Saimaa University of Applied Sciences
Technology Lappeenranta
Degree programme in Information Technology
ICT-entrepreneurship

Sampo Kivistö

Defect management in SAAS application

Thesis 2013

Tiivistelmä

Sampo Kivistö
Defect Management in SAAS application, 27 sivua
Saimaan ammattikorkeakoulu
Tekniikka Lappeenranta
Tietotekniikka
ICT-yrittäjyys
Opinnäytetyö 2013
Ohjaajat: yliopettaja Päivi Ovaska, Saimaan ammattikorkeakoulu, Department Manager Kari Ryynänen, Visma Solutions Oy

Opinnäytetyössä tutkittiin, kuinka virheiden hallintaa voitaisiin parantaa SAAS-ohjelmistossa, mitkä prosessimallit tukevat virheiden hallintaa ja mitä työkaluja virheiden hallintaan ohjelmistokehittäjien avuksi on olemassa.

Työ aloitettiin selvittämällä, kuinka Microsoft Operations Framework-prosessimalli auttaa virheidenhallinnassa. Tämän jälkeen käytiin läpi saatavilla olevia työkaluja virheiden hallintaan .NET-ympäristölle. Lopuksi työssä tutkittiin kuinka ohjelmistotasolla virheiden määrään ja laatuun voidaan vaikuttaa.

Työn tuloksena löydettiin malli ohjelmisto virheiden hallintaan sekä testikäyttöön otettava virheidenraportointityökalu.

Asiasanat: virheidenhallinta, virheidenraportointityökalut, virheidenhallintaprosessi

Abstract

Sampo Kivistö
Defect Management in SAAS application, 27 pages
Saimaa University of Applied Sciences
Technology Lappeenranta
Degree Programme in Information Technology
ICT-entrepreneurship
Bachelor´s Thesis 2013
Instructors: Ms Päivi Ovaska, Senior Lecturer, Saimaa University of Applied Sciences, Mr Kari Ryynänen, Department Manager, Visma Solutions ltd.

The purpose of this thesis was to find information how to improve defect management in SAAS application, which process models support defect management and what error reporting tools are available.

The thesis was started by finding out what Microsoft Operations Framework provides to support defect management. After that the research continued by finding useful tools that can help with managing defects. Last, it was researched how to improve error report quality at software level.

As a result the process model to manage defects was found and the error reporting tool was taken into test in the organization.

Keywords: defect management, error reporting tools, defect management process

Contents

Acronyms abbreviations and notations

.NET	Software framework developed by Microsoft
Apdex Score	Numerical measure of user satisfaction
CLR	Common Language Runtime
COBIT	Framework created by ISACA for IT management
CSI	Continual Service Improvement
Defect	Deficiency in a software product
Incident	Interruption or failure in IT Service (ITIL)
ITIL	Information Technology Infrastructure Library
MOF	Microsoft Operations Framework
SAAS	Software as a service
SMF	Service Management Function
RCA	Root cause analysis
Root cause	Fundamental reason for the occurrence of a problem

1 Introduction

The bachelor's thesis topic originated from a local software company called Visma Solutions Ltd. They had a problem that there was no agreed model of the process how people fix defects, also these defects were spread to different places including: email, application logs and server logs. This made it very difficult to manage these defects as one error can be logged thousand times in the email box. This project seeks information on which process model would fit best for SAAS defect management, what tools are available to help deal with the large number of defects and how to improve exception handling in .NET environment. Finding an appropriate process is accomplished by following Microsoft Operations Framework problem management guidance. Research for available tools is done by reviewing the most popular error reporting tools for .NET environment. Improving exception handling is done by gathering the best practices of exception handling for .NET environment.

1.1 Motivation

IT services that we use in our daily life are getting more and more popular. These services however generate numerous errors and most of them are not directly visible to end-users. These errors include performance issues, application failures, security bugs and availability problems. Problems such as these are very common, but without proper process and tools to analyse and fix them they can reduce service quality dramatically.

These internal errors are called software defects. Defects can occur for different reasons and they always have a root cause. To remove the defect permanently software developers need to find the root cause of the defect and develop a fix for it. Finding these root causes can be a long and tedious process especially when one defect can depend on data, environment, logic or a mix of them. Sometimes defects do not even contain the required information that developers need.

Developers are not the only ones who need defect information. According to Kirsi Korhonen's doctoral dissertation: "To manage software quality successfully by defect data, project decisions must be based on some understanding of the cause

- effect relationships that drive defects at each stage of the process" (Korhonen 2012, 3).

2 Microsoft operations framework

2.1 Introduction to MOF

Microsoft Operations Framework (MOF) is a question-based guidance for IT organizations that shows what is needed for the organization now, as well as activities that will keep the IT organization running efficiently now and in the future. It is worth to note that MOF does not tell organizations how any of its process should be implemented, but leaves this for the organizations to decide. Mentioned activities and processes are organized into Service Management Functions (SMFs) which are grouped together in phases that mirror the IT service lifecycle. Each lifecycle contains a unique set of goals and outcomes supporting the objectives of that phase. The IT service lifecycle has three phases and one layer that wraps them all together the plan phase, the deliver phase, the operate phase and the manage layer. Figure 1 describes the connection of each phase and the layer. (Pultorak 2008)

Figure 1. Microsoft Operations Framework IT service lifecycle (Pultorak, 2008)

2.2 Problem management SMF

The Problem management SMF is part of Operate phase of the MOF IT service lifecycle. Operate phase belongs between Deliver and plan phase as shown in figure 1. This SMF is used to guide IT professionals to resolve complex problems

that are out of scope of customer service. The following figure shows where problem management SMF belongs within the operate phase.

