CHALMERS

Detecting usage patterns

A study on developmental benefits achieved through detecting usage patterns in applications by using a logging component

Master of Science Thesis in Software Engineering

Jacob Larsson
Thorvaldur Gautsson

Department of Computer Science and Software Engineering
Chalmers University of Technology
Gothenburg, Sweden, June 2015

Detecting usage patterns

A study on developmental benefits achieved through detecting usage patterns in applications by using a logging component

JACOB LARSSON
THORVALDUR GAUTSSON

Supervisor: MIROSLAW STARON
Examiner: MATTHIAS TICHY

Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Gothenburg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Gothenburg, Sweden, June 2015

Acknowledgements

Many people supported and helped us during the project. We wish to express our gratitude to Oscar Lund and Henrik Fagrell for their constant and invaluable feedback. They gave us suggestions, set up meetings with various companies and enthusiastically encouraged us throughout the whole process. We are very grateful for their help.

Our thesis supervisor, Miroslaw Staron, offered us help, support and guidance throughout the project. We are also very grateful for his help.

We would like to extend our thanks to the employees of Diadrom Systems AB. They welcomed us as one of them and invited us to social gatherings, company meetings and treated us well throughout our time at the company.

Finally, we would like to thank everybody who participated in our workshops, held meetings with us, or gave us feedback in more informal settings.

Jacob Larsson & Thorvaldur Gautsson

Gothenburg, Sweden

Monday 1st June, 2015

Abstract

Background: The inherent complexity of software development has in recent decades necessitated the advancement of engineering methods in order to facilitate the construction of software applications in an efficient manner. For this reason, many different methodologies and processes have been developed. Problems still remain however, for example in the field of usage pattern analysis. In many cases there is a gap between the assumed use of an application and its actual use. Understanding e.g. which features are used, when they are used, and whether they can cause crashes can enable developers to better focus their efforts towards productive development and thus contribute towards further engineering knowledge which can then be applied in software development.

Aim: To explore how logs can be used to understand usage patterns in order to improve a software application. This is achieved by building a logging component which can with a minimum effort be integrated into an existing application.

Method: The study presents and analyzes a logging component developed using the design science research methodology. The process consisted of five steps, in which the problem domain was analyzed, requirements were identified, a prototype was designed, built and evaluated, and finally conclusions drawn. Evaluation took place through quantitative measurements as well as a survey which was posed to employees at four different companies. After being completed, the logging component was then additionally assessed through a case study in which the logging component was integrated with two different applications and semi-structured interviews held with participants at two workshops.

Results: The research found that a logging component that visually presents usage patterns is useful for developers in order to to aid further development of an application. Participants of the workshops as well as those who were surveyed considered the most useful information to be data on exceptions and crashes, but data on feature usage was also considered important. All developers interviewed and over 90% of the people surveyed knew one or more projects which would have benefited from an external logging component with the functionality presented in this research.

Conclusion: The study confirms that a visual presentation of logs by an external logging component can aid developers in further developing an application through presenting information about exceptions and crashes as well as features used by users of the application. An external logging component of the type developed in this research can therefore lead to faster and more efficient development. Recommended future work includes extending the study to cover applications written in more programming languages, as well as testing the logging component through an experimental study.

Keywords: Logging, usage patterns, features, code injections, data analysis

Contents

1

Introduction

The objective of most software development projects is to release or to update a functional software product, within a reasonable amount of time after the project started, within a certain budget, and which satisfies a particular set of requirements [1]. Since the birth of software engineering as a discipline however, software development has become more and more complex, which makes attaining such an objective increasingly difficult [2].

One of the main reasons for the increasing complexity in the field is the rapid progress of computer hardware development. It is likely that no other technology in human history has progressed as rapidly. Indeed, the number of transistors which can be put cost efficiently on an integrated circuit has doubled every two years for many decades [3]. Edsger W. Dikjstra diagnosed this as the chief cause of increased complexity when he wrote in 1972 that "as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming had become an equally gigantic problem." [4]

Gigantic problem or not — complexity now appears to be a fundamental property of software development [2] and it has many different manifestations. One example is a general decrease in the performance of applications, which inspired Niklaus Wirth to famously remark in 1995 that *"software is getting slower more rapidly than hardware becomes faster"* [5]. Performance decrease can e.g. be caused by developers adding new and unnecessary features which make the source code more intricate and convoluted than needed. This often results in less stable applications which are consequently more prone to crashes.

At a fundamental level, complexity can also prevent developers from correctly grasping how an application should preferably function according to its users. The more features and possibilities an application offers, the more difficult it is to know how it is used. This

means that there often exists a gap between the assumed use of an application and its actual use. In some cases an application can have a vast quantity of features, but only a limited amount which constitutes a majority of the total usage time. In other cases, the features most commonly used may be different ones from the preconceived notions of the developers.

There are a few ways in which the gap between the assumed use of an application and its actual use can be bridged. This thesis concerns itself with one approach: logging user behavior. For this purpose, the research question investigated is *how can an external logging component be used to aid in the process of software development by providing developers with information about usage patterns?* The term *usage pattern* is defined in the study as *how users use an application at a high level*, i.e. how they interact with the application through mouse clicks and keyboard input, which pre-defined features of the application they use, when and how often they use those features, and when and how they cause exceptions. The logging component scrutinized in the research therefore requires the developers of an application to determine and specify what exactly defines a feature. The usage pattern is the relationship between the users of the application and those features.

In order to answer the research question, an external logging component which can report usage patterns to a developing organization was constructed from scratch. The logging component was thereafter evaluated and the results of the evaluation analyzed. The study was conducted in cooperation with the IT company Diadrom Systems AB, which is primarily active in the area of diagnostics for high-tech products. Diagnostics — which includes logging — and data analysis, is an area of sustained and long-term growth due to the expanding technology content of various products and their increased complexity.

This study documents the investigative process and its results. The following chapters of the thesis describes the background for the problem, related work within the field, and then introduces the research methodology and design which was used to investigate the research question. After the results are presented, a discussion on ethical issues as well as various benefits of usage pattern detection follows. Finally, conclusions are drawn and future work is suggested.

2

Background

The thesis includes a range of different topics from different fields and areas which the reader might not be familiar with. In this chapter, concepts and tools which were used in the development of the logging component but which the general reader cannot be presumed to know are presented. Academic topics and fields of research which are related to the study are likewise explained. This information is intended to give sufficient context for the study and to place it in a frame of reference.

2.1 C#

C# is an object-oriented programming language which has its roots in the C family of languages. The C# language was developed at Microsoft within the .NET software framework but was later standardized by ECMA International as the *ECMA-334* standard and by ISO/IEC as the *ISO/IEC 23270* standard [6].

C# is a popular programming language. A recent study of the popularity of programming languages in 100.000 open source projects found that by measuring lines of code, C# was the 8th most popular programming language [7]. This is despite the language having only been introduced in 2001 and generally being more commonly used in corporate environments than typical open source projects.

