations as to increasing the amount of manual testing and broadening the scope and variation in automated testing. While it was observed that most faults occur in the middle tier of the system, it was concluded that no significant advantage can be gained by targeting that tier in particular.

The result of this study was a useful view into the situation of one particular system and its current problems. However, the conclusions drawn from these results are only applicable to a very limited extent to other multi-tier systems, as there are too many variables in each system for such comparison. However, the methods and processes utilized in the study can be used to extract another set of comparable results from another system.

Keywords	Software Testing, Software Quality, Multi-Tier Computer System

Contents

Abbreviation

1 Introduction 1

2 Structure of the Study 3

3 Multitier Computer System 4

4 Product Development Organization and Processes 5

4.1 Software Development Process 8
4.2 Testing Process 11
4.2.1 Unit Testing 11
4.2.2 Component Testing 13
4.2.3 System Testing 14
4.2.4 Production Testing 16

5 Related Research 17

5.1 Software Testing Related Research 17
5.2 Optimization Techniques 18
5.3 Predicting Reliability 20
5.4 Human Component 22
5.5 Summary 24

6 Analysis Process and Source of Data 24

6.1 Trouble Reports 25
6.2 Issue Classification 26
6.2.1 ISO-25010 26
6.2.2 Test Level Mapping 29
6.2.3 System Tier Mapping 30
6.2.4 Issue Priority 31
6.3 Data Extraction 32
6.4 Data Recording 33

7 Results 33

7.1 Analyzed Issues and Top Level Statistics 33
7.2 Cross-referencing System Tier and Priority 37
7.3 Cross-referencing System Tier and Test Level 38
7.4 Cross-referencing Priority and Test Level 39
7.5 Cross-referencing System Tier and Quality Model Mapping 40
7.6 Cross-referencing Test Level and Quality Model Mapping 42
7.7 Cross-referencing Priority and Quality Model Mapping 44
7.8 Analyzed Issues in Relation to Complete Time Period 45

8 Discussion and Conclusions 47

8.1 Recommendations 47
8.2 Further Observations 50

References 53

Abbreviations and acronyms

ACO	Ant Colony Optimization
ATDD	Acceptance Test Driven Development
GA	Genetic Algorithm
LB	Load Balancer
SLA	Service Level Agreement
SRM	Software Reliability Model
SRGM	Software Reliability Growth Model
UML	Unified Modeling Language

1 Introduction

The age of Internet has brought along a lot of new business. Companies that existed long before Internet was even a concept have now embraced the new era of digital commerce. As is always the case with new approaches and techniques, whether digital or not, a new set of tools and platforms is required when innovations are turned into real life solutions.

The evolution of Internet has seen a vast number of different technical visions and ideas. The growth has demanded the evolutionary process to be extremely rapid. Currently in the realm of online services cloud services are becoming the standard offering, with promises of easy scalability and almost bulletproof robustness.

While the evolution of cloud and other technologies marches forwards, many online services are still heavily relying on a bit simpler approach called multi-tier architecture. The most typical multi-tier system consists of three tiers, or layers.

The first tier hosts the user interface related functions. For online services this usually means hosting the actual web pages which the end customers engage with their browsers. The second tier hosts transaction and business logic, which in online systems means handling for example a web store transactions, customers buying goods and services from the provider. The third tier hosts the data storage services, meaning in online context for example customer information and product inventory databases.

Due to the division of functions between the tiers the system layout then presents itself. The presentation tier communicates with the logic tier, which in turn then communicates with the data storage tier. This gives then the tiers also the order of front, middle and back, respectively. It is typical that all of the three tiers run software made by different providers. The web server software in the front tier can come from one provider, the transaction logic middleware from another provider and the database and data warehousing software in the

back tier from a third provider.[1]

One of the cornerstones of software development is testing. While testing cannot usually cover 100% of the possible scenarios, it can give reasonable level of assurance that the software fulfills the set requirements. There are many types and levels of testing, and several definitions for all of them. While they all have their uses, and are important, the question arises: are they all equally important? None can be totally neglected, but are there benefits in focusing more on one than the other.

The question becomes even more interesting when the nature of the multi-tier environment is taken into account. Each tier has to function in order for the complete system to be usable. While in theory all tiers can have faults, are there in reality more faults in one than the other?

While it is clear that all the tiers have to function and the software in the tiers have to be subjected to all the levels of testing, could there be benefits in emphasizing some test level for some tier?

In order to give an answer to that question, data has to be gathered regarding where faults occur in such a system and which level of testing addresses best those types of faults. Once the data has been turned into numerical format the values can be directly compared against each other. It can be concluded from the figures whether there are differences in the frequency of faults between the tiers and also if any particular level of testing seems more efficient with any of the tiers.

This study is based on one actual multitier computer system. The system is developed and maintained by the provider, and the system capabilities are offered as a service to customers. This in essence means that the provider is responsible for all aspects of producing the service.

When a customer encounters an issue with the system they will create a trouble report.

The service provider will then use this report as basis for corrective action. These reports will be used as data for this thesis. Data from reports issued during a six month period will be extracted and analyzed in order to create a view which levels of testing are in need of improvement. Recommendations will also be made regarding the nature of improvements.

2 Structure of the Study

The study starts by explaining the multi-tier computer system. Then it moves on to the product development organization and processes and covers related research. Chapter 6 explains the analytical process utilized in the study, followed by the results of the analysis in Chapter 7. Finally, discussion and recommendations are presented in Chapter 8.

3 Multitier Computer System

In this section the basic system layout and working principles of a multitier computer system are explained. The multitier computer system has three separate distinct tiers, front, middle and back (Figure 1).

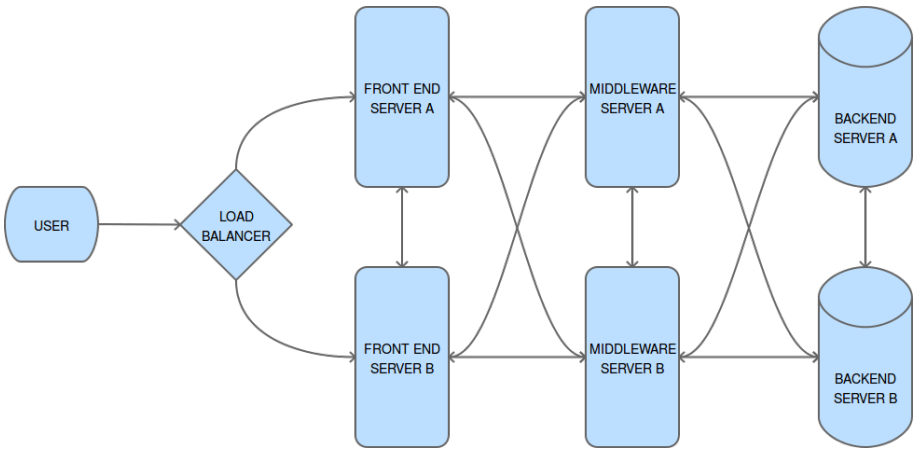

Figure 1: Multitier Computer System

The front tier hosts the user interface on a web application server platform. The middle tier hosts the business logic and functionalities on a middleware application server platform. The back tier hosts data storage and other backend functions.[2]

All tiers are duplicated (A and B servers in Figure 1). In reality there could be more servers on any given tier. The servers are interconnected from A to A, A to B, B to A and B to B, so that loss of one connection will not cause operational degradation. The duplication is done for redundancy and for load sharing. In principle the system should remain fully operational even if each tier suffers catastrophic failure of one server.

During normal operation, the load balancer in front of the system is sharing the load between the front end servers in 50%/50% manner, so that the load is evenly spread. If a server would seize to function, the load balancer would direct 100% of the traffic to the remaining server. The servers are dimensioned in such a way based on the usage statis-

tics that the customer will not experience any performance degradation if one server is lost from a tier.

4 Product Development Organization and Processes

In this section the product development organization and used processes and workflows are explained. Developing new functionality as well as maintenance work is also covered.

Product development and operation of the production environment is organized based on the different functional roles (Figure 2). Roles and responsibilities are divided among the groups as follows:

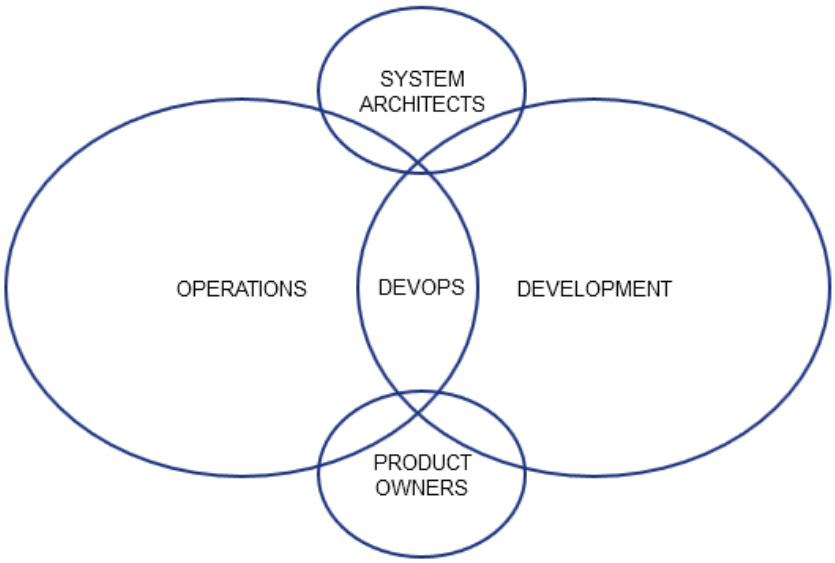

Figure 2: Product development and maintenance organisation

- System architects are responsible for the overall design of the system, regarding both software and hardware. System architects are also responsible for the overall functional design of the system.
- Members of operations are responsible for keeping the production environment operational, as well as handling customer requests related to day to day activities in the system.
- Members of development are responsible for planning the implementation of the

software and implementing both software and related tests.