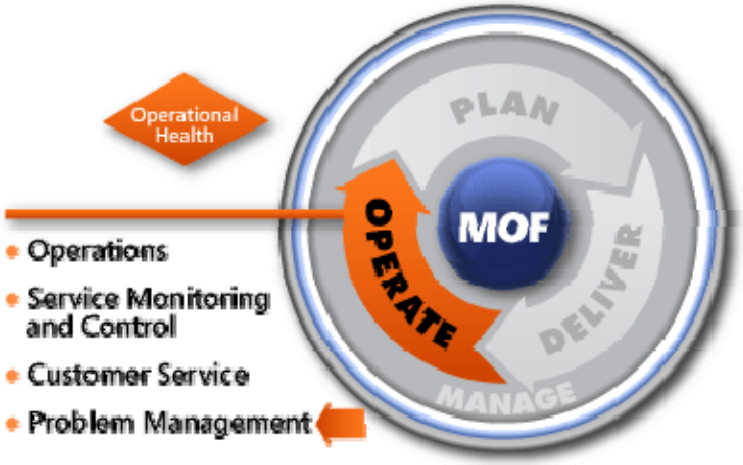

Figure 2. Position of the problem management SMF within the IT service lifecycle (Pultorak 2008)

The problem management SMF includes recording the problem, researching the problem to identify root cause and developing workaround, reactive fixes or proactive fixes for the problem. Problem management should be applied to all aspects of IT - including application development, server building, desktop deployment, user training and service operation. This thesis heavily focuses on application development. The basic idea is that as more problems are recorded, researched and resolved the software will experience fewer defects (Pultorak 2008.)

2.3 Researching the problem

Problem management process contains four different processes: to document the problem, to filter the problem, to research the problem and to research the outcome. This thesis is only interested of researching the problem process. The following figure shows the process flow of researching the problem.

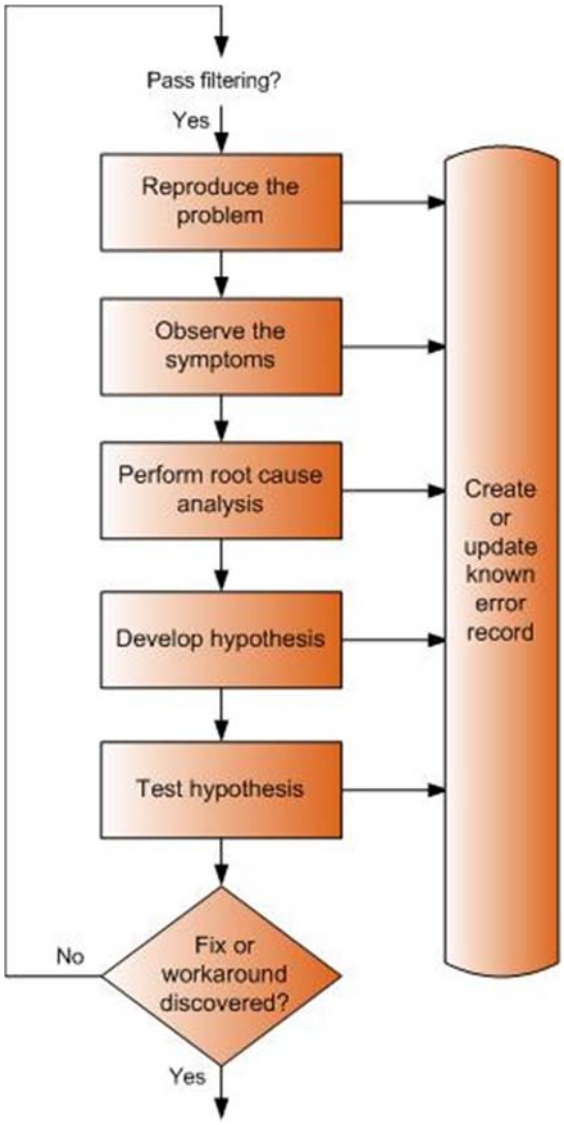

Figure 3. Researching the problem process of problem management SMF (Pultorak 2008)

The following five chapters describe each part of the process, the key questions and the best practices.

2.3.1 Reproducing the problem

Researching the problem process starts with reproducing the problem part. It takes problem record as input and the desired output is an environment where the problem can be studied and observed. According to Microsoft operations framework guidance the following things are considered as the best practices of this part. Firstly, production systems should not be used for problem management at all. Secondly, making changes to systems or services during this phase is not

allowed. Thirdly, the steps to reproduce the problem should be documented in full detail in the problem record. Key questions during this phase are “Can the problem be reproduced at will?”, “What user context or security access is required to reproduce the problem?” and “Will special lab equipment be required or can this be reproduced on any system?” (Pultorak 2008.)

2.3.2 Observing the symptoms

Observing the symptoms takes problem record as input and also lessons learned during the reproduction phase. The output of this phase is understanding of the timing, triggers and results of the problem. This phase’s key questions are the following: “What are the symptoms of the problem?”, “How can they be observed?” and “What tools are required to capture and record the occurrence of the problem?” This part of the process is meant to get more information about the problem so that the next part Performing root cause analysis is possible. Getting as much information as possible will help with the next part. (Pultorak 2008.)