2.2 WPF

Windows Presentation Foundation (hereafter: WPF) is a graphical presentation system for applications which run on the Microsoft Windows family of operations systems. WPF allows developers to construct applications that incorporate vector graphics, interactive animations, and media-rich front ends [8]. WPF employs a declarative markup language called XAML to define various interface elements and link them together. XAML is based on the XML markup language.

WPF is intended by Microsoft to replace older graphical presentation systems — and could be said to be the *heir apparent* to the older Windows Forms presentation model. [8].

2.3 Intermediate Languages

Before a code written in C# is translated into machine code, the C# compiler first translates the code into an intermediate language (hereafter: IL), which as the name suggests is a language that lies between the high-level language the programmer writes code in, and machine code executed by the computer. The purpose of an IL is to translate the code into a form which is suitable for transformations in order to optimize the code. An IL can also function as the lowest level programming language that can reasonably be read by humans before the code is translated into a continuous string of numerical machine code.

The IL which the C# compiler translates into is called the Common Intermediate Language (CIL) which is a specification constructed by Microsoft for the .NET framework. There are other intermediate languages available; the GNU Compiler Collection (GCC) uses several different ILs for instance. Only the C# intermediate language (i.e. CIL) will be considered in this study.

To illustrate CIL, a C# program and its corresponding translation into CIL are shown in code snippets 2.1 and 2.2.

```
using System;

class Program
{
  static void Main()
  {
    int i = 0;
    while (i < 10)
    {
      Console.WriteLine(i);
      i++;
    }
  }
}
```

Code snippet 2.1: C# code

```
.method private hidebysig static void Main() cil managed
{
  .entrypoint
  .maxstack 2
  .locals init ([0] int32 num)
  L_0000: ldc.i4.0
  L_0001: stloc.0
  L_0002: br.s L_000e
  L_0004: ldloc.0
  L_0005: call void [mscorlib]System.Console::WriteLine(int32)
  L_000a: ldloc.0
  L_000b: ldc.i4.1
  L_000c: add
  L_000d: stloc.0
  L_000e: ldloc.0
  L_000f: ldc.i4.s 10
  L_0011: blt.s L_0004
  L_0013: ret
}
```

Code snippet 2.2: IL translation of the C# code in code snippet 2.1

2.4 Weaving and Fody

Weaving is a technique that can be used in programming languages that use ILs or other forms of pre-compilation steps. The technique is closely linked to *aspect oriented programming* in which it is common to combine many different components into a single application at compile time [9]. The weaving concept refers to injecting some type of functionality into the code of an existing program before it is compiled. This can be done

for example by injecting code into a project after it is translated into an IL but prior to it being transformed into machine code.

By using weaving, it is possible to achieve better modularization through the separation of concerns. For example, injecting calls to a logging component during compile time enables developers to keep their code separate from the logging code, and in effect to not have to think about the logging code at all.

Fody is a tool which can be used for weaving projects developed in the .NET framework by Microsoft. Fody simplifies the weaving mechanism by providing a structure to simplify the process of manipulating the CIL of a .NET application. Fody therefore makes it possible to e.g. weave together two or more projects without intricate knowledge of how the weaving process itself works.

2.5 Visual Studio and NuGet

Visual Studio is an integrated development environment (IDE) from Microsoft meant for developing applications for the Windows family of operating systems. The IDE supports a number of different programming languages, but is mainly used for those which were designed by Microsoft, i.e. C#, F#, VB.NET and so on. Visual Studio can additionally be used for developing web applications.

NuGet is the package manager for the .NET framework developed by Microsoft. NuGet automates the process of downloading, installing, configuring and updating packages for a given project in Visual Studio. If a package is dependent on another NuGet package, it is defined in the NuGet. If a NuGet is installed but a dependency is missing, the package manager will install the missing dependencies automatically.

2.6 NoSQL and MongoDB

NoSQL stands for *not only SQL* and concerns database management systems that employ other mechanisms of storage than the traditional table based SQL relational databases. NoSQL databases can have advantages over SQL databases, for instance they generally process data faster than relational databases - but they also have disadvantages, such as e.g. query complexity and overhead [10].

One advantage of NoSQL systems is that unlike traditional relational database management systems, they have the ability to scale horizontally [11]. This means that it is possible to add a new server to the database cluster without generating much overhead

as would be the case with most relational databases, which force the different servers to communicate frequently.

MongoDB is a NoSQL database management system. It uses a JSON-like format which is called BSON for storing data and JavaScript for querying the database. Various programming languages such as e.g. C# have so called MongoDB drivers in which native queries can be written.

2.7 Dashboards

A dashboard can be defined as a measurement system for visualizing indicators which have been chosen in order to adequately represent data [12]. The purpose of a dashboard is to display the most important information to achieve a certain objective, arranged and concentrated so that it can be glanced at relatively quickly [13]. In addition to visualization, dashboards can contain calculations derived from the data, performance indicators, textual representations of data, or other forms of presentation.

Dashboards can be of value to many different stakeholders in an organization. They can for example be used by upper management to gain information which can help to take high level decisions. They can also be used by software developers to monitor e.g. the quality of a particular software product [12]. An example of a dashboard is displayed in figure 2.1.

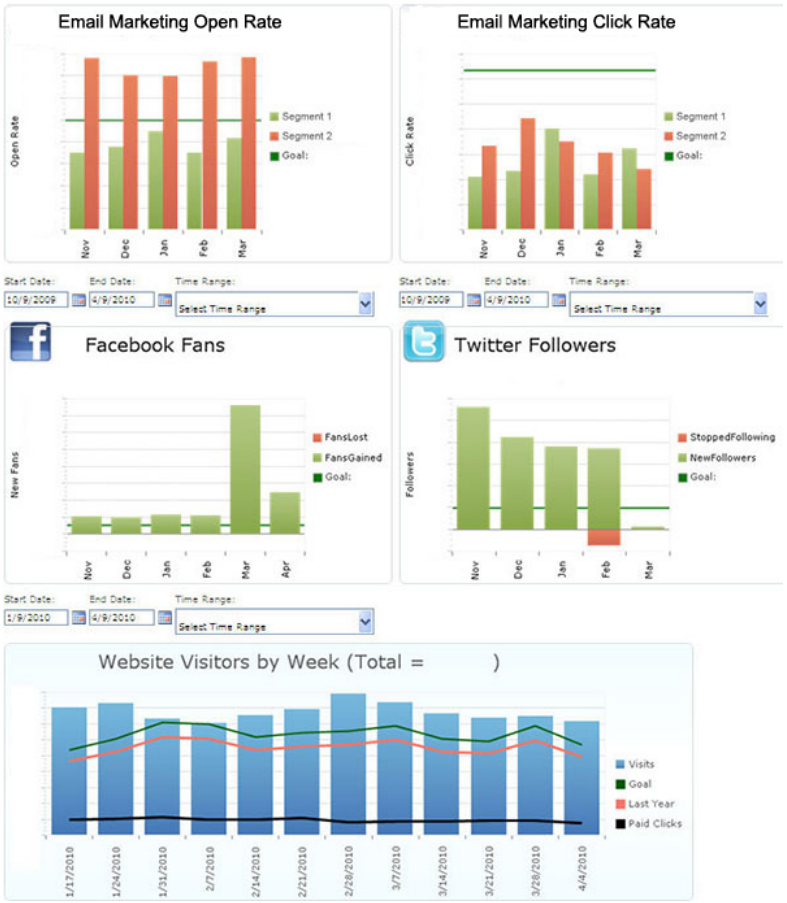

Figure 2.1: Example of a Dashboard

2.8 Business Intelligence

The area of Business Intelligence concerns analysis of large amounts of data and providing information which can assist in the business decision making process [14]. Two areas within Business Intelligence relevant to the study are Data Mining and Process Mining. Data Mining is about finding patterns and relationships between data. Process Mining focuses more on analysing and finding process relationships [15].