- Term devops is combination of development and operations. Members of this group are responsible for ensuring that the introduction of new functionalities and improvements to production environment is seamless and that all supporting tools and functions are in place.
- Product owners are responsible for division and organization of work items, as well as communicating with all relevant stakeholders.

All test levels that are in the scope of this thesis are handled by the development group. The development group is organised into teams (Figure 3).

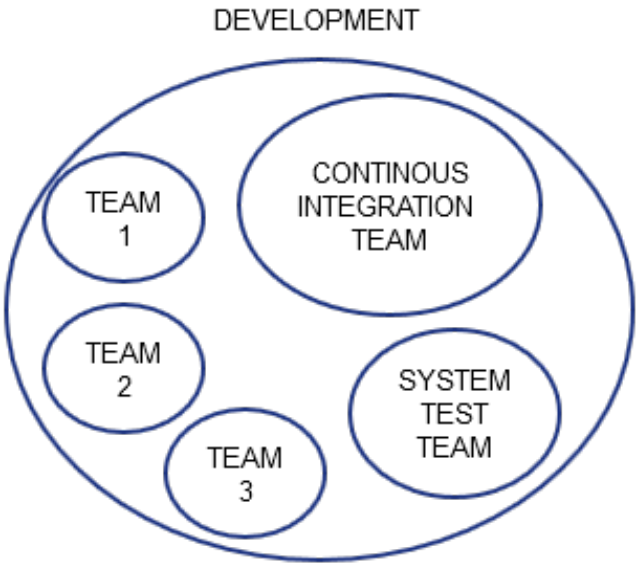

Figure 3: The development group team division

Most of the teams are comprised of specialists of all the related tiers in the system, meaning that a team of this type can work independently with a functionality that involves all three tiers. In addition to that type of teams there are two supportive test teams. The two supportive test teams are continuous integration team and system test team.

The responsibilities of the continuous integration team are:

- Operate and maintain continuous integration environment
- Oversee the quality of the component test case material

- Issue rules and guidelines regarding test structures
- Develop test tooling
- Report findings and issues to relevant stakeholders

The responsibilities of the system test team are:

- Operate and maintain the system test environment
- Develop and execute the system test scenarios
- Develop and maintain the monitoring system for the system tests
- Develop test tooling
- Report findings and issues to relevant stakeholders

The two teams collaborate regarding test tooling and aligning procedures, as well as with other teams when test related assistance or guidance is required.

4.1 Software Development Process

The software development process is divided into two tracks, feature track and maintenance track.

The feature track (Figure 4) is the process for delivering new functionalities or improvements.

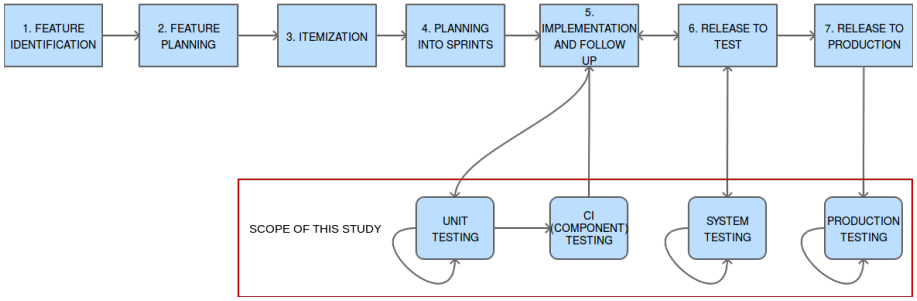

Figure 4: The development process of the feature track. The scope of this study outlined in red.

The feature track process contains the following steps:

1. Feature identification. The need for new feature or functionality is identified and described in broad terms.
2. Feature planning. All the requirements are gathered and the new implementation is planned in detail.
3. Itemization. The plan made in step 2 is broken down into logical work items.
4. Planning into sprints. The work items are worked into the sprints of the development teams.
5. Following up the sprints. Product owners follow up with the team the progress of the development of the item, and communicate to other stakeholders when needed.
6. Release to system test. The feature or improvement is ready to be released to system testing, with all the supporting items, including documentation.
7. Release to production. The new functionality or improvement is delivered to production.

The maintenance track process is meant for correcting issues reported by the customer

and improving known long running issues. The process follows the same steps as the feature track process, except for steps number 1 and 2, which are the following:

1. Problem analysis. The handled issue is analysed to identify the root cause.
2. Correction planning. The correction for the problem is planned. After step 2 the aforementioned process is then followed through steps 3-7.

In both tracks testing takes place in steps 5, 6 and 7. During step 5, Implementation and Follow Up, unit tests are created as part of the implementation (Figure 5).

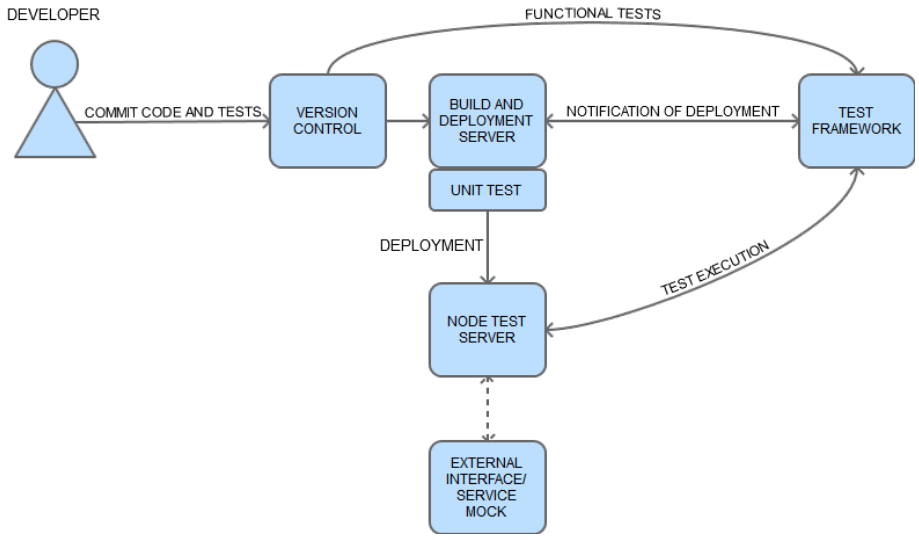

Figure 5: Implementation and follow up

Tests are executed when the code is built to a deployment package for the continuous integration environment. Alongside unit tests, component tests are developed and delivered to continuous integration environment, to which the built software package is then installed and component testing is carried out. Also regression test cases are executed to verify that the newly developed functionality has not impacted the existing functionality.

During the implementation phase the developers store their work in a version control system. A version control system offers many useful features for software development. The main function in this process for a version control system is to act as a central storage for the development teams. Once the work is stored into the version control system, the build and deployment server recognizes the change in the version control and builds a

new deployment package, which is then automatically deployed on to a node test server. At the building phase the unit tests are also ran.

As the test scope at this phase is to verify only the functionality of the particular tier (for example middle tier), the other tiers are simulated, and the interfaces are mocked. Mocked interface follows the correct interface specification, but does not perform the actual function "behind" the interface. For example, if the front end tier would query the status of the middle tier, the mock will reply “OK”, without the actual middle tier being present. From the perspective of the front end tier everything would seem normal.

Once the new build has been deployed, test framework will execute the component tests for the particular tier. Also these tests are stored in the version control system. The test framework will monitor the execution of the tests and generate a report of the results. Developers can then use the report as feedback and act accordingly.

When the implementation is done, and both unit tests and component tests are passing, the process moves to step 6, Release To System Test. At this stage the software package is deployed to system test environment, and tested as a part of the complete system test scope.

Figure 6 has been simplified to only show one node per tier, while in the actual system there are two or more servers per tier.

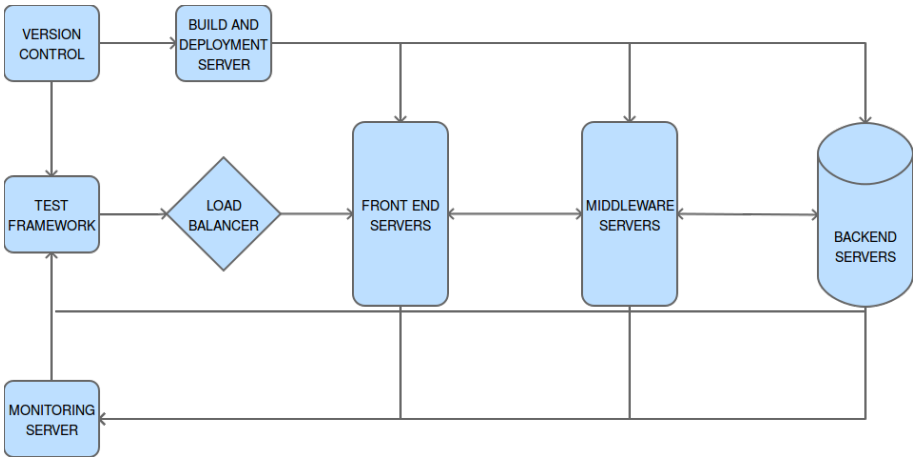

Figure 6: The system test process structure and system layout

The process begins with building deployment packages to all the nodes in the system, and deploying them. The deployment is semi-automatic. It is started manually by a push of a button, but the execution itself is automated. Once the software is installed and the system is operational, test execution will be started in the same semi-automatic fashion as the deployment. The test framework is monitoring the execution of the test scenarios from all the servers, and is using the data to control the execution. Data regarding system metrics (CPU load, memory usage etc.) is gathered by the monitoring server. The data is used to follow up the performance of the system in real time. All the information is stored also for later usage.