2.3.3 Performing root cause analysis

Performing root cause analysis is the most important part of the whole process, because this is where things can go well or wrong. This is also the part that takes most of the time in the process. Before starting this phase it is required to select a root cause analysis technique and update the problem record. The desired output is hypothesis to test. The only key question MOF gives for this phase is “What technique should be used for performing root cause analysis?” More of this can be found in Chapter 4 Root cause analysis techniques. (Pultorak 2008.)

2.3.4 Developing hypothesis

The input for developing hypothesis part is the output from root cause analysis and problem record. In this phase developers basically develop what was found in the earlier phase. Key questions to keep in mind are the following: “What actions might work around this problem?”, “What actions might fix this problem?”, “Could this problem be the result of another problem?” and “Have changes been made to the service or system recently that may have created the problem?” The desired output is hypothesis to test and new or updated known error record. The

best practice during this phase is simply to document. As the problem management process is repeated, it can be much more efficient using data created in the earlier phases. (Pultorak 2008.)

2.3.5 Testing hypothesis

Testing hypothesis takes hypothesis to test and problem record as input. The desired output is simply a new or updated problem record. This is the phase where the developed functionality or workaround is tested. Key questions are the same as in the previous phase. The best practices: Test no more than one hypothesis at a time, keep a control in place to results of the testing, and document all of the results – both positive and negative. (Pultorak 2008.)

3 Root cause analysis techniques

One of the most difficult areas of the problem management process is analysing the root cause of a problem. Since problems are best solved by attempting to correct their root causes, this is a critical part of resolving any problem. This part can be made little easier with some well-known techniques to help finding possible root causes. It should be noted that the selected root cause analysis technique should depend on the defect and available resources. (Pultorak 2008.)

3.1 Five whys

The most common method for root cause analysis is known as “Five whys”. The idea of five whys is to ask question “why” five times for example:

- **Problem**: User sees error 500 when trying to open a link

1. **Why**: Because error handling class is showing error page to hide critical information
2. **Why**: Because repository class is throwing exception
3. **Why**: Because database cannot find the required foreign key
4. **Why**: Because the required foreign key is not there
5. **Why**: Because database is not up-to-date

This method is quick and useful in some situations, but it always leads to one cause, also the outcome varies depending of whys that have been questioned. Repeating the procedure with different answers gives different output. This technique also requires a lot of information to start with or it can lead to useless outcome. (Six Sigma 2013, 5 Why's.)

3.2 Fishbone Diagram

The idea of Fishbone diagram is to categorize the problem, placing the most likely cause first. Fishbone diagram helps to visualize the problem and areas that depends on the problem. Fishbone diagram is handy for complex problems, but it is too time consuming for simple ones.

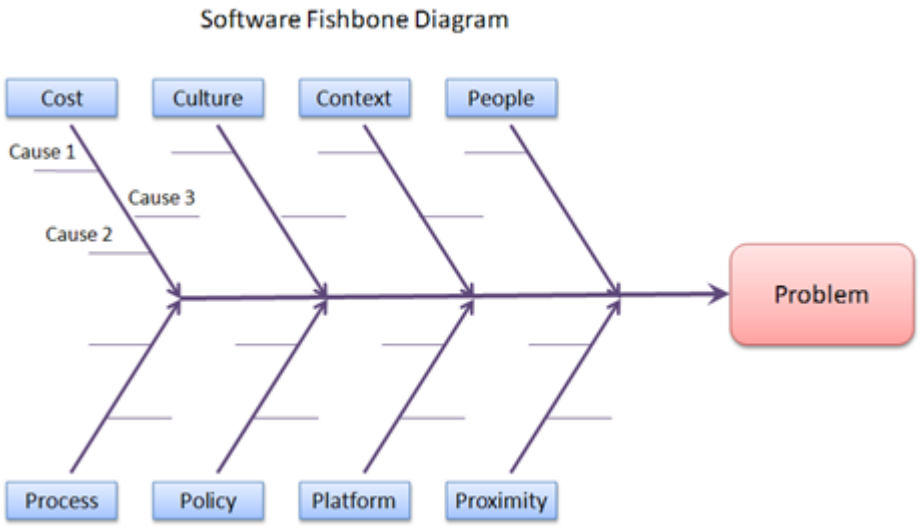

Figure 4 Software Fishbone Diagram Example (MSDN 2008, Root Cause Analysis for Software Problems)

Using fishbone diagram proceeds by first selecting the most likely categories why did some problem occur. Second, after the categories are in place start asking question why this category causes the problem, until you cannot think of any other reason. Repeat this for all the categories. Third, when there are possible root causes in place, verify the logic by adding 'cause' before the statement. (Youtube, Root Cause Analysis (RCA) using Ishikawa/Fishbone Diagrams 2010.)

3.3 Fault Tree analysis

Fault tree analysis is another visual technique used to help with root cause analysis. Fault tree graphically illustrates events that might lead to a failure so the failure can be prevented. This is usually used to prevent something from happening, for example, finding security issues or critical logic failures, but it can be used for defect's root cause analysis as well as shown in figure 5 (Microsoft Office, Create a fault tree analysis diagram.)