2.9 Aspect Oriented Programming

Aspect oriented programming refers to a particular programming paradigm which has as its main aim the goal of increasing modularity within the source code of an application by separating concerns — and in particular so called cross-cutting concerns — into separate components. Cross-cutting concerns are concerns which by their nature span many different modules which makes it difficult to create separate and independent entities out of them. Examples of cross-cutting concerns are: security, synchronization, persistence — and logging [16].

In aspect oriented programming, different aspects are created and then integrated together at compile time in a so called *weaving* step.

2.10 Visual GUI Testing and SikuliX

Visual GUI testing is a script based test technique which uses image recognition to interact with the GUI components of an application [17]. A script in the visual GUI testing style might for instance locate a particular image, write text in a box below it, or find a certain button and click it. In this way it is possible to mimic user behavior as the script locates the elements which have been predefined by the tester.

However, little research has been done within the field and industrial use is small [17]. The research done within the field seems to suggest that while the technique is promising and can in many cases yield positive results, but that there are many challenges within the field, for instance script maintenance costs, and how to support test execution [17].

There are many fields of contact between visual GUI testing and the research conducted in this thesis, just as there are between logging and testing in general. It is for instance possible to find defects in a problem both by testing it and by logging user behavior and analysing which user actions lead to crashes. In this sense, a logging component could decrease the need for visual GUI testing. If the logs obtained from the logging component yield similar data to that gained by visual GUI testing, then it means less testing will probably be needed.

SikuliX is a visual GUI testing automation tool which executes pre-defined tasks in sequential order. It uses python as a scripting language for defining the tasks to be executed. A task can consist of keyboard inputs, mouse clicks etc. SikuliX can use image recognition to locate objects on the screen and then executing a task on that image, e.g performing a mouse click or waiting for a certain image to appear before another task is executed. As an example, code snippet 2.3 writes "Hello world!" in notepad and then closes the file without saving it.

```
openApp("notepad.exe")
wait(1)
type("Hello World!")
type(Key.F4, KeyModifier.ALT) # Close Notepad
type(Key.TAB) # Select "Don't Save"
type(Key.ENTER)
```

Code snippet 2.3: Python code executed by SikuliX

2.11 Dynamic Program Analysis

Dynamic program analysis is a term which refers to analysis of the properties of a running application. This is in contrast to *static program analysis* which considers the source code of an application without actually executing it [18]. Static program analysis can help to maintain code quality and should be integrated with the build process of an application, whereas dynamic program analysis happens after the application has been built, which makes it possible to evaluate for instance temporal issues.

Dynamic program analysis — unlike its static counterpart — cannot prove whether an application satisfies a given property, but can detect violations of properties which happen during runtime [18]. Dynamic program analysis can additionally provide other useful information to the developers of an application.

The logging component which is the subject of this research employs dynamic program analysis through reporting how an application is used after it has been compiled and shipped to its users.

2.12 Profiling of programs

Profiling of programs is a type of dynamic program analysis in which variables such as e.g. the time complexity of an application or its memory usage are measured. The idea is to study the behavior on a program based on the previous times it has been used. Each of these previous usage times utilize a different set of input parameters or files and then the information required for analysis is collected and evaluated [19].

In order to profile a program, a profiler tool which is integrated with the program is needed. The information then obtained by profiling a program is generally used to help with application optimization.

3

Related Work

The study touches on many different fields and areas of research. In the *background* chapter those areas were discussed, and various terms and concepts which the reader might be unfamiliar with were explained. In this chapter, the work of others within fields considered relevant to the study is presented. In cases where design recommendations were given, such as by Staron et al. regarding dashboards [12], they were generally followed for the design of the logging component.

3.1 Dashboards

Staron et al. studied in 2013 how three different companies used dashboards to monitor and control quality aspects of software products under development [12]. The study gave recommendations based on their analysis for other companies which wished to introduce dashboards in their projects. Among the recommendations given for constructing the dashboards were:

- The number of indicators in a dashboard should be limited. The term indicator refers to an observed value of some sort in this context
- The indicators selected should match the ones desired by stakeholders who have the mandate and means to act in the project
- Multiple parts of the developing organization should be involved

The recommendations for choosing the right indicators and measures were that it was imperative to focus on the product, and to focus on the end result. The reason for putting

the focus on the product instead of a process or project is that companies generally do not sell their projects or processes but rather their products. The same reasoning also explains why focus should be put on the end result. The end result is what companies provide to their customers — not the intermediate versions which precede it. Monitoring artifacts such as requirements or architecture instead of the end result is therefore not sufficient and can lead to short-sighted decisions [12].

Dashboards can have many different indicators. Staron, Meding & Palm studied in 2012 the value of indicators and concluded that an indicator called *release readiness* was particularly important [20]. The indicator was developed in an action research project conducted at Ericsson AB. Further work was done on indicators by Staron et al. in 2013 by studying code stability indicators at three different software development companies: Ericsson AB, Saab AB and Volvo Group [21].

3.2 Visual GUI Testing

Although some research has been conducted within the field of Visual GUI testing, it is still a relatively new field. There are many fields of contact between visual GUI testing and visual logging, as there are between logging and testing in general. One important relationship between these two fields is that a logging component could decrease cost for testing and perhaps decrease the need for visual GUI testing; if the logs obtained from the logging component yield similar data to that gained by visual GUI testing, then its importance decreases.

Börjesson and Feldt evaluated in 2012 two tools for automated visual GUI testing on software system developed by a Swedish aerospace company. They found that visual GUI testing can perform better than manual system test practices and that it furthermore has benefits over alternative GUI testing techniques. They stated however that visual GUI testing still had challenges which had not been addressed [17].

Emil Algéroth presented in a paper in 2013 a proof of concept for a novel technique which combined GUI based testing, visual GUI testing and random testing [22]. Evaluation of the technique showed that the technique was applicable for both functional and quality requirement conformance. Furthermore, the results showed that there existed a need in the industry for the technique.