Once system testing has been carried out, the software package can be released to production in step 7. Tests are carried out immediately following the deployment to production to verify the deployment. Afterwards testing is carried out to monitor the system and to gain information about the performance of the system.

4.2 Testing Process

The four testing levels (unit, component, system and production testing) are described in detail in the following sections. The complete testing process consists of these four levels. As faults end up in production system as well, it is obvious that one or more levels can be improved.

4.2.1 Unit Testing

Most code oriented level of testing is known as unit testing. In the context of this thesis the term unit testing is following the principles described here, disregarding other descriptions or definitions made elsewhere.

Unit testing is code level testing approach, where tests are added when the actual appli-

cation code is being developed. In practice this means that when a new functionality is added to the code base, tests are added as additional code to make sure that the added functionality works as expected.

For example, a developer adds a new function called SUM which should take as input two parameters, integer A and integer B, add A and B together and return result integer C. After developing the function the developer could add a unit test case in which integer values 6 and 3 are passed to the function and the return value is then checked. If the return value is 9, the test will pass, with any other value the test will fail.

In addition to giving assurance to the original developer that the developed code fulfills the specification the unit tests also help when existing code is being modified. If the purpose of the modifications is not to change functionality, unit tests will assure that the functionality is unchanged. Using the function SUM the unit test would protect against unintentional change from adding to subtracting the integers. If that were to happen, the unit test would pass integers 6 and 3 to the function SUM and receive integer of value 3 in return. As the test is expecting value 9, the test would fail.

If the intention is to change the functionality the unit tests will give information regarding the original functionality. It will also demand modification to the unit tests so that they will reflect the new functionality after the changes. Using the function SUM the developer changes the functionality so, that the function takes three parameters as input, integers A, B and C, adds them together and returns the result as integer value D. After the change the original unit test case will fail to run, and the developer will need to modify the test case as well.

These tests can be executed by the developer while doing the actual development. They are executed again once the software is being compiled for actual use in the target system.

4.2.2 Component Testing

Once the code is developed and tested, it can be compiled and installed into the target system. Depending on the system in question, in order to be able to verify the wanted functionality some initial setup procedures might be needed. The user interface component will be used as an example, while the same basic principles apply to other components as well.

For example changes have been made to the customer login functionality and it needs to be verified that the functionality fulfils the specifications. The specification states that the user can login with the correct username and password, that if three erroneous attempts with incorrect password are made consecutively the account will be locked, and that if account is locked user cannot log in even with the correct password.

To be able to test this, a user account is needed, so the user account data is put in place. The test scenario will follow the specification. Successful login is verified, three erroneous attempts are made and not gaining access is verified, as well as attempting to log in while account is being locked and not gaining access is verified.

When the change is introduced, the testing is carried out manually for the first time. However, it is likely that several rounds of testing are needed during the development of the functionality, and also as the functionality will be regarded as legacy after release to production, the regression testing also needs to be addressed. For these reasons the tests will be automated as part of the development process, and will first serve the need of the development process and afterwards will be used for regression testing.

In addition to verifying the specified functionality, test cases are added also to cover for unwanted scenarios. In the example above, an unwanted scenario could be that user A can log in to the system by using the password of user B.

For the development and regression testing purposes the automated component test

cases are put in to and executed by continuous integration environment. The continuous integration environment will monitor code commits to version control system, from which newly committed code is built to deployable packages and installed onto the testing platform. If installation has been successful, the initial configuration and test data provisioning is done, after which the system is ready for testing.

Tests are executed automatically and results are reported to all the members of the development teams, which can take corrective action if faults are reported.

4.2.3 System Testing

In the other test levels the tests are carried out in non-production like environments, regarding hardware, data volumes and transactions. At system testing level all software components are installed and configured to make up the complete end-to-end system. The platforms and hardware are similar to production, and test data volumes represent the amounts in the production system.

System testing can be divided into three parts: production traffic testing, stability traffic testing and robustness testing.

In production traffic testing test scenarios are based on actual user behavior in the production system. The amounts and frequencies of the utilization of the functionalities and features represent the actual usage of the production system. For example, if in production functionality A is used 15 times per hour on average and functionality B is used 2 times per hour on average, this must be reflected on the test scenario in order to get usable, realistic test results. Also the number of parallel test cycles, which each represent one user, is based on data from the production system.

Once the statistics from the production environment are analyzed, it is possible to compose the test scenario from separate test cycles. The cycles consist of logical set of tests

of the functionalities that one user would actually use, with given set of test data. For example, cycle one could consist of tests for functionalities A, B and C with data set X. Cycle two could consist of test for functionalities B, D and F, with data set Y. The cycles are planned to have run time of less than one hour, so that the test scenario can be ran for an arbitrary time, and when discontinued, it will run to a controlled stop within an hour.

In addition to monitoring test pass rate, several system metrics are being monitored in real time, such as CPU, memory and IO utilization during the tests.

Regarding the user login test case example used earlier, in the system test level test scenario will include several parallel login test cases in order to see that the system can support the required amount of simultaneous user logins. In addition to the aforementioned hardware level metrics such as CPU usage, response times and error logs will be monitored to reveal any possible problems during the execution of the test scenario.

The results are graphed and compared against previous runs in order to see improvements or possible degradations in system behaviour.

The stability traffic testing is used to assess the stability of the system. While some of the possible faults have been picked up by unit and component testing, some of the faults may only manifest themselves while the system has been running under load for some time. The problem is to define how long is “some time”. While there cannot be a definite answer for this (it can not be foreseen how long it will take for faults to manifest), it is possible to try to speed up the process of exposing faults.

Some of the faults are caused by memory leaks. They are caused by code allocating some memory when it is executed, but not releasing it ever. When the code is executed enough times, the memory will be exhausted and execution of the code will fail. Depending on which part of the system this type of fault is, it could potentially bring down an entire server. Since the manifestation of the fault of this type is based on the times the functionalities are used, the probability of exposing this type of faults can be increased by increasing the number of the execution of the code, in other words, increasing the load. So in order

to speed up the process, as well as expose other types of load based faults, the load is increased from the actual production usage to roughly two folds, after a ramp up period.

The robustness of the system is tested by using special scenarios, which introduce planned faults to the system, for example restart of a server or a network failure between the components. During a robustness test a normal system test scenario is executed in the system, and once the fault is introduced to the system, the system behavior is observed. Since the basic functionality has already been covered by earlier test levels, the focus here is to observe the recovery and alarm functionalities. Different types of scenarios target different aspects of the whole system setup. For example scenario introducing a network fault targets the software behavior when another component becomes unreachable, as well as it targets the thoroughness of the network topology planning.

4.2.4 Production Testing

After software has been deployed to production the possibilities for testing are significantly reduced, but certain things can still be tested and verified. While all forms of disruptive testing are out of the question, testing can still be carried out from end user perspective by utilizing test user accounts in the production system.

Production testing can be divided into two types: deployment verification and system surveillance.

Deployment verification is carried out after new software version has been deployed into production environment. The purpose is to verify that all the components were correctly deployed and configured. A predetermined set of tests are carried out after the deployment process has finished. The set contains a static part that is always carried out, for example, testing the aforementioned login functionality, and a release specific part, focusing on the new functionalities.

The deployment verification is carried out because although all the system functionalities and the deployment process itself have been verified prior to this point, even a minor mistake during the deployment to production could potentially cause severe service impact if it would go unnoticed even for a short period of time.

System surveillance related testing is carried out to follow up on the end user experience. While system alarms and internal metrics are constantly monitored, testing from the actual end user perspective should yield realistic information regarding the user experience. A test set of the most used system functionalities is constructed and is executed automatically at scheduled intervals. Response times are monitored and if degradation, or in worst case, service unavailability is observed, corrective actions can be taken. By graphing the response times, possible trends can be observed, and releases can be compared to one another.

5 Related Research

In this section a review is presented about the state of research related to software testing. Identified key points are emphasized and found best practices are discussed further. First the general state of software testing related research is reviewed. On the second part certain software testing optimization techniques are discussed. On the third part methods for predicting software reliability are reviewed. The fourth part consists of matters related to the human aspect of testing. At the end of this section the findings are summed up.

5.1 Software Testing Related Research

First the importance of software testing related studies and research is to be noted, as estimations about the percentage of software testing of all the effort used in a software development project vary from 40 to 80 percent. [3, 4] It is obvious that companies and organizations seek means to reduce that number and on the other hand want to make sure that such an significant effort will be made in the most efficient possible manner.

The second item to be noted is that no single existing body of work that would cover the subject of this thesis has been identified. This perhaps is due to the combination of items in the technology problem handled in the study which makes the subject very case specific and thus makes it a difficult subject for pure research. However, since testing efficiency and optimizing testing has been and continues to be of great interest in both academic and business domains, significant amounts of research can be found regarding these subjects in general.