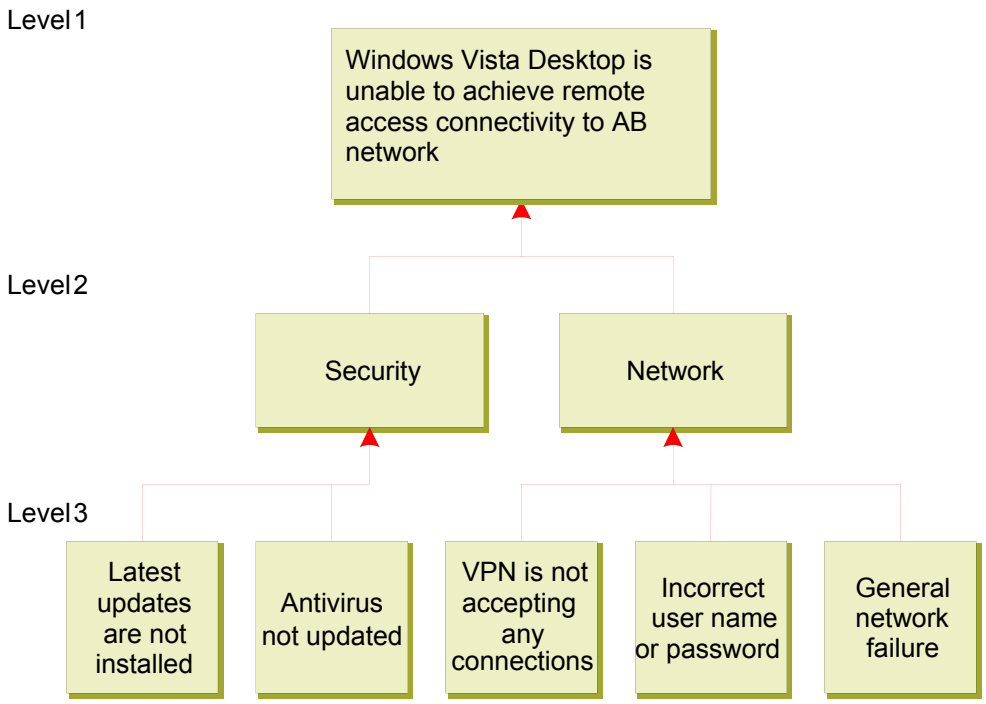

Figure 5 Fault tree analysis technique (Pultorak 2008).

Fault tree analysis begins by defining the top defect. After that it continues by adding events that could lead to the defect. Repeating this until there is the final stage when there is no need to add anymore events. Figure 5 shows an example of level three root cause analysis using fault tree technique.

4 Difficulties in managing software defects

4.1 Managing large number of defects

One of the most common difficulties of managing software defects is the fact that software is used to solve complex problems. The book "Code Complete" by Steve

McConnell has a section about error expectations. According to McConnell Industry average is about 15-50 errors per 1000 lines of delivered code. He later on mentions that this is structural code that has some logic behind it and might include a mix of coding techniques. (McConnell 2004, Amartester 2007 Bugs per lines of code.)

However customers usually see the most common problems first and this same problem will get logged to the error management tool multiple times. Luckily there are ways to find this kind of errors to increase the product quality.

4.1.1 Detecting important defects

One way to reduce the number of defects is to fix first the defects that are caused by the same root cause. This can be accomplished by doing root cause analysis for defects. However, doing RCA manually for all tickets will take too much time and sometimes will not even provide the desired output.

Heuristics are a good way to filter down the number of the defects. For example heuristics could be used as occurrence filter to see how many duplicate exceptions have happened in a specific time. Table 1 shows the occurrence based exception filter.

Exception	Exception Message	Occ.
System.DivideByZeroException	Attempted to divide by zero.	2
System.NullReferenceException	Object reference not set to an instance of an object.	52
System.IO.FileNotFoundException	Could not find file 'data.lib'.	3

Table 1 Exceptions by occurrence

Assuming all defects in Table 1 are the same level issues fixing System.NullReferenceException should have higher priority than others as it seems to happen more often.

Kevin Bartz from Hardvard University and Jack W. Stokes and John C. Platt from Microsoft Research have taken this even further. They have created a mathematical algorithm to analyse different error reports. This algorithm also categorises the errors by error and callstack details. The primary use case is to allow a developer to check if a similar error report was already resolved. A secondary use case is to provide diagnostic help, for example, multiple different errors occurred,

but all of them share common attributes. These attributes are highlighted and can provide clues about the underlying failure. (Usenix Finding Similar Failures Using Callstack Similarity.)

Each error report gathers the following metadata about the failure:

- Type of failure: a crash, hang or deadlock.
- Name of the process that launched offending stack
- Exception code: For crashes only, four-byte code such as '0xc0000005', hangs and deadlock do not have exception code
- Offending callstack: The ordered sequence of the offending thread's stack at the time of the failure.

Module	Function	Offset
kernel32	ByteCallback	0x3
kernel32	WideCharExpand	0x2
kernel32	MultiByteToWideChar	0x9
A3DHF8Q	–	0x3820523
A3DHF8Q	–	0x3723952
A3DHF8Q	–	0x3945323
kernel32	ProcUserText	0x4
user32	TextDecode	0x4096
user32	ReadDialog	0x4096
user32	OpenDialog	0x4096
ntdll	RtlThreadStart	0x0
ntdll	RtlInitThreadThunk	0x0

Table 2 Example of callstack (Usenix, Finding Similar Failures Using Callstack Similarity)

In table 2 there is an example of a callstack that is used to analyse the defect. This callstack has five different frame groups which are associated with modules A3DHF8Q, user32, ntdll and kernel32. To make it possible to categorize two different errors by the same root cause Microsoft uses already existing data made by their developers as shown in figure 6.