Liebel et al. evaluated in 2013 how GUI based testing is performed in industrial practice by conducting a multiple-case study at six different software development companies. They found that manual GUI based system testing is widespread, but automated GUI based testing exists only on a small scale. The study also found that there were a number of problems with GUI based testing, such as e.g. tool limitations and high costs [23].

3.3 Post-deployment data collection

Backlund et al [24] studied post-deployment data collection by conducting a case study on a web-based portal system. They began by identifying which data needed to be collected, then implemented the data collection and finally performed an experiment by comparing the collected data with answers from test subjects. The data used was collected through aspect-oriented programming and included various user actions, such as e.g. button clicks and task completion times. They found a correlation between the survey data and the measurements. For instance, both the survey and the measurements suggested that a task called *change password* was the most difficult task for the users to perform.

4

Research Methodology

The study was conducted over a six month period, starting in January and ending in June of 2015. The study utilized a mixed design which combined two research methods: design science research and a case study. The design science research approach was used to construct a logging component which serves as a proof of concept for detecting usage patterns in external applications. After the logging component had been constructed it was integrated with a prototype application and then assessed. Further evaluation was conducted in a case study in which both qualitative and quantitative aspects were considered.

The logging component was developed in cooperation with Diadrom Systems AB (hereafter: Diadrom) at their company office in Gothenburg. Representatives from Diadrom provided continuous feedback, both in formal as well as informal settings. Employees from the company were also part of the requirement elicitation and evaluation processes.

4.1 Research Question

The research question which the study analyzed was:

How can an external logging component be used to aid in the process of software development by providing developers with information about usage patterns?

In this context the term *usage pattern* is defined as *how users use an application at a high level*, i.e. how they interact with the application through mouse clicks and keyboard input, which pre-defined features of the application they use, when and how often they use those features, and when and how they cause exceptions to be executed. The logging

component scrutinized in the research therefore requires the developers of an application to determine and specify what exactly defines a feature. The usage pattern is the relationship between the users of the application and those features.

4.2 Research Structure

In order to investigate the research question, a custom made logging component which could be integrated into an existing application was built from scratch. A research methodology which combined design and development with research and analysis was therefore needed for the study. The methodology chosen for this purpose was design science research, where a model proposed by Vaishnavi and Kuechler was used as a guideline [25]. After development had ceased, the finished logging component was then evaluated in a case study using the model proposed by Runesson and Höst [26] in order to obtain credible results.

4.2.1 Design Science Research

Design science research is by its nature a problem solving process, in which knowledge and understanding is acquired through building an artifact and then applying it in a specific problem domain. That is, design science research revolves around creating an innovative, novel, and purposeful artifact in some specific domain for the purpose of solving a problem by using the knowledge gained from building and applying the artifact [27]. The artifact which was to be developed — i.e. the logging component — was an innovative idea to address a problem, and after the artifact was completed, it was used to research whether it could aid in the development of an application. For this reason, design science research was an appropriate methodology, as it aims at creating a part of the phenomenon which is under investigation.

The design science research model adopted for the study describes a process which is composed of five steps [25]. The idea is that following these steps enables practitioners to approach problems in a systematic fashion and to continuously progress towards more specific solutions. Although the process is structured in a sequential order, it is not necessary to follow it consecutively. Each step should increase the knowledge about the research at hand, which means that in some cases it is logical to e.g. start at a different step than the first one, or move backwards from one step to another [28]. The steps of the process are:

1. Awareness of the Problem
2. Suggestion
3. Development
4. Evaluation
5. Conclusion

The process, the flow between the steps, and the output of each step is illustrated in figure 4.1. The Development, Evaluation and Conclusion steps can contribute new knowledge, which can lead to an increased awareness of the problem, and the Conclusion step can lead to new design theories. Each step has a distinctive output: a proposal, a tentative design, an artifact, performance measurements and finally results.

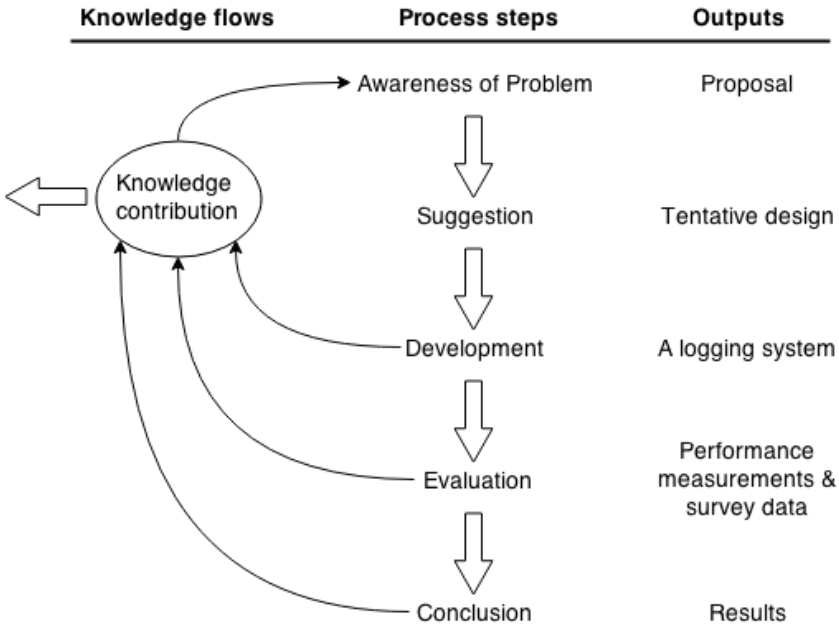

Figure 4.1: The design science research model used, based on the model by Vaishnavi and Kuechler

4.2.2 Case Study

After the design science research phase was over, the logging component was further evaluated through a case study. A case study is an empirical method which has as its objective the investigation of contemporary phenomena in their appropriate context [26]. The data collected can be either quantitative, qualitative or a combination of both types.

In the case study phase, the limited evaluation which had taken place within the design science research phase was extended to cover previously developed applications which were already in use.

The evaluation in the case study was conducted on two applications which had already been developed and were both in actual use. The assessment involved both qualitative and quantitative aspects, using both metrics which were measured as well as structured group interviews.

The case study process model which was used to evaluate the logging component consists of five steps. As is the case with design science research, the case study research methodology is a flexible style of research and proceeding from one step to the next is often not a sequential process. Iterating between the different steps of the process is relatively commonplace [26]. However, since the case study conducted in the research centered on evaluating a finished version of an application and had a clear objective from the beginning, the steps of the process were followed consecutively and without any overlap.

The steps of the process are:

1. Objectives defined and case study planned
2. Preparation for data collection
3. Collecting evidence
4. Analysis of collected data
5. Reporting

4.3 Research Workflow

The workflow of the research consisted of three main stages. Figure 4.2 illustrates the entire workflow process.