Software testing methodology has been studied quite extensively from different points of view. Unfortunately a lot of the research suffers from the same basic problems, such as the lack of standardization regarding the comparison methods, the differences in the pieces of software used to evaluate the methods, different classifications and specifications of the methods and software bugs.[5]

Several attempts have been made in the information technology communities to establish common set ups in order to align the testing methodology related research and to yield more comparable results. These initiatives in general call for common classification of the test methods, detailed description of the target software used for the evaluation, standardized evaluation system and standardized structures for recording the results.[6]

However, the situation has not generally improved greatly during the last couple of decades, as the earliest comments that were found were made in the late eighties and most recent initiative to remedy the problems during the current year 2015.[5, 6]

5.2 Optimization Techniques

As the purpose of this study was to find ways to improve the quality of a service through means of software testing, in this section two (ant colony optimization and genetic algorithms) optimization techniques are discussed. Among other things these techniques can be utilized in optimizing software testing. For example better execution paths and optimized test data sets can be achieved by utilizing these techniques correctly.

Like many other fields of research, also computer science has taken leaves from natures book. Some of the lessons learned from mother nature have also been applied to the optimization of software testing. Among the most studied approaches are genetic algorithms and swarm intelligence.

One of the best known swarm intelligence theories is the so called ant colony optimization, abbreviated ACO. This approach is based on the biology and behavior of real life ants. It has been observed that ants have a system for indirect communication, which makes it possible for the ants in the colony to find the shortest routes between their nest and the located food sources. The system is based on the pheromones that the ants deposit along their route. Other ants can smell the pheromones and they prefer traversing on paths with higher pheromone concentration. The key to the optimization function is the evaporation of the pheromones over time. The ant that found the shortest path to the food source will also have the shortest path back to the nest and will thus leave the strongest pheromone trail. When other ants are leaving the nest, they will most likely select the route with the strongest pheromone concentration and as other ants traverse on the shortest path, the pheromones they emit will increase the likelihood of the path being selected by others.[7]

ACO and other optimization algorithms and methods are often demonstrated by using a so called "travelling salesman problem", abbreviated TSP. TSP is a basic route optimization problem, where the target is to solve the shortest possible route for the salesman to travel between several cities. Route optimization is useful for many applications, but for the domain of software testing the basic idea needs to be adapted to suit the needs of the problem in question. The equivalent of the real life map can be the source code of the software to be tested, a control flow diagram or some other data structure which can be utilized as a graph depicting the problem. Certain variables as the population of the ant colony, the initial pheromone values and their evaporation rate will be set by the entity solving the problem. When the initilization has been carried out, the actual optimization is then done in iterations, and the pheromone levels are adjusted accordingly for the next iteration. After a number of iterations (the number of iterations that will yield optimal solution depends highly on the other factors) the optimal or near-optimal solution should emerge.[7]

Genetic algorithms, abbreviated GA, are found in nature on an even more fundamental level than ACO. GAs are based on the idea of chromosomes and their evolution. First a population of possible solutions is created at random. Each of the possible solutions is evaluated by using a fitness function to see how well it fits or solves the set problem. The more suitable solutions are selected for crossover where genes from two chromosomes are exchanged and two new chromosomes are created. Mutation operation can also be used to add variations to the genes. Each iteration of the population represents a generation and iterating is stopped once required fitness level is reached.[8]

As with ACO, GAs can be utilized in the field of software testing in many ways, for example in searching execution paths or generating test data. The key element is the fitness function, as it will measure the feasibility of the solution proposed by the algorithm. The fitness function could for example measure how complex an execution path a set of variables proposed by a chromosome can generate.

It is important to notice that the solutions will try to satisfy the fitness function and not the real world problem that the fitness function should represent. It may happen that the fitness function measures well the feasibility of a solution when the proposed solution contains values that are realistic regarding the real world problem. But the same fitness function can produce even higher results with unrealistic values, if no safeguards have been set. For example, if the fitness function measures the complexity of an execution path and gives high complexity values to a loop mechanism in the software code, the algorithm may suggest a set of test data that will produce infinite loop in order to create the highest fitness value. In a similar way the chromosomes should not contain information that is not relevant to the problem, as the unnecessary information slows down and potentially skews the process.

5.3 Predicting Reliability

As the old saying goes, a clever person can solve a problem, while a wise person can avoid it. While avoiding software bugs in the real world might be a tall order, methods exist

to at least predict the reliability of the software beforehand. This information can then be used to steer the entire testing process in order to improve the results.

So called software reliability models are designed to help to estimate the reliability of software, or in other words, estimate the probability and frequency of encountering a software fault in a particular piece of software.

Software reliability models (abbreviated SRM) and software reliability growth models (abbreviated SRGM) are in general either analytical or data-driven. Analytical models are based on certain assumptions regarding software faults and software development processes and utilize probabilistic mathematical approaches, such as non-homogeneous Poisson process to predict the occurrences of faults. Data-driven models are based on the data gathered during the software development regarding the fault that have been identified during testing. This data is then used to estimate the number of faults that are yet undiscovered.[9, 10]

Genetic algorithms can be applied to some of the models as well, as the models can be "trained" or optimized based on the past data to be able to predict the faults more accurately.[10]

Another way to steer the testing effort is to do a risk analysis of the software in question. One method for this approach is to base the analysis on UML (Unified Modeling Language) models of the software under development. The modeled use cases and the derived sequence diagrams are analyzed and risk factors are calculated for each component and connector. After the calculations a list of critical components and connectors can be created and the overall system risk factor can be calculated. [11]

On the other hand, the observed software reliability can in turn be used to assess the reliability of the selected software testing methods and approaches. Obviously if some failures occur during the usage of the software it is clear that the testing does not cover all aspects of the software and its possible use cases. The reason for failing to reveal a fault in the testing level may be due to the used testing approach. If the testing is focused

for example on boundary testing, it might fail to notice a software failure that occurs when a user input from a valid range is used. For example, if the input field should accept integer values between 1 and 100, a tester might conclude that most likely failures will happen with a negative value, zero, a float value, or a value above the range, but if a failure happens when value 92 is used, it might not be caught at all during testing. [12]

While testing the boundaries is very useful in securing the reliability of the software, in the real world usage most failures occur when users are actually trying to use the software as they are advised. It might even be argued that if a user is deliberately trying to cause a failure, the resulting event should not necessarily be called a failure, since it was the desired result.

5.4 Human Component

While the methods and tools mentioned above provide ways of optimizing testing, there is still perhaps a lot to be gained by focusing on the most vital component of all, the human. After all most of the software created today is made by humans for humans, and as such usually the testing needs to be planned and in some way or another also carried out by humans.

As with most activities, there are also several ways to organize the software testing activities. Some organizations use only dedicated testing teams (or subcontract the job). Some organizations put the testing responsibility mainly on the people who develop the code as well. Some organizations have a mixture of both (dedicated testing teams for some testing activities, some testing activities to be carried out by the code developer teams).

As mentioned earlier in Chapter 4, in the organization where the work for this thesis was carried out, the testing responsibilities are divided. While this division serves a purpose, it does have it shortcomings. In the following some of the issues that have been observed both in the client organizations as well as elsewhere in the industry are listed:

1. Lack of documentation. Due to changes to the customer requirements and high velocity of development, the documentation often lags behind the curve. This causes confusion and possibly erroneous testing in a separate testing team (team that is not directly involved in the actual development of the code). In addition it increases the overhead, as time is consumed in finding out the missing details and organizing extra meetings in order to clarify issues.
2. Limited access to colleagues. Due to the problem mentioned in issue 1, often the need arises to clarify matters directly with the members of the development team. However, this might prove out to be difficult, as the team might be located in another place or another country even, in a different time zone. Especially in the case of a different timezone, the only possible communication channel might be email, which then might postpone the solving of a problem until the following day at the earliest.
3. Poorly written trouble reports. Similar to issue 1, trouble reports lacking detailed information about the observed defect or failure will slow down or cause erroneous reproduction of the fault and may lead to erroneous verification of a fix for the trouble report. This is later discussed regarding the trouble reports used as source material for this thesis.
4. Lack of information regarding non-functional requirements. This has it's roots in issue 1. If non-functional requirements are not understood or are unclear, the verification gives misleading information.
5. Lack of information regarding the needed environment setup. This can be combination of most of the issues mentioned above. If the functionality under test, the possible trouble report and the non-functional requirements are not correctly understood, the test environment setup will most likely be incorrect for the planned verification and thus yield incorrect results.

As can quite easily be seen the main problems are the lack or quality of documentation and communication. One possible way to alleviate those problems between code developers and testing personnel is to utilize so called acceptance test driven development (ATDD). Put simply, in the development process utilizing ATDD, the tests are created first and the software second. The tests act at the same time as documentation and acceptance criteria. Another bonus of this method is to actually guarantee that the needed test scope will be reached, since if the tests are created after the code is ready, it is easier to neglect

testing. According to surveys this approach has been gaining traction during the past decade.[13]

5.5 Summary

While software testing is not exactly a new concept, it is still riddled with some of the problems that were observed already a long ago. As can be seen from some of the most current initiatives, the standardization of terms and methods is still lacking, and that a call for more controlled and reproducible experiments exist . The purely theoretical approaches have their intellectual merits, but are often plagued with either such an elaborate set up procedures or such an limited scope, that they are nigh on impossible to utilize in practice. While the models work in principle, the used example and test programs are exceedingly simple in comparison to software used in modern multi-tier systems.

As stated earlier, no single body of work that would address the issue being handled in this thesis has been identified. That is not surprising, and in all likelihood if corresponding piece of work were to be found, it would not be directly applicable due to the numerous variables that make most of software projects different to one another.

6 Analysis Process and Source of Data

In this section the analytical process is explained and the source of data is discussed. The process was developed based on the available source data and the attributes that can be extracted from the data. The data consists of trouble reports issued over a period of six months.