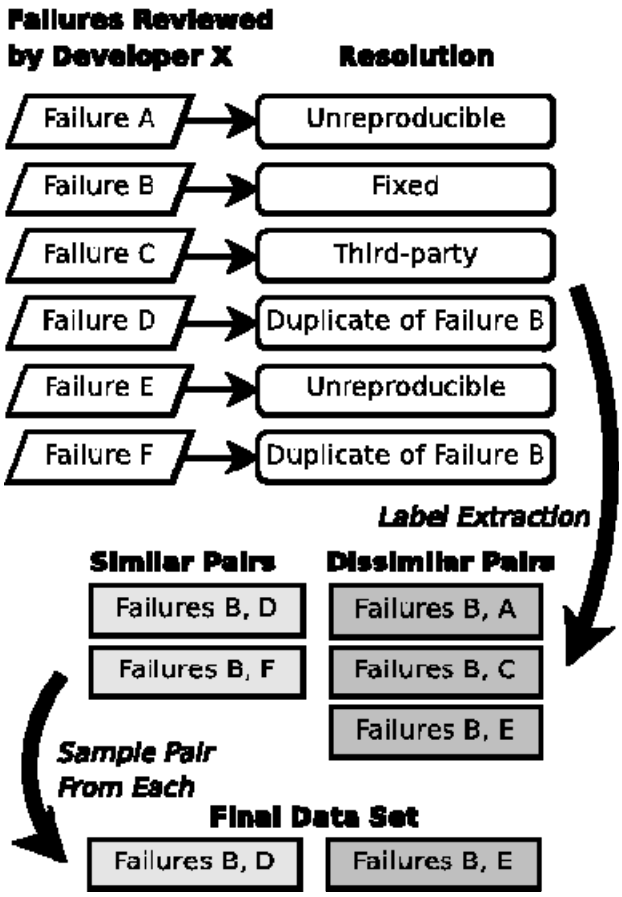

Figure 6. Illustrating how to categorize similar errors (Usenix, Finding Similar Failures Using Callstack Similarity)

After resolving a defect developers mark the defect resolution as third-party, duplicate of another defect, unreproducible or fixed. When this is done the created resolution data can be used to pair the errors making similar and dissimilar pairs. The last part of the process is to pick randomly one similar and one dissimilar pair and the data of those two pairs is used to filter the large number of defects to find similar root causes. (Usenix, Finding Similar Failures Using Callstack Similarity.)

4.1.2 When to throw exception

Another way of reducing number of defects is to remove unnecessary exception calls. User The Digital Gabeg on Stackoverflow forums has a nice guideline when to throw exception. “An exception is thrown when a fundamental assumption of the current code block is found to be false”. (Stackoverflow, When to throw an exception.)

```
public static class Extension
{
    public static bool IsNumeric(this string s)
    {
        float output;
        return float.TryParse(s, out output);
    }
}
```

Figure 7 Example of not throwing exception

In figure 7 there is a function that is checking “is this text numeric”. This function should never throw exception, because it can always be answered true or false. Every single string either is numeric or is not, there is no exception. Using Try-Parse function helps to accomplish this, because in case it fails the function returns false.

```
public static class Extension
{
    public static bool isLengthMoreThanFive(this List<int> intList)
    {
        if (intList == null)
            throw new ArgumentNullException();
        return (intList.Count > 5);
    }
}
```

Figure 8 Example of throwing exception

Figure 8 shows an example when exception should be thrown. The function is simply just checking a length of an integer based list. However it is making an assumption that the object that it is given is actually an integer based list. If we hand it a null object it could not check its length and therefore returning just true or false would break its own logic. In this case throwing exception is necessary, and developers should fix the root cause of the problem why null object was handed to this function. (Stackoverflow, When to throw an exception.)

4.2 Available tools for defect management

Without proper tools managing large number of software defects is nearly impossible. These defects are logged to many different places, for example SQL-server log, Server application log, txt files, email etc. It is highly recommended to use some error reporting tool or service to keep the defects organized. This chapter

goes through some error reporting tools that could be used for the defect management. For this research Visual Studio 2012 Ultimate Update 3, Microsoft Windows 8.1 and Google Chrome 30.0 software are used.

4.2.1 Raygun

Raygun is SAAS based real-time error logging solution made by Mindscape. The main objective of Raygun is to make finding, diagnosing and fixing errors easier. When user generates an error in the application it is automatically reported to Raygun. These errors can be viewed by using Raygun's web UI.

Installing Raygun error reporting to .NET web application is rather straightforward. The first step is to install Mindscape.Raygun4NET package using NuGet package manager. This can be done either by using UI or command line tool by the following command: *Install-Package Mindscape.Raygun4NET*. After successfully adding Mindscape.Raygun4NET package to the required project the next step is to configure web.config file as shown in following figure 9.

```
<configuration>
  <configSections>
    <section name="RaygunSettings" type="Mindscape.Raygun4Net.RaygunSettings, Mindscape.Raygun4Net"/>
  </configSections>
  <RaygunSettings apikey="YOUR_API_KEY_HERE" />
</configuration>
```

Figure 9 Raygun's web configuration

After configuring the web.config file the only part left is to send errors to Raygun. This can be done with the following code block.

```
protected void Application_Error() {
    var exception = Server.GetLastError();
    new RaygunClient().Send(exception);
}
```

Figure 10 Example of sending exception to Raygun

Adding the above code block to Global.asax file will send any error to Raygun. These errors can be viewed by using web browser. Raygun has a simple and clean dashboard that shows the overview of the project. Dashboard can be used to see the latest exceptions and their occurred count. There is also trend line that

shows the trend of occurred exceptions by selected time period. This is useful to follow the projects' overall quality.