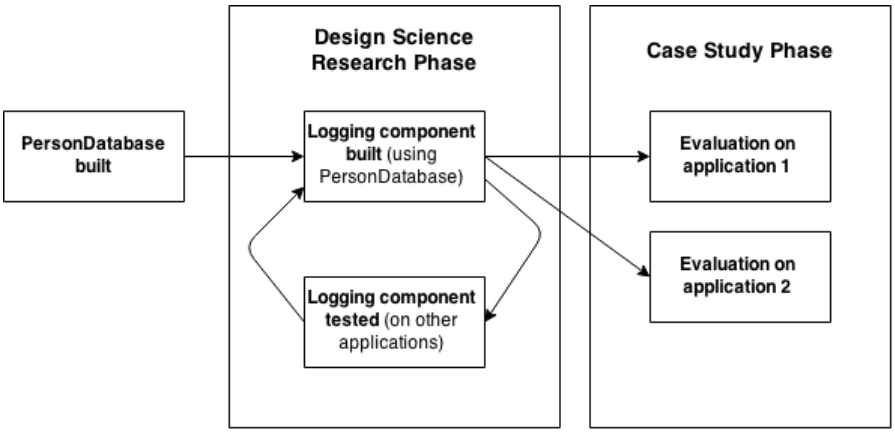

Figure 4.2: Study methodology

Before work on the logging component began, a software application called *PersonDatabase* was built in order to have an application which the logging component could be built around. This application had the purpose of storing information about employees of a company.

After PersonDatabase was completed, the design science research phase commenced. The logging component was developed from scratch, with feedback coming from many different sources. While the logging component was developed primarily around PersonDatabase, it was constructed to be as comprehensive and broad as possible. For this reason it was also regularly tested on the source code of various other applications while being built.

After the logging component was finished it was then evaluated on two different applications in a case study.

5

Research Design

The study consisted of two different phases, as previously described in chapter 4. In the design science research phase, a logging component was iteratively built and then evaluated. In the case study phase, the logging component was further analyzed through integration with external applications which it had not been previously tested on.

5.1 Design Science Research

The workflow which was followed in the design research phase is illustrated in figure 4.1 in chapter 4. The following sections describe the steps of the process.

5.1.1 Awareness of the Problem

A major problem in software development is that it is often difficult for developers to know how an application is used by its users. There can be several reasons for this. The requirements elicitation process may for instance not correctly have predicted what the users want, or the user needs may have changed as the application has evolved. In these cases a gap between the assumed use of an application and its actual use has emerged. This gap can have particularly detrimental effects in cases where only a limited set of the features of a system constitutes the majority of the total usage time. It can for example lead to focus being put on the development of features which do not contribute much business value, instead of the few features which are truly important.

In order to effectively improve an application, it is therefore important to close this gap

by obtaining information pertinent to the actual usage pattern of the application. Such knowledge can reveal in which way the application should be changed and aid in the prioritization of development work.

There are three main empirical methods which can be used for this purpose:

- User interviews
- User observations
- User logs

The first two approaches both have their disadvantages. Interviewing users is a laborious process [29] and can lead to incorrect conclusions being drawn. The interviewees may for example have incentives to provide incorrect information if they want to appear smart to the interviewer. Another possibility is that the interviewer does not have enough domain knowledge and as a consequence asks the users the wrong questions. Preparing and conducting interviews is also an expensive process which may have to be repeated for every new change which is made to the application.

Observing a user is likewise problematic and can lead to unreliable results. The so called Hawthorne effect, which states that individuals generally improve or modify an aspect of their behavior in response to their awareness of being observed [30], is particularly relevant in this context. Furthermore, it is difficult to observe users while they use the application in their normal environment — e.g. at home or at their workplace. As is the case with interviews, the process of observing users additionally requires a considerably amount of resources, and it needs to be repeated if changes are introduced to an application.

An alternative way to understand usage patterns is to log how users use the application. This could theoretically lead to more reliable results since no human to human interaction is necessary. Some problems exist with this method — building and integrating a logging component requires e.g. both time and resources. As opposed to the first two methods however, the time and resources spent will likely decrease drastically in future iterations of an application since a logging component can be reused once it been developed. If a change to the application is introduced later, the logging component will be able to provide new information about usage pattern without any greater expenditure. This is because it is only the integration step which needs to be repeated — not the actual development of the logging component. A logging component which can be reused and quickly integrated with an application would therefore be preferable.

Logging user behavior does have some disadvantages which are not present in the interview or the observation methods. Logging the usage pattern requires resources in the

form of CPU, memory and disc space. It will therefore lead to a decrease in performance for the application. Depending on the implementation of the logging component, this can affect performance in different aspects. It is however clear that there are limits to how much performance can be sacrificed, as a large decrease in performance could make the application unusable.

Another disadvantage is the ethical aspect. Logging user interaction requires explicit knowledge and allowance from the user — else it would be unethical. A logging component could also potentially capture passwords and other private information by mistake. A logging component would therefore have to be configured by the developers of the application to minimize the collection of private information. This issue is further discussed in section 7.3

5.1.2 Suggestion

After the problem domain had been analyzed, the suggested approach to the problem was to develop a logging component in order to evaluate whether such a component could in an effective way be used to improve understanding about the usage pattern of an application. Before development commenced, a few fundamental requirements had to be specified.

The requirements were collected by giving presentations and receiving feedback afterwards, conducting workshops, holding meetings — but also through informal discussions. Employees and customers of the company Diadrom were the main source of information during the requirements elicitation process. A part of the requirements were gathered before development started and more were then added alongside the developmental process. The requirements were documented through user stories which can be found in appendix E.

During the requirements elicitation process three different stakeholders were identified: Developers, managers and IT support. People in IT support would likely be interested in viewing the activities of a particular user before that user contacted support. Managers would be interested in statistics, e.g. which features of an application are most commonly used, but also how high-value customers use the application. The developers on the other hand would be interested in information about exceptions, and general application statistics.

Conducting a thorough research on all these stakeholders was not possible considering the time frame for the study. There is however a considerable overlap in the area of interest between the different stakeholders and by satisfying one stakeholder it is therefore also possible to partly satisfy the others. The focus of the research was put on the developers and the benefits they could attain from a logging component.

The requirements elicitation process led to the following criteria which the logging component was required to address:

- It should be possible to reuse the logging component
- It should be easy to integrate the logging component with an already developed application
- It should not reduce performance to a degree where it affects the usage of the application
- It should be possible to use the logging component on a computer that is not continuously connected to the remote database
- It should be possible to regulate what will be logged
- It should be possible to see how the user interacted with the GUI
- It should be possible to see which features of an application are used the most
- It should be possible to see what the user did before an exception occurred

The research focused on investigating whether logging can be a more precise way of identifying usage patterns for an application while satisfying these criteria.

5.1.3 Development

The criteria defined in section 5.1.2 and the user stories presented in appendix E were used as fundamental requirements for the logging component and its development was therefore centered around satisfying the needs documented there.

A decision was taken to divide the logging component into different components which each had its separate function within the whole project. The organization of these components changed gradually throughout the project as the logging component evolved. The development was done in an agile style with iterations where every new iteration contained improvements over the preceding one.