6.1 Trouble Reports

This study was carried out in relation to one actual multi-tier system. The main source of information was the customer issued trouble reports, which revealed the defects that had been detected only by the end user in the live production environment.

A trouble report contains a set of preformatted data elements, which include among other things the priority of the issue and the component in which the fault was observed. Due to the virtually infinite number of possible reasons for a fault, the actual fault is described in free-form text. This initial description of the fault is usually done by the person handling the communication with the customer who is reporting the fault.

Once the issue has been taken up by a specialist of that particular component in which the fault was observed, the specialist will then investigate the issue and provide a more detailed and technical description of the fault. At this stage it may also happen, that it is discovered that the initial analysis has been incorrect and that the fault has actually been originated in another fashion and/or in another component than was initially though.

While the issue is being worked on, stakeholders can add information to the reports, in the form of free-form text and various types of attachments, for example log files.

Once the solution for the issue has been determined, the person or group working on the issue will describe the technical solution in free-form text. After the solution has been implemented and it is ready to be delivered to production, the report can be closed.

The reports were selected based on their creation date. Only reports issued after the last major changes to the system were studied, as changes in the system setup will render older reports obsolete. Introduction, removal or change of hardware or software platform, or change of system topology were considered a major change. Normal software platform and application updates were not considered as major changes. After eliminating infrastructure related issues (for example network disturbance, hardware failure etc.) the

remaining reports reveal lack of testing regarding that specific issue experienced by the customer.

6.2 Issue Classification

The issue in each trouble report selected was classified in order to extract the information needed to answer the questions set for the study.

The classification system is based on four elements: the quality model based on ISO/IEC 25010:2011, test level mapping, system tier mapping and issue priority. All four elements are described in the following sections.

6.2.1 ISO-25010

In order to have a comparable base for the classification of faults, the quality areas observed in the classification system are based on the ISO/IEC 25010:2011 standard.

Hierarchically ISO/IEC 25010:2011 standard is the work of ISO/IEC JTC 1 (joint technical committee in the field of information technology) and subcommittee SC 7 (software and systems engineering).

Full descriptive name of ISO/IEC 25010:2011 is Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models.

The standard defines the following:

"a) A quality in use model composed of five characteristics (some of which are further

subdivided into subcharacteristics) that relate to the outcome of interaction when a product is used in a particular context of use. This system model is applicable to the complete human-computer system, including both computer systems in use and software products in use.

b) A product quality model composed of eight characteristics (which are further subdivided into subcharacteristics) that relate to static properties of software and dynamic properties of the computer system. The model is applicable to both computer systems and software products."

In the study the option b, product quality model, was utilized.

As stated above the product quality model consists of eight characteristics and their subcharacteristics. The characteristics are listed in Table 1.

ISO 25010 characteristics	
Characteristic	Subcharacteristics
Functional suitability	Functional completeness Functional correctness Functional appropriateness
Performance efficiency	Time behaviour Resource utilization Capacity
Compatibility	Co-existence Interoperability
Usability	Appropriateness recognizability Learnability Operability User error protection User interface aesthetics Accessibility
Reliability	Maturity Availability Fault tolerance Recoverability
Security	Confidentiality Integrity Non-repudiation Accountability Authenticity
Maintainability	Modularity Reusability Analysability Modifiability Testability
Portability	Adaptability Installability Replaceability

Table 1: ISO 25010 product quality model characteristics

For detailed description of each characteristic and subcharacteristic please refer to the actual ISO/IEC 25010:2011 standard and related documentation.

For each issue in the selected trouble reports the appropriate characteristic and subcharacteristic are selected and marked down. This will provide information regarding which quality areas have shortcomings regarding testing and also provide standardized base for comparison.

6.2.2 Test Level Mapping

Test level mapping is used for signing a certain type of fault to a certain test level. The mapping includes other type of issues besides pure software code faults, for example procedure related issues causing faulty deployment of the software. The mapping for test levels unit, component, system and production are explained in respective order.

1. Method level code implementation fault. The fault is caused by mistake made in the basic level of code implementation. For example incorrect implementation of an arithmetic method which when given correct input values will return incorrect return value. This type of fault is mapped to unit test level.
2. Functional level code implementation fault. The fault is caused by incorrect or incomplete implementation of the functionality. In this case the developer has not made a basic code related mistake, but has overlooked certain aspect of the wanted functionality during implementation. For example requirements for login functionality require the use of special characters in the users' password, while the implemented code accepts only alphanumeric characters. This type of fault is mapped to component testing level.
3. Code stability implementation fault. The fault is caused by an implementation, which in certain circumstances can cause the software to fail. For example code reserving memory every time the code is executed, without ever releasing the memory. The code will function until memory is exhausted, after which the code will fail to function, in worst case causing failure of a complete server. This type of fault is mapped to system test level, as it is the only phase in which the code is subjected to load exceeding the set requirements.
4. Code suboptimal implementation fault. The fault is caused by suboptimal implementation, which will not cause an actual failure, but significant and noticeable reduction in system performance. For example poorly designed database query, which will function correctly, but will be expensive to execute and cause a performance degradation when subjected to load. This type of fault is mapped to system test level, as it is the only test phase where the system is subjected to meaningful load.
5. Third party product fault. The fault is caused by an issue in third party product, such as an operating system or database server software. For example a fault in the

database server software that causes the database service to fail, when too many connections are opened towards the database service. This type of fault is mapped to system test level, as at that phase system is subjected to load exceeding the set requirements.

6. Configuration fault. The fault is caused by erroneous configuration. For example fault caused by allocating too little storage space for a service, which might cause the service to fail once the limit is reached. This type of fault is mapped to system test phase, as the configuration in system test environment and production environment should reflect one another.
7. Documentation fault. Fault in the documentation provided to the customer causing customer to attempt to use the system in an unintended manner, causing the customer to be unable to achieve desired end result. This type of fault is mapped to system test level, as the documentation is reviewed when the system has been integrated and can be verified from customer perspective.
8. Procedure related fault. The fault is caused by incorrect procedure or incorrect execution of a procedure. For example a configuration change needed for the new software release is carried out incorrectly, and as a result the system fails to function correctly or the capacity is reduced. This type of fault is mapped to production testing.
9. Functional availability fault. The fault is caused by temporary or permanent loss of functionality. For example, due to unexpected increase in load, the execution times increase, resulting in lower availability of functionalities for the customer. This type of fault is mapped to production test level.

Theoretically any fault could be caught in any testing level. In this model the fault is mapped to the first level in which the fault could have been observed.

6.2.3 System Tier Mapping

The issues were also mapped to the system tier to which the fault is located on. As mentioned earlier (Chapter 6.1) the component where the fault is present is one of the

information elements already provided in the reports, so there is no need for any other mapping system.

Nonetheless the tier was marked down as one of the extracted data items in order to see if the distribution of faults had any significant differences among the tiers.

6.2.4 Issue Priority

The trouble report contains the issue priority as mentioned in Chapter 6.1. There is no need for any other mapping for the priority. There are four separate levels of priority, from highest to lowest as follows: critical, high, medium, low.The customer may perceive a technically non-threatening fault, such as a wrong customer logo on the user interface, as having high impact on their business. Such a fault does not cause any actual technical degradation to the service itself.

From providers point of view, the four priorities are meant to be used in the following way:

1. Critical priority. To be used for issues causing extreme degradation of service or complete loss of a functionality.
2. High priority. To be used for issues causing significant impediment for the use of service or partial loss of a functionality.
3. Medium priority. To be used for issues causing moderate impact on the service quality or usability of a functionality.
4. Low priority. To be used for issues causing minor impact or inconvenience to the use of service.

The customer sets the priority upon the creation of the trouble report. There are certain response times for each priority, governed by a service level agreement (SLA) between customer and the provider. The priority can later be modified if the actual business impact turns out to be different from what was originally expected.

6.3 Data Extraction

Data extraction is the process of going through all selected trouble reports and extracting the aforementioned data element from the issue. Each trouble report that has been selected for this thesis (according to rules mentioned in Chapter 6.1) was processed in the manner visualized in Figure 7.

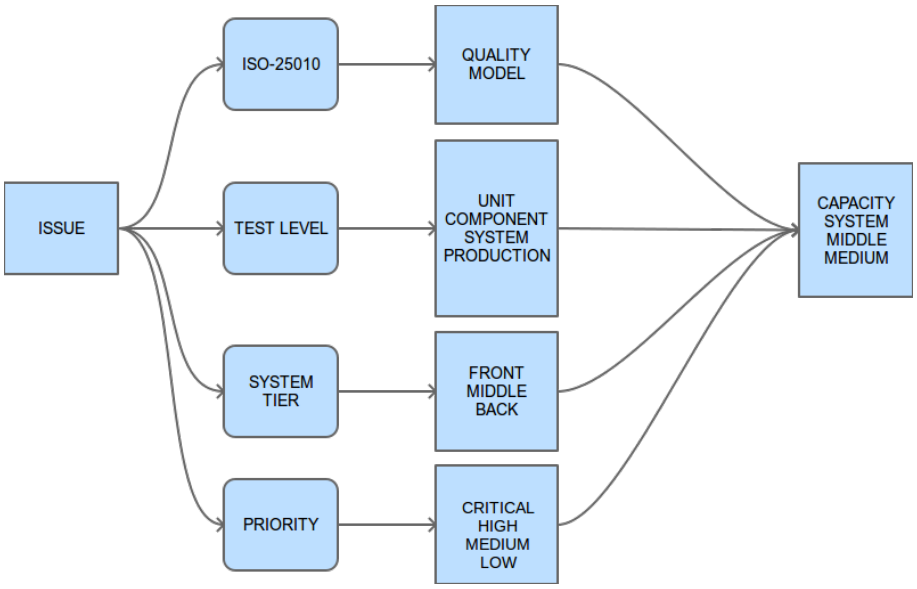

Figure 7: The classification of the issues

The output from each handled issue contains four data elements:

1. Quality model adherence : to which quality characteristic does the issue adhere to.
2. Test level : to which test level has been mapped to.
3. System tier : to which system tier is the issue located in.
4. Priority : the priority of the issue.