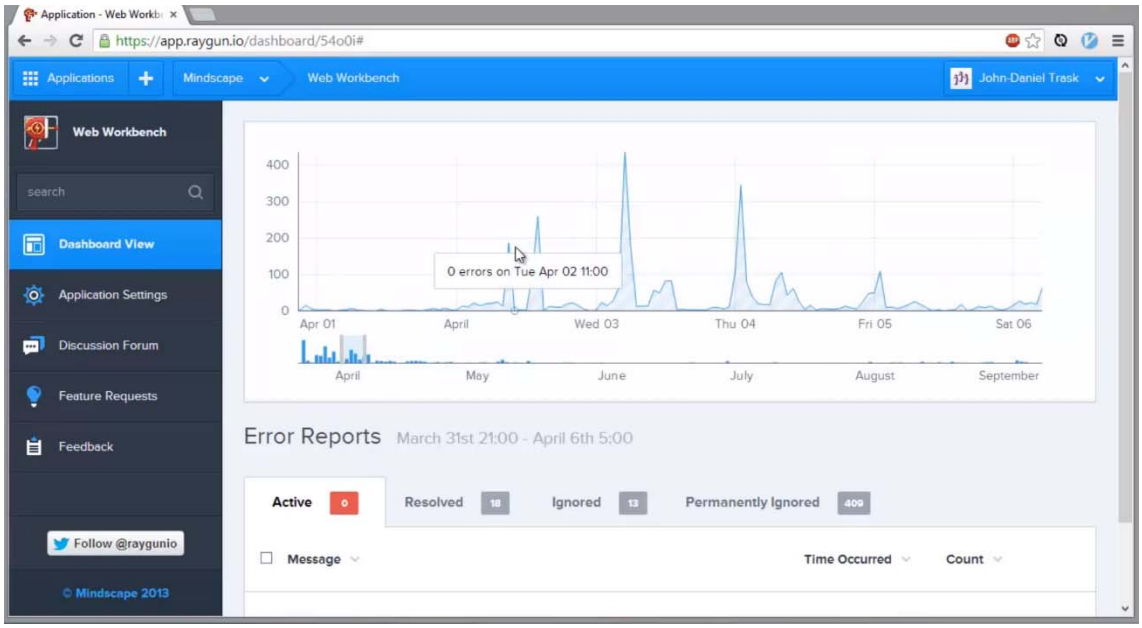

Figure 11 Raygun's user interface (YouTube, Raygun September updates)

In figure 11 under the trend line there is Error Reports summary that contains all the occurred errors of the selected project. These errors can be filtered by their status or ordered by their time occurred or count. By clicking occurred error you can open detailed error page, where you can view the insights of the error. Figure 12 shows an example of this page.

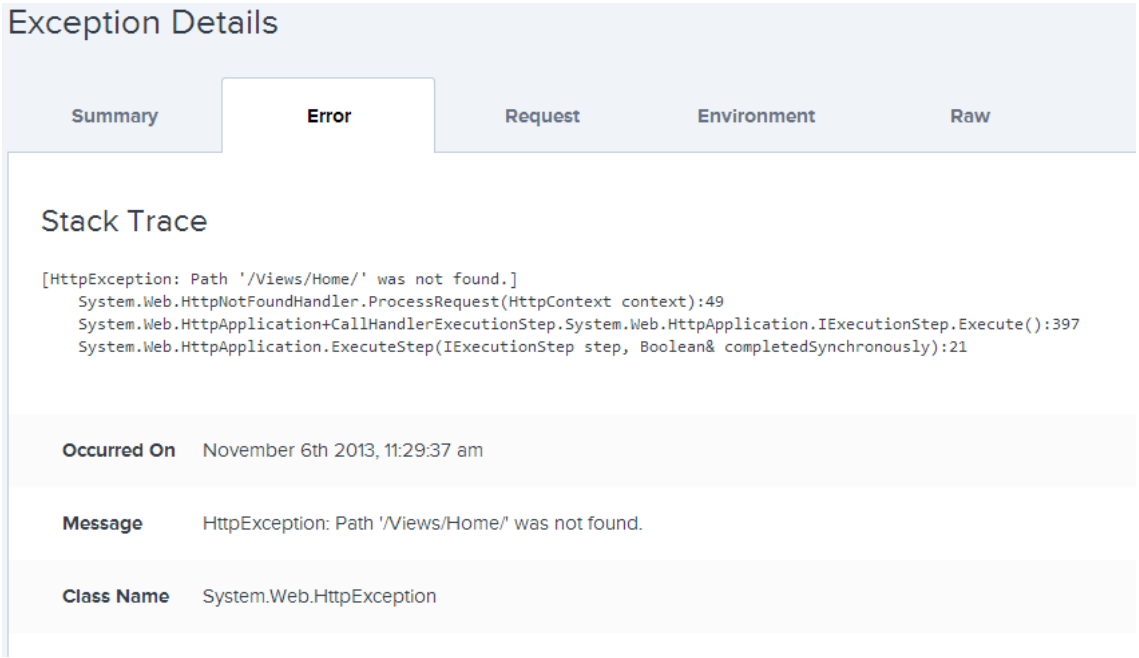

Figure 12 Raygun's exception details page

The first tab of the detailed page includes a short summary of the occurred error. The second tab contains callstack information, exception message, occurred time and class name. The third page has all the available request based data of the error for example: form variables, header values, url, user-agent data and server variables. The fourth tab contains all environment related data: OS version, computer architecture, available memory, CPU, OS and the used web browser. The last tab of detailed error page contains the raw data of the occurred error with some basic text formatting and colours.