The final version of the logging component is presented and analyzed in chapter 6 and further discussed in chapter 7.

5.1.4 Evaluation

Evaluation on a small scale took place throughout the entire project. Every large design decision necessitated discussion and planning meetings, and smaller design decisions were often preceded by an assessment of the current state of the project. During the course of the development, presentations with a question and answer session were additionally held at four different organizations:

1. A Swedish automotive company
2. A Swedish truck manufacturing company
3. A civil authority under the Swedish Ministry of Defence
4. Diadrom Systems

These presentations were an important forum for receiving feedback on how development should proceed, as well as what the audience considered to be valuable. After the presentations, the attendees were given a survey with 10 questions. The purpose of the survey was to find out whether the logging component could aid in the process of software development by providing information about usage patterns, as well as to investigate which of its functionalities were considered most valuable and whether there were any ethical objections to the component. The survey questions can be found in appendix C and the results are presented in chapter 6.

5.1.5 Conclusion

As the PersonDatabase application which was used in the evaluation step was rather basic, further evaluation of the logging component was considered to be necessary. For this reason, a case study was conducted where the logging component was integrated with two applications which were far more complex than PersonDatabase. This is documented in section 5.2. Both applications had left the development stage and had been released to the end users.

5.2 Case Study

The finished logging component was evaluated on two applications which were considerably larger and more complicated than the *PersonDatabase* program which had been used during the design science research phase. Both of these applications were developed before work on the logging component commenced and both have been released to their end users.

5.2.1 Objectives defined and case study planned

The purpose of the case study was to see whether the conclusions drawn in the evaluation step of the design science research phase were still valid in cases with complex real-life applications.

The first application which was assessed was an application developed at Diadrom for a Swedish aerospace company (hereafter Company X and Application X). This application consists of approximately 40 thousand lines of code and is considered by employees at Diadrom to be one of the most complicated applications they have developed. For reasons of confidentiality the application cannot be described further.

The second application was an open source application named *ScreenToGif* which allows its users to record a selected area of their screen, manipulate and edit, and finally save as a gif image file [31]. The application is released under the open source Microsoft Reciprocal License which is a copyleft licence from Microsoft. As of May 15th 2015, the application has been downloaded close to 25 thousand times from SourceForge.

5.2.2 Preparation for data collection

The data collected was both quantitative and qualitative. The quantitative data was used to support the conclusions drawn from the qualitative data.

Qualitative data

In order to obtain qualitative data, two workshops were held where semi-structured interviews were conducted. Application X was evaluated through a workshop held at Diadrom which was attended by developers at the company. ScreenToGif was evaluated through a workshop held at Chalmers University of Technology which was attended by students in the Software Engineering M.Sc. programme, all of which had experience as developers in industrial companies. The complete list of questions asked in the interview can be found in appendix B.

Both workshops followed the same structure. First, a background was given to the project and the research question presented. Next the integration of the logging component and the target application was shown. The attendees were then asked a series of open-ended questions relating to how difficult they perceived the integration process to be. Thereafter a live demonstration was given to show how the logging component worked on the application which it had been integrated with. Following that, the various functionalities of the logging component were discussed and questions about its benefits were posed. The

remaining part of the workshop was then used to present open-ended questions about the logging component and ask the attendees what their general impressions of it were.

Quantitative data

Several criteria which the logging component had to fulfill had been defined in the *awareness of the problem* step in the design science research phase. These criteria were used as a foundation for constructing the measurements that were used to obtain quantitative data. The following aspects were measured:

- Time to execute a sequence of operations with or without the logging component
- CPU usage with the logging component
- The size of the database after a sequence of operations
- The size of an average screenshot

In order to obtain data which could be generalized, measurements were taken for three different applications. The applications tested were the applications used in the workshops, i.e. ScreenToGif and Application X, as well as the PersonDatabase application which was used in the evaluation step of the design science research phase. The measurements were conducted using the Visual Studio Profiler, the `db.stats()` function in MongoDB and SikuliX. The Visual Studio Profiler is a tool for analyzing performance issues in an application and gathering performance data. The MongoDB function returns statistics about a particular database. SikuliX is a visual GUI testing tool and is further explained in section 2.10.

The first aspect was measured on the target application, both with and without the logging component. To measure the time to execute a sequence of operations in the application, SikuliX was used. The tool was used to define a sequence of operations which was then executed 100 times using a loop. The time of each execution was then measured from when the pre-defined sequence started until it stopped. By doing this on the application with and without the logging component it was possible to investigate whether the logging component significantly slowed down the application. The python code which was run in SikuliX for the *PersonDatabase* application can be found in appendix D. The code used for ScreenToGif and Application X was similar.

To measure CPU usage the Visual Studio Profiler was used. Due to restricted access and technical limitations it was not possible to measure CPU usage for Application X, as permission was not granted to install the application on computers which had the measuring tools needed. The size of an average screenshot as well as the size of the database were measured using the `db.stats()` function in MongoDB.

5.2.3 Analysis of collected data

After all data had been collected it was then analyzed to determine whether the criteria that had been laid out were fulfilled. The Student's t-test was used to verify that there was a statistically significant difference between the time it took for the PersonDatabase and ScreenToGif applications to execute a sequence of operations with and without the logging component. Since the variance of the data sets for Application X varied greatly, the Welch t-test was used in that case since it performs better for data sets of unequal variance.

The null hypothesis was that there should not be a time difference in executing the sequence of operations dependent on whether the logging component was integrated with the application or not. The results of the measurements can be found in appendix D and analysis of the data can be found in chapter 6. The analysis of the quantitative data was used to support the conclusions drawn from the qualitative data.

5.2.4 Reporting

The results obtained from the analysis of the collected data are presented in chapter 6.

6

Results and Analysis

The system which was built evolved throughout the development process as requirements changed, tests revealed bugs and new information was obtained. Gradually the system began to take shape, and its usefulness continued to increase. When the development process ended, the system was then evaluated using the methods which were discussed in chapter 5. In this chapter, the final version of the logging component is presented and the results from the evaluation are analyzed.

6.1 System architecture

The logging component provides a way to log various user actions and program behavior of an application and store those logs in a remote database. Among the user actions which are logged are e.g. user clicks and button clicks. Handled and unhandled exceptions as well as method calls are additionally logged. Screenshots are taken when the user interacts with the application to make it possible to understand how the application behaved from the users' point of view. This is further explained in section 6.1.2.

The system which was developed operates by weaving logging statements into the source code of an application at compile time. Weaving is a technique for automatically injecting code into previously written code and is further explained in section 2.4. The first version of the logging component required developers to define precisely what would be logged in the source code of their application, but since this was considered unpractical, an automatic weaving functionality was soon added.