In addition to the four set data elements notes were attached to each processed issue to record additional information if needed.

6.4 Data Recording

After processing each trouble report the results were marked down in the following fashion (example data):

Number	Quality model	Test level	System tier	Priority
1	Modularity	System	Middle	Medium
2	Operability	System	Back	Low
3	Availability	Production	Front	High

Table 2: Example of the data recording

As can be seen in Table 2, only the related subcharacteristic was recorded. Since all the subcharacteristics are unique this also denotes the actual main characteristic. The records were stored to a spreadsheet with the optional notes. Spreadsheet software gives a multitude of options to calculate different sets of numeric results from the data as well as create graphical representation of the results if and when needed.

7 Results

In this section the results are reviewed. The numeric values are presented in tables and interpretations of the values is given between the different mapping categories (for example system tier mapped against priority).

7.1 Analyzed Issues and Top Level Statistics

In this section the selected issues are discussed. The reasons for dismissing certain reports is explained. Also the top level statistics are explained.

During the six month period selected for this study 182 trouble reports were issued. Of those 182 reports 63 were dismissed due to the following reasons:

1. The report was still open at the time of the analysis, meaning that no solution for the reported problem had been decided.
2. Insufficient information provided, meaning that the problem had apparently been fixed and the report closed, but information provided about the nature of the problem and/or solution were not sufficient to determine the suitable test level or the classification to the quality model
3. Improvement or development items being reported as trouble reports, meaning that the issue mentioned in the report was not an actual problem in the current software, but an improvement item.

The applicable 119 trouble reports were then analyzed, the results are mentioned below in several tables. The main statistics can be observed in Table 3.

System tier	Amount	Test level	Amount	Priority	Amount
Front	28	Unit	14	Critical	1
Middle	56	Component	48	High	16
Back	35	System	55	Medium	49
*	*	Production	2	Low	53

Table 3: Top level statistics

From the top level statistics in Table 3 it can be seen that most of the trouble reports had been issued regarding the middle tier (approx. 47 % of the issues), second most regarding the back tier (approx. 29 % of the issues) and the least regarding the front tier (approx. 24 % of the issues).

As the majority of business logic is implemented in the middle tier it is not surprising that the majority of problems occur there. More interesting is the fact that more issues are attributed to back tier than front tier, as the back tier usually relies more on third party applications such as database software, which at least in the case of a commercial solution is assumed to be tested by the developing company. However, the functionality of databases and similar back end systems is often customized with custom procedures and scripts, which need to be tested in-house. One limiting factor for the number of reports attributed to front tier is the visual nature of the tier. The feedback to the developer is direct and visual in the front tier when compared to the other two tiers, and thus the feedback loop to fix possible issues is tighter.

Regarding test levels the highest number was attributed to the system test level (approx. 46 % of the issues), followed closely by component test level (approx. 40 % of the issues). In the third place comes unit test level (approx. 12 % of the issues) and in the fourth place is production test level (approx. 2 % of the issues).

Statistics regarding test levels would suggest that unit testing in general seems to be in quite good order, as only approximately one tenth of the issues were attributed to that test level. The absolute bulk of the issues attributed to system and component test level combined with the relatively low number on unit test level seem to indicate that while the implementation in one tier has been correctly carried out, problems arise when tiers and functionalities are integrated. The low number attributed to the production test level is to be expected as it is unfeasible to test extensively in production environment. However, as there are some issues attributed to production test level there still apparently are such differences between test setups and production environment setup that call for certain testing activities to take place in the production environment and configuration as well.

The largest number of issues were of low priority (approx. 45 % of the issues), followed by medium priority (approx. 41 % of the issues), then in the third position is high priority (approx. 16 % of the issues) and in the fourth place critical priority (less than 1 % of the issues).

The bulk of the issues (86%) being attributed to low and medium priorities indicate that the system is being reasonably well maintained. If the maintenance process were seriously flawed or neglected, the bias of the priorities would most likely weight heavily on the high priority and also escalate to critical if high priority issues are not correctly handled. Only one critical issue was observed during the six month period.

Issue distribution to the ISO 25010 quality model is presented in Table 4.

Issue classification to ISO 25010 characteristics		
Characteristic	Subcharacteristics	Amount
Functional suitability	Functional completeness	21
	Functional correctness	44
	Functional appropriateness	0
Performance efficiency	Time behaviour	3
	Resource utilization	1
	Capacity	6
Compatibility	Co-existence	0
	Interoperability	0
Usability	Appropriateness recognizability	7
	Learnability	0
	Operability	19
	User error protection	1
	User interface aesthetics	0
	Accessibility	4
Reliability	Maturity	1
	Availability	6
	Fault tolerance	2
	Recoverability	1
Security	Confidentiality	0
	Integrity	2
	Non-repudiation	0
	Accountability	0
	Authenticity	0
Maintainability	Modularity	0
	Reusability	0
	Analysability	0
	Modifiability	1
	Testability	0
Portability	Adaptability	0
	Installability	0
	Replaceability	0

Table 4: Issue classification to ISO 25010 quality model

Regarding the mapping to the ISO 25010 quality model characteristics the majority (approx. 55%) were mapped to functional suitability. At a clear second place was usability (approx. 26%) with performance efficiency and reliability sharing the third place with 10 issues each (approx. 8% of the overall number for each). The few remaining issues were divided between security and maintainability (approx. 3% of the overall number).

These numbers are not a clear indicator of quality per se, they only indicate in which quality areas the challenges lie. Also it has to be remembered, that the lack of issues attributed

to certain quality characteristic does not necessarily denote "100%" quality within that characteristic either.

7.2 Cross-referencing System Tier and Priority

Table 5 lists the correlation between the system tier and priority.

	Critical	High	Medium	Low
Front	0	4	8	16
Middle	1	9	29	17
Back	0	3	12	20

Table 5: System tier and priority

Interestingly while all the tier have a different number of issues overall, all the tiers were attributed almost the same number of low priority issues, the majority of which were attributed to the back tier.

It was already established earlier that the majority of the issues were attributed to the middle tier. However, while the majority of all issues were of low priority, for the middle tier the majority of issues (approx. 51%) were of medium priority. Also the only critical issue is attributed to the middle tier.

While the front tier has the lowest overall number of trouble reports, it has the highest ratio between high and medium priority issues. This may be due to the binary nature of many front tier features. While a fault in the middle or back tier might reduce functionality, a fault in the front tier may block the user accessing functionality altogether and thus result in a high priority trouble report.

7.3 Cross-referencing System Tier and Test Level

Table 6 lists the correlation between the system tier and test level.

	Unit	Component	System	Production
Front	3	7	17	1
Middle	9	21	25	1
Back	2	20	13	0

Table 6: System tier and test level

Regarding the middle tier the majority of the issues (approx. 45%) are mapped to the system test level, followed by the component test level (approx. 38%). High numbers in component and system test levels combined with the relatively low number (approx. 16%) in the unit test level indicate that the issues attributed to the middle tier are not necessarily faults in the tier itself per se, but are manifested mainly when tested as part of the integrated system.

As the role of the middle tier is to act as an intermediary between the front and back tier, it has practically twice the number of interfaces than either front tier or back tier. When the interface on the front tier is altered, the change must be mirrored in the front tier facing interface of the middle tier, but maybe also in the back tier facing interface. This puts high focus on the component and system test levels, as no unit test can address these issues and the production test level is obviously the wrong place to be addressing this sort of issues.

What seems surprising is that for the front tier the majority (approx. 61%) of the issues are attributed to the system test level. However, this is explained by the fact that documentation related issues are mapped to the system test level. As the documentation for customers mainly instructs the users regarding the functionalities of the user interface, faults of that nature are mapped to the system test level of the front tier.

Regarding the back tier it is unexpected to see the highest number (approx. 57%) of issues being attributed to the component test level. This indicates that the experienced

faults are mainly in the customized and application specific adaptations rather than faults in the core functionality, which would be mapped to the system test level.

7.4 Cross-referencing Priority and Test Level

Table 7 lists the correlation between the priority and test level.

	Unit	Component	System	Production
Critical	0	0	1	0
High	2	4	9	1
Medium	6	22	20	1
Low	6	22	25	0

Table 7: Priority and test level

The bulk mass of issues are attributed to the medium and low priority with the component and system test levels (approx. 75%). The system test level is attributed the highest number of high priority issues (approx. % of all high priority issues). Also the only critical issue is attributed to the system test level.

The majority of the issues being attributed to the system test level is perhaps indicative of the general nature of the system function and composition. While the user operates on the front tier, business logic is carried out on the middle tier and data resides on the back tier. All of the tiers have to function seamlessly together in order to provide the complete functionality the user is expecting. It has to be also considered that while interface stubs and mock do facilitate component level testing on separate tiers, the higher value can be obtained when the whole system is integrated to testing.

The production test level is only attributed with two issues in total. But on the other hand they are of high and medium priority, thus increasing the relative significance over the sole number. The main difficulty with the production test level is choosing the test scenarios, as their possible value has to outweigh their possible negative aspects, such as resource consumption.