4.2.2 New Relic

New Relic is another SAAS based error reporting tool, however they have taken different approach compared to Raygun. New Relic is monitoring the whole computer, this includes performance monitoring, CPU usage, memory usage, remaining disk space and RAM and Events including: errors, alerts, deployment statuses and thread profiler.

The installation procedure of New Relic is different from Raygun in a way, because it needs to have New Relic agents installed on the server. New Relic has x86 and x64 windows installer packages available for the agents. After agents are installed, their services need to be started and IIS restarted. New Relic also

offers .NET API that comes as NuGet package so developers can send errors manually to the service, installation of this NuGet package is very similar to Raygun's approach and therefore it will not be described here.

New relic can show which pages are taking the most time to load as well as which pages are generating most errors. Figure 13 shows an example of new relic's application monitoring.

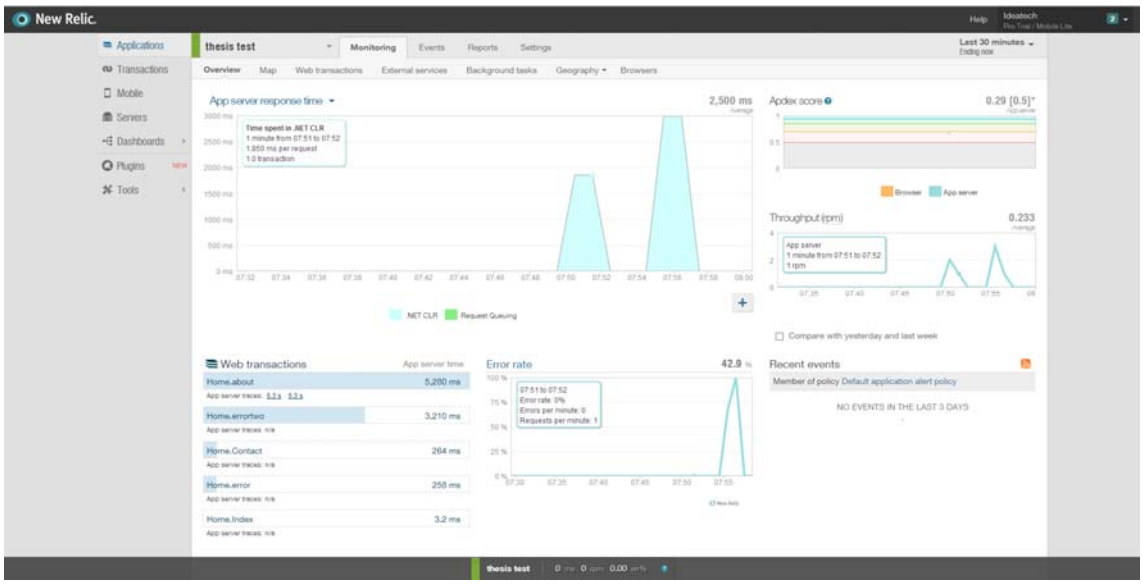

Figure 13 New Relic's application monitoring page

New Relic's errors detail page is not comparable to Raygun's, because it is lacking some information for example all browser specific data is missing and with New Relic you cannot mark error as resolved you can only delete or hide all similar errors. New Relic can be used to error reporting, but it seems to focus more on monitoring the server, however it has some nice graphs that give the overall picture of the project quality including error rate, apdex score and avg server response times etc.

4.3 Best practices in error handling

Error handling will allow the application gracefully to handle errors and display error messages properly. Error logging is usually a part of error management and allows developers to find and fix occurred errors (Asp.Net, Asp.Net Error handling). A well-designed error handling can make the program more robust and less prone to crashing, because the application handles such errors (MSDN, Best

Practices for Handling Exceptions). There are multiple ways how error handling can be implemented.

4.3.1 Defects do not contain required information

Sometimes defects do not contain the required information that developers need to find the root cause for a defect. This is usually caused by poor error handling and there are many reasons why this can happen.

```
try
{
    //Do some operation that can fail
}
catch (Exception ex)
{
    //Do some local cleanup
    throw ex;
}
```

Figure 14 poor example of throwing exception

With the above code shown in figure 14, the callstack is truncated and only contains error information starting from the method that failed. The origin of the exception will always appear to be in application code. This is not always the case. Exceptions can originate in various external systems and eventually get thrown as CLR exceptions. SqlException and SoapException are a good example of these. The SqlException can be generated at the Database driver or Data Access Layer and therefore is not an application level problem. SoapException can generate outside of the process boundary and passed into the CLR as a general SOAP exception.

```
try
{
    //Do some operation that can fail
}
catch (Exception ex)
{
    //Do some local cleanup
    throw;
}
```

Figure 15 Better example of throwing exception

Figure 15 shows a solution to the problem described above. Instead of using throw ex, simply using throw retains the whole callstack of the occured error (Anujvarma, C#, .NET Exception Handling Best Practices).

4.3.2 Exceptions to avoid in .NET

Microsoft's .NET documentation identifies practices to avoid when throwing exceptions: First, exceptions should not be used to change the flow of a code block as part of ordinary execution. Exceptions should be used only to report and handle error conditions. Second, Exceptions should always be thrown instead of returned as a return value or parameter. Third, The following exceptions: System.Exception, System.SystemException, System.NullReferenceException, or System.IndexOutOfRangeException should not be intentionally used in source code. These exceptions are meant to be thrown by the .NET framework itself. For example instead of throwing NullReferenceException use ArgumentNullException in case the required parameter is null. Fourth, do not create exceptions that can be thrown in debug mode but not in the release mode. To identify run-time errors during the development phase, use Debug Assert instead of just throwing exceptions (MSDN, Creating and Throwing Exceptions). Using debug assert can be handy if run-time debug details are needed, because when source-code is compiled to release build, debug assert calls will be removed so they do not affect the performance of the software. (MSDN, Assertions in Managed Code.)