Immediately after the logging component started to generate data, the need for a graphical user interface (GUI) to view the data emerged. It became necessary to develop a GUI

both for verifying that the correct data was being logged, and to be able to view how the logging component worked. A decision was taken to completely separate the GUI component from the rest of the code. This was done for several reasons. The primary reason was that it should be made possible to change the GUI component — e.g. to a web application — without affecting the structure of the logging component itself. If the logging component were extended to cover applications written in more programming languages, it would also be possible to use an unchanged GUI component to view all the data. Another reason was to uphold the principle of separation of concerns. Finally, not all users of an application should necessarily be able to view the collected data and by separating the GUI component, only those with access to it can view the data.

The development therefore resulted in three different components:

- OzzyLogging — a logging component for logging usage patterns
- IterateDatabase — a GUI component for presenting the data
- Weaver — a weaving component for code injections

Together these three components constitute a system which defines the logging and presentation of data for C# WPF applications. The system was given the name *Ozzy*. The relationship between the components parts of Ozzy is presented in figure 6.1. The GUI component and the Weaving component are dependent on the logging component. The weaving component needs to know which methods in the logging component it should inject calls to. The GUI component requires knowledge about the model of the logging component where different types of logs are defined.

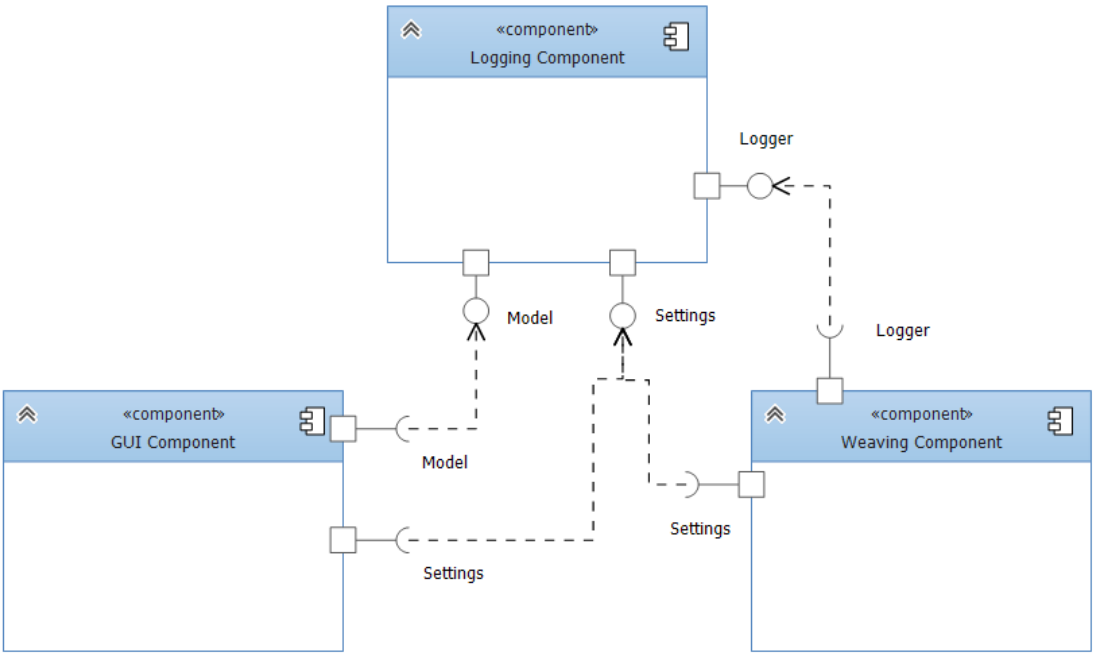

Figure 6.1: Component diagram of the system architecture.

6.1.1 Weaving Component

The Weaving component (hereafter: Weaver) only has one purpose: to inject code into a target application to connect it with the logging component. To achieve this, an open source weaving tool called Fody was used. By using Fody it was possible to define how and where code should be injected into the application without specific knowledge about the Microsoft Build Engine and the Visual Studio APIs. Since Fody had been released as a NuGet package it was possible to create a NuGet package out of the Weaving component that would automatically install Fody.

As shown in figure 6.1, Weaver is dependent on the logging component. The logging component contains a class called logger that exposes several methods which could be used to create logs. Since Weaver injects calls to those methods it is necessary for Weaver to have a reference to it. In code snippet 6.1 the log statements that is inserted into the beginning of every method is shown, with example input parameters.

```
Logger.LogMethodCall(null, LogLevel.Debug, "System.Object parameter", "
    turbo_spice.Commands", "PersonEditCommand", "Execute");
```

Code snippet 6.1: An example of a log statement which is inserted into the beginning of a method. The input parameters represent the following in ordered sequence: tag, log level, method parameters, namespace, class name, method name.

The logging component also has a number of defined attributes which can be used to put an attribute on a class or a method. The attribute is interpreted by Weaver which performs an action dependent on the attribute. An example of an attribute used on a method can be seen in code snippet 6.2, where the attribute has the purpose of instructing Weaver to avoid adding a log statement to that method. This can be useful if one method is called a disproportionate number of times which could slow down an application and affect usage pattern statistics in unwanted ways. It is also possible to specify namespaces, classes and methods in the settings if the developer do not want to use the attributes.

```
[NoMethodCallLog]
public void Execute(object parameter)
{
    Person person = parameter as Person;
    viewModel.EditPerson(person);
}
```

Code snippet 6.2: An example of using the NoMethodCallLog Attribute to prevent Weaver from injecting a log statement

All the logic for the weaving component is written in C#. However, since the injection targets the intermediate language (i.e. CIL) it is necessary to define how the CIL code should look like and how it should be changed. This means that a large part of Weaver is written in a way that is similar to CIL code. The CIL code for code snippet 6.1 is shown in code snippet 6.3. The instructions begin by pushing all parameters that are necessary for the log statement to a stack. Then the method *LogMethodCall* in the class *Logger* is called.

```
IL_0000: ldnull
IL_0001: ldc.i4 0
IL_0006: ldstr "System.Object parameter"
IL_000b: ldstr "turbo_spice.Commands"
IL_0010: ldstr "PersonEditCommand"
IL_0015: ldstr "Execute"
IL_001a: call void [OzzyLogging]OzzyLogging.Logging.Logger::LogMethodCall(
    string, valuetype [OzzyLogging]OzzyLogging.Model.LogLevel, string,
    string, string, string)
IL_001f: nop
```

Code snippet 6.3: An example of CIL code that is inserted into a method. Similar code is inserted into the beginning of all methods. The corresponding C# code is displayed in code snippet 6.1

6.1.2 Logging Component

The logging component (hereafter: OzzyLogging), was developed using C# WPF. The reason for this choice was that it is a commonly used language at Diadrom, and the company could provide applications written in C# WPF to evaluate the logging component on. To store the large amount of data which the logging component produces a decision was taken to use a NoSQL database called MongoDB. Unlike relational databases, NoSQL databases do not require users to define a structure for storing the data before it can be entered, and since it was not clear at the start of the project which kind of data would be logged, it was considered better to use a NoSQL database such as MongoDB. By using MongoDB instead of a traditional SQL relational database, less time was therefore required for database configuration and instead focus could be put on logging and analysing usage patterns. NoSQL databases also has the ability to scale horizontally[11] which also is an advantage when dealing with large datasets.