7.5 Cross-referencing System Tier and Quality Model Mapping

The mapping between the system tier and the ISO 25010 quality model characteristics can be seen in Table 8.

System tier to ISO 25010 characteristics mapping		
Subcharacteristic	System tier	Amount
Functional completeness	Front Middle Back	5 11 5
Functional correctness	Front Middle Back	2 24 18
Time behaviour	Middle Back	1 2
Resource utilization	Middle	1
Capacity	Middle Back	2 4
Appropriateness recognizability	Front Middle Back	4 2 1
Operability	Front Middle Back	8 8 3
User error protection	Middle	1
Accessibility	Front	4
Maturity	Middle	1
Availability	Front Middle	3 3
Fault tolerance	Middle Back	1 1
Recoverability	Back	1
Integrity	Front	2
Modifiability	Middle	1

Table 8: Mapping system tier to ISO 25010 quality model

Interestingly the majority of issues attributed to the front tier seem not to be software bugs per se (for example issues mapped to functional correctness) or incomplete implementation (for example mapped to functional completeness). Instead the majority of the front tier related issued are mapped to operability. This would suggest that while the correct functionality is available, the user interface does not support the user in efficient operation of the system.

The front tier also has the highest number of issues attributed to the characteristic "Appropriateness recognizability", suggesting that the documentation provided is not supporting the user as needed or is not providing the correct image of the service and its functionalities. The front tier is also attributed with all of the accessibility related issues. This is logical given the role of the tier.

The middle tier was attributed with the majority (approx. 55%) of the issues mapped to functional correctness. This is not surprising given the key role of the middle tier and the fact that most development is focused on that particular tier. Also the majority of the functional completeness related issues are mapped to the middle tier, which indicates narrow and limited implementation.

An unexpected result is to see the high number (approx. 41%) of functional correctness related issues mapped to the back tier. This would indicate issues with custom adaptations in the back tier as well as shortcomings in system integration.

7.6 Cross-referencing Test Level and Quality Model Mapping

The mapping between the test level and the ISO 25010 quality model characteristics can be found in Table 9.

Test level to ISO 25010 characteristics mapping		
Subcharacteristic	Test level	Amount
Functional completeness	Unit Component System	4 13 4
Functional correctness	Unit Component System	8 23 13
Time behaviour	System	3
Resource utilization	System	1
Capacity	Component System	2 4
Appropriateness recognizability	System	7
Operability	Unit Component System	2 7 10
User error protection	System	1
Accessibility	System	4
Maturity	System	1
Availability	System Production	4 2
Fault tolerance	Component System	1 1
Recoverability	System	1
Integrity	Component	2
Modifiability	System	1

Table 9: Mapping test level to ISO 25010 quality model

The issues related to availability are divided between the system (approx. 66%) and production (approx. 33%) test levels. This seems to indicate that while the majority of problems are systemic (depending on system integration and/or setup), still a considerable proportion stem from configuration, as production data and configurations are more varied that test setups.

The issues related to functional correctness are for the most part (approx. 52%) mapped to the component test level, which seems logical. It also suggest that the majority of issues

which could be characterized as classical "bugs" are related to a single tier or interfaces, but do not affect the complete system.

The capacity, time behaviour and resource utilization related issues are mainly mapped to the system test level, which is as expected, since faults of this nature are only revealed when the system is under considerable load.

7.7 Cross-referencing Priority and Quality Model Mapping

The mapping between the priority and the ISO 25010 quality model characteristics can be seen in Table 10.

Priority to ISO 25010 characteristics mapping		
Subcharacteristic	Priority	Amount
Functional completeness	High Medium Low	3 13 5
Functional correctness	Critical High Medium Low	1 6 13 24
Time behaviour	High Medium Low	1 1 1
Resource utilization	Medium	1
Capacity	High Medium Low	1 2 3
Appropriateness recognizability	High Medium Low	1 1 5
Operability	High Medium Low	1 12 6
User error protection	Medium	1
Accessibility	Medium Low	1 3
Maturity	Low	1
Availability	High Medium Low	3 1 2
Fault tolerance	Medium Low	1 1
Recoverability	Medium	1
Integrity	Low	2
Modifiability	Medium	1

Table 10: Mapping priority to ISO 25010 quality model

While the majority of all issues (approx. 37%) and also the one critical issue are mapped to the functional correctness, bulk of the issues in this category are still of low priority. This indicates that the functional shortcomings in general have not caused significant issues

to the users.

Compared to functional correctness there is quite a low number of issue mapped to capacity (44 and 6, respectively). However, half of the issues mapped to the capacity characteristic are of high priority, indicating that users are experiencing much more significant disturbance to their usage of the system by the faults of this nature.

Interesting but understandable logic seems to reveal itself when the priority distribution between the issues related to subcharacteristics availability, operability and accessibility are compared. The issues related to availability have the highest bias to the upper end of the priority scale, operability has bulk on the medium priority and accessibility has the majority on the low end of the scale. This seems logical as the issues related to availability are mainly related to complete unavailability of a certain functionality whereas the issues related to operability are concerned with complicated, misleading or user-unfriendly interfaces, but without loss of functionality. Accessibility ranks the lowest in priority probably due to the fact that most users adapt to minor accessibility issues if the needed functionality is available.

The issues related to quality characteristic performance efficiency, including subcharacteristics time behaviour, resource utilization and capacity, rank overall mainly to medium and low priority. Also numerically they contribute less than one tenth to the overall number of issues, so it can be concluded that performance does not rank among the worst shortcomings of the system.

7.8 Analyzed Issues in Relation to Complete Time Period

As mentioned earlier, 63 trouble reports were culled from the complete number of 182 from the six month period and the remaining 119 trouble reports were analyzed. For the sake of completeness, the priority and the system tier of the 63 trouble reports were looked at to see if they would have changed the overall distribution were they included.

From those 63 further two were removed at this point, as those reports did not indicate any sort of fault or problem.

Taking the priority first, from the 61 reports, 12 were of high priority, 19 of medium priority and 30 of low priority, percentage-wise approx. 20% high, 31% medium and 49% low. The priorities of the analyzed 119 reports were 1 critical, 16 high, 49 medium and 53 low (approx. 13% high, 41% medium, 45% low). As can be seen, the inclusion of the 61 reports would have raised the percentage of high priority issues to approx. 16% and lowered the share of medium priority issues to approx 38%, with the number of low priority issues raising with about one percentage point to approx. 46%.

Regarding the system tier, from the 61 reports, 15 were attributed to the front tier, 26 to the middle tier and 20 to the back tier, percentage-wise approx. 25%front, 43% middle and 32% back. The results from the analyzed 119 reports were 28 attributed to the front tier, 56 attributed to the middle tier and 35 attributed to the back tier (approx. 24% front, 47% middle, 29% back). Including the 61 reports would then have raised the share of issues attributed to the front tier to approx. 25%, lowered the number of issues attributed to the middle tier to approx. 46% and increased the number of issues attributed to the back tier to approx. 31%.

It can be concluded that the inclusion of the 61 reports would have had some impact on the distribution between the priorities and no real significance on the distribution between system tiers.

Another interesting fact is that there is no significant skew that would emphasize any priority or system tier regarding the reports that were still open during the time of the analysis. This seems to indicate that the priority or system tier has no impact in general

regarding the handling time of the issue.

8 Discussion and Conclusions

In this section the conclusions drawn from the material and analysis process are presented. Also recommendations for possible actions to improve the testing process in order to increase the service quality are given. The limitations of the study and possible ideas for future studies and related work are also discussed.

8.1 Recommendations

The results of this study were not a dramatic revelation, but provided a valuable insight into specific shortcomings of the testing processes. While the current situation is on no way negative, there are several ways in which the processes can be enhanced.

Quality-wise the biggest issue is easy to see in Figure 8.

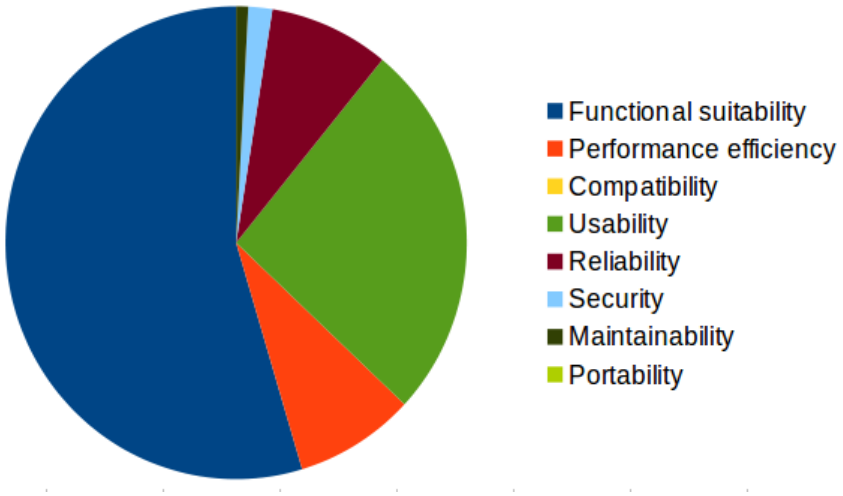

Figure 8: Issue distribution to the ISO 25010 quality model characteristics

Over half of the pie is eaten up by the quality characteristic functional suitability, divided into functional completeness and functional correctness. Basically this boils down to two things: incomplete functionality experienced by the customer and plain old run-of-the-mill

bugs.