4.3.3 Naming of exceptions

One thing that .NET community prefers is to name all exceptions end with Exception. This may seem a minor thing, but codes are not written to computers but for human readers. Naming of variables is one important thing even in error management. Proper naming provides abstraction, but also higher maintainability of code. Readability is important. (MSDN, Designing Custom Exceptions.)

5 Reflection

I found Microsoft Operations Framework, when I was actually looking for information about ITIL (Information Technology Infrastructure Library). Microsoft Operations Framework was chosen to this thesis instead of ITIL, because I found a lot of similarities between it and the current way of working in Visma Solutions. Going through MOF problem management process helped me to understand better the big picture of what is going on at each stage of the defect management process. Reading about .NET

MOF provides a detailed and accurate researching problem process, but at the same time it leaves it for the organization to decide how they implement it. Another good side of MOF is that most of the models are independent and organizations can freely choose which parts they want to implement, without needing to implement the whole framework. One of the negatives of MOF is that there is not as much literature available compared to ITIL.

Raygun and New Relic were chosen to this thesis, because of simple installation process and because they both represent different approach of error reporting. Also Airbrake error reporting tool was tested, but because of some incompatibility issues I was having with it in .NET environment it was dropped out from the research. Raygun seemed to be more suitable for defect management in Visma Solution Severa's context than New Relic, because of better diagnostic data and resolution possibilities it provides for exceptions.

This project did not take a position on disaster recovery and that would be an obvious target for further research. Also more comparison between ITIL, MOF and COBIT process models could be done to get the best possible end result. Comparison was left out of the project, because it requires a lot of time and repetition to go through all these models.

6 Summary

The main goal of this thesis was to create a model for the defect management process and find a way to handle a large number of defects. This was accomplished by doing a research about MOF Researching the problem process. This process model seems to fit rather well to Visma Solution, because their current way of working does not differ that much from it. The tool Raygun was chosen to be tested in the future to see if it could actually help with managing the defects. However, because testing the tool and the process in organization will take time there will not be any results to show in this thesis. In the end as the goal was to improve the defect management it seems that this project was successful.

References

1. Korhonen, K. 2012. Supporting Agile Transformation with Defect Management in Large Distributed Software Development Organisation. Doctoral dissertation. Tampere University of Technology. Publication 1032.

2. Pultorak, D. 2008 MOF 4.0: Microsoft Operations Framework 4.0

3. Six Sigma. 5 Why's http://www.isixsigma.com/dictionary/5-whys/. Read 9.11.2013.

4. MSDN. 2008. Root Cause Analysis for Software Problems. http://blogs.msdn.com/b/nickmalik/archive/2008/03/31/root-cause-analysis-for-software-problems.aspx. Read 16.10.2013.

5. YouTube. Root Cause Analysis (RCA) using Ishikawa/Fishbone Diagrams. 2010. http://www.youtube.com/watch?v=Kz5Pr8aPKtw. Read 5.10.2013.

6. Microsoft Office. Create a fault tree analysis diagram. http://office.microsoft.com/en-us/visio-help/create-a-fault-tree-analysis-diagram-HP001207628.aspx. Read 10.10.2013.

7. Usenix. Finding Similar Failures Using Callstack Similarity. https://www.usenix.org/legacy/event/sysml08/tech/full_papers/bartz/bartz_html/. Read 1.11.2013.

8. McConnell S. 2004. Code Complete: A Practical Handbook of Software Construction. Second Edition.

9. Amartester. Bugs per line of code. http://amartester.blogspot.fi/2007/04/bugs-per-lines-of-code.html. Read 9.11.2013.

10. Stackoverflow. When to throw exception. http://stackoverflow.com/questions/77127/when-to-throw-an-exception. Read 1.11.2013.

11. YouTube. Raygun September updates - charts, filtering & sorting. http://www.youtube.com/watch?v=rSlS5Ajm4g0. Read 6.11.2013.

12. Asp.NET. Asp.Net Error handling. http://www.asp.net/web-forms/tutorials/aspnet-45/getting-started-with-aspnet-45-web-forms/aspnet-error-handling. Read 1.11.2013.

13. MSDN. Best Practices for Handling Exceptions. http://msdn.microsoft.com/en-us/library/seyhszts.aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-1. Read 1.11.2013.

14. Anujvarma. C#, .NET Exception Handling Best Practices. http://www.anujvarma.com/c-net-exception-handling-best-practices/. Read 9.11.2013.

15. MSDN. Creating and Throwing Exceptions. http://msdn.microsoft.com/en-us/library/ms173163.aspx. Read 6.11.2013.

16. MSDN. Assertions in Managed Code. http://msdn.microsoft.com/en-us/library/ttcc4x86.aspx. Read 6.11.2013.

17. Raygun. Exceptional Error Tracking. http://raygun.io/. Read 7.11.2013.

18. New Relic. New Relic monitoring application. http://newrelic.com/. Read 7.11.2013.

19. MSDN, Designing Custom Exceptions http://msdn.microsoft.com/en-us/library/vstudio/ms229064(v=vs.100).aspx Read 11.11.2013