The problems with functional completeness seem to stem from incomplete understanding of the requirements or problems handling changes in the requirements. The plain vanilla bugs (meaning the issues regarding functional correctness) seem to mainly originate from component and system level issues, such as interface related and system integration problems. In both cases it seems that the core problem in revealing these issues at any level of testing is lack of scope and variation. This also addresses the question regarding the importance of the different test levels. While all the test levels as such are needed, the current situation would call for increased focus on the component and system test levels.

To improve the quality of functional suitability characteristics the study would suggest three courses of action to be taken:

- Increase resources used for manual testing when implementing new functionalities. While automated test cases are efficient in securing base functionality, in general they do not provide lot of variation from the basic positive and negative paths. A person experienced in the use of the system will spot discrepancies quickly and will also detect unrelated problems.
- Increase the scope of component and system level automated tests. As lot of bugs are revealed when the system is used by the customer with varied inputs and parameter values, it would be beneficial to increase the test input and scenario variation in automated tests as well. Randomizing test input where applicable should also be considered in order to root out problems even in the most unlikely scenarios.
- Improve the requirement handling process. While not being a test process improvement per se, it seems obvious that improving the requirement handling and flow of information would reduce especially the issues attributed to functional completeness. The developers and consequently the whole service will benefit if the developers have more exact and detailed requirements already at the very beginning of their work for a new feature or improvement.

While it was established earlier that most of the issues are located in the middle tier of

the system it can be concluded that not much can be gained in targeting specifically this tier or any other, for that matter. As stated above, the issues are largely due to system interfaces and integration, thus rendering targeting any particular tier pointless.

The second largest quality downfall seen in Figure 8 is the usability characteristic, divided between (from highest to lowest) operability, appropriateness recognizability, accessibility and user error protection.

The biggest culprit by far is operability. Operability is mapped to usability testing, which is part of the system test level. Also accessibility related problems fall under usability testing. Problems in appropriateness recognizability stem from shortcomings in the provided documentation, testing of which is also part of system test level.

To alleviate these issues it is recommended to take the following actions:

- Increase focus on usability testing. As usability is a difficult item to measure automatically, it is costly and labor intensive manual process, but will in all likelihood lead to increased customer satisfaction. This applies to accessibility related problems as well.
- Improve the documentation testing and overview process. Focus on details and accuracy, even with the expense of scope if trade-off is needed to keep to invested time in balance.

Other quality characteristics that take up the rest of the pie in any meaningful amount are reliability and performance efficiency. These fall under the system test level in the form of fault tolerance, robustness and load testing. While these are obviously very important aspects in testing, they do not need any immediate actions according to the numbers or priorities of the issues attributed to these characteristics.

Another possible improvement item that is not directly test process related is the quality of the trouble reports. While the trouble report as such is mainly a communication tool

to facilitate fixing a detected problem, the information could be utilized more extensively in steering the development efforts, provided that the information content is clear and sufficient. This could be achieved by creating a set of base rules for the technical content regarding explaining the problem and the implemented solution, and educating the users about the expected standard.

Also incorporating the ISO 25010 quality model characteristics in the reports should be considered to ease analyzing the quality status at any given point in time. The ISO 25010 quality characteristic could be a mandatory fixed field in the basic structure of the trouble report, so that the information would then available for all future cases.

8.2 Further Observations

Regarding the trouble reports it has to be acknowledged that while they are useful for noting a problem and acting as a communication tool throughout the solution process, they do not lend themselves to being a coherent source of documentation about the issues. While there are certain static fields in the trouble reports, as priority, affected systems and so fort, the actual freeform textual explanations about the problem and the solution are very varied. In worst cases they contain next to no useful information at all for a person who is not or has not been actively involved in the handling of the issue.

Obviously the primary function of the trouble reports is to facilitate solving the issue and not being a historical record of the events. Such limitations aside, the trouble reports do offer a quite clear statistical view on the situation and history of the system.

Relating to the results of the ISO 25010 quality model mapping, it would appear judging by the issues attributed to characteristic functional completeness that there are some challenges in handling the requirement processes end to end. As analyzing the requirements or actual implementations are not part of the present study, it remains unclear where those shortcomings might actually reside. But it seems that in some cases the understanding of the desired capability or functionality is incomplete. Logically then testing does not cover

the scenarios which are of interest to the user. This will lead in the end to a trouble report once the user experiences the lack of functionality, which from their point of view was understood to be included in the system.

Observing system quality through only trouble reports may be misleading, as only problems or issues are then noted. At the same time it would be a false conclusion to think that if some quality areas do not have any issues attributed to them that they would be perfectly handled. To have a more complete picture of the quality of a system or service overall several other data sources would be needed as well.

First it has to be acknowledged that the applicability of the results of the study is quite limited. While there are a lot of similar systems in existence and many are built from the same basic components, there are too many variables at play to assume that data gathered from one system would be valid regarding another system. Even though the working methods and tools can be identical, the variance builds up from the different competences of individual developers, the differences in the hardware environments, the available budgets, and so on. Even the most minute differences add up to a vast delta between complete systems when thoroughly examined.

However, due to the overall similar topology between multi tier systems, the division of product development efforts and faults could vaguely follow the same pattern as in this study. The highest probability for a software fault in general might lie in the middle tier, if it is assumed that the majority of the system logic is located there and all the unknown technical details are dismissed.

Testing in itself, in its implementation and execution is always context dependent and it would be unwise to draw too far reaching conclusions regarding other systems based on the present study. However, test levels themselves are generally understood and applied in a similar manner and the mapping system and analysis method for the trouble reports could be utilized in other settings as well to establish another set of results.

The ISO 25010 quality model offers a solid basis for quality assessment and can create

comparable results between multitude of systems, provided it is used in a repeatable manner. It was used in this study to denote "negative quality", that is to say, to point out the lack of quality in certain areas rather than the existence of quality. But it would be possible to approach the same subject from another viewpoint and still reach comparable results.

In this study a view of the shortcomings in certain quality characteristics was created by analyzing trouble reports. While this gives an accurate view of the actual technical problem areas, it does not give an actual view on the quality perceived by the customers. While a trouble report is a clear indicator of a fault, some aspect that is not technically a fault and does not get reported might still decrease the perceived quality. On the other hand, while the customer reports a problem related to a certain functionality, it does not necessarily mean that the customer perceives the quality of the functionality in a negative sense.

To create an accurate view of the perceived quality a study should be made in which the customers would provide direct input regarding their views on the quality as they experience it.

References

1 Causevic, Adnan, Sundmark, Daniel, Punnekkat, Sasikumar. An Industrial Survey on Contemporary Aspects of Software Testing. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=5477060 [Accessed November 1, 2015].

2 Steiert, Hans-Peter. Towards a Component-based n-Tier C/S-Architecture. ACM;. URL: http://dl.acm.org.ezproxy.metropolia.fi/citation.cfm?id=288443 [Accessed August 03, 2015].

3 Eldh, Sigrid, Hansson, Hans, Punnekkat, Sansikumar, Pettersson, Anders, Sundmark, Daniel. A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques. IEEE;. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1691683&isnumber=35654 [Accessed August 4, 2015].

4 Hosseingholizadeh, Ahmad. A Source-Based Risk Analysis Approach for Software Test Optimization. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=5485639 [Accessed November 1, 2015].

5 Hamlet, Richard. Theoretical Comparison of Testing Methods. ACM New York, NY, USA;. URL: http://dl.acm.org.ezproxy.metropolia.fi/citation.cfm?id=75308.75313 [Accessed October 31, 2015].

6 de Oliveira Neto, Francisco, Torkar, Richard, Machado, Patrícia. An Initiative to Improve Reproducibility and empirical Evaluation of Software Testing Techniques. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=7203016 [Accessed October 31, 2015].

7 Suri, Bharti, Singhal, Shweta. Literature Survey of Ant Colony Optimization in Software Testing. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=6349501 [Accessed October 9, 2015].

8 Rao, Koteswara, Raju, GSVP, Nagaraj, Srinivasan. Optimizing the Software Testing Efficiency by Using a Genetic Algorithm - A Design Methodology. ACM New York, NY, USA;. URL: http://dl.acm.org.ezproxy.metropolia.fi/citation.cfm?id=2464526.2464539 [Accessed November 1, 2015].

9 Liu, Yu, Li, Duo, Guo, Chao. Software Reliability Modeling with Fault Detection Data when Knowing Fault Severity. IEEE;. URL: http://ieeexplore.ieee.

org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=7107257 [Accessed November 1, 2015].

10 Lo, Jung-Hua. A Study of Applying Support Vector Machine and Genetic Algorithm to Software Reliability Forecasting. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=5559867 [Accessed November 1, 2015].

11 Goseva-Popstojanova, Katerina, Hassan, Ahmed, Guedem, Ajith, Abdelmoez, Walid, Nassar, Diaa, Ammar, Hany, et al.. Architectural-Level Risk Analysis Using UML. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/abstractAuthors.jsp?arnumber=1237174 [Accessed November 1, 2015].

12 Frankl, Phyllis, Hamlet, Richard, Littlewood, Bev, Strigini, Lorenzo. Evaluating Testing Methods by Delivered Reliability. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/xpl/articleDetails.jsp?arnumber=707695 [Accessed November 1, 2015].

13 Haugset, Børge, Stålhane, Tor. Automated Acceptance Testing as an Agile Requirements Engineering Practice. IEEE;. URL: http://ieeexplore.ieee.org.ezproxy.metropolia.fi/stamp/stamp.jsp?tp=&arnumber=6149535 [Accessed December 1, 2015].