When the logging component has been integrated with a target application, each time a new user uses the target application the logging component looks up whether the user already exists in the database. If that is not the case, a new user is created. User IDs are created automatically by the logging component through hyphenating the user's account name and computer name. This results in the user of the target application never having to specify a username or ID.

The logging component provides an interface for logging information and timestamps for the following:

- Method Calls
- Handled Exceptions
- Unhandled Exceptions
- Mouse Clicks
- Mouse Scroll (start and stop)
- Button Clicks
- Specified Keyboard Shortcuts

The logging component logs method calls in order to track the data flow of an application. The method logs contain data about the namespace, class name, method name, parameter types, parameter names and the time of execution. This can allow developers to find any given method in their source code, and the associated timestamp makes it possible to view the sequential order of execution.

Logging all method calls would not have been feasible without the weaving component, which injects all the method calls during compilation. After the method calls have been logged, it becomes possible for the GUI component to calculate statistics about application flows and features. The GUI component is described in section 6.1.3.

Logging exceptions is also an important aspect of the logging component. An interface is provided for logging both handled and unhandled exceptions. To log the handled exceptions it is necessary to insert a log statement into every "catch"-block in the application, which is automatically done by the weaving component. To log unhandled exceptions, an event handler in C# WPF is used. All that is necessary is to provide a method that the WPF framework calls when an unhandled exception occurs. The exception logs contain data about the exception type, stacktrace, timestamp, message, and data about inner exceptions.

To capture the user interaction, screenshots are taken for mouse clicks, mouse scroll, button clicks and specified keyboard shortcuts. All screenshots contain a timestamp which makes it possible to follow the interaction between user and computer in a sequential order. All of the logs, except for the exception logs, are accompanied by a screenshot. The purpose of the screenshots is to allow developers to view what actions a user has taken. By taking a screenshot for the different ways the user interacts with an application it is e.g. possible to view what the user did before an exception occurred or how the user used a certain feature.

The logging component was released as a NuGet package and is meant to be installed using the NuGet package manager in Visual Studio. It can be installed using other means, but as a NuGet it becomes fast and simple for developers to add it to their projects. However, this only adds the logging component to their project and nothing more. By just adding the logging component it would still be necessary to manually add all the log statements into the source code of a target application. To be able to use automated integration, the weaving component also has to be added.

One of the goals with the logging component was to make it possible to seamlessly integrate it with an application. However, it is not realistic to assume that there will never be a need for configuring the logging component. For this reason, the logging component is released with a set of default settings which can be changed to better fit each application it is used on. For example, it is possible to change how many logs should be kept in a buffer before sending them to the database, and how the logging component should behave when the connection to the database is lost.

The underlying model used by the logging component can be seen in figure 6.2. *IMongoDBObject* defines an interface for methods and properties that are necessary for a model class to implement, in order for the object to be stored in the MongoDB database. *ILog* is an interface that defines all the standard methods and properties needed by all the logs. Since all the logs should be possible to store in the database, the ILog inter-

face also requires that the IMongoDBObject is implemented. The two abstract classes, AbstractMongoDBObject and AbstractLog, are standard implementations of the their corresponding interfaces. The AbstractLog is then extended by ImageLog, MethodCallLog, SessionLog, ExceptionLog and UserMarkedLog. Each of the five different logs are stored in a separate collection in the database. ImageLog, MethodCallLog, SessionLog and ExceptionLog all contain what their names suggest. The UserMarkedLog is a log that can be created by a user of an application by using a keyboard shortcut. This enables the user to create a mark when they experience problems with the application, e.g latency issues and crashes.

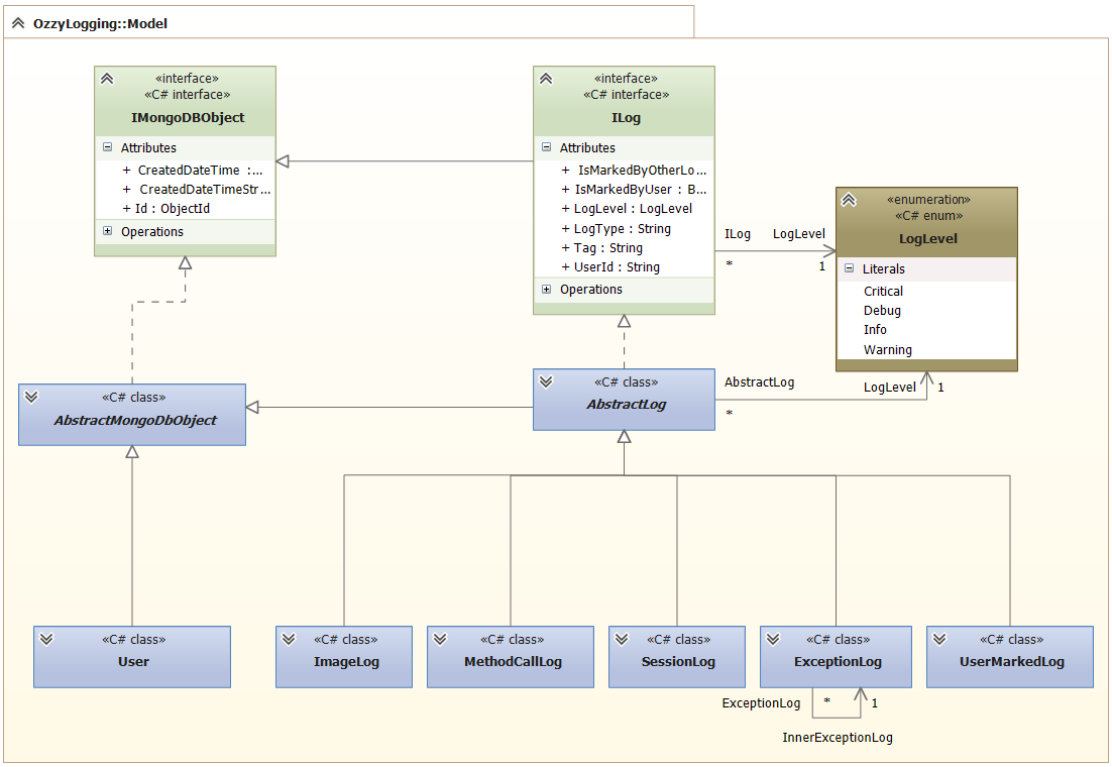

Figure 6.2: UML Diagram of the Model in the Logging Component

6.1.3 GUI Component

The GUI component was developed in C# WPF. In order to present data in a clear and understandable way, two external libraries were used. Firstly, a library called *Blacklight Toolkit* was used to create a dashboard in which different views could be displayed. Secondly, views which show charts are rendered using a library called *Modern UI (Metro) Charts*. Both libraries are open source projects released under the Microsoft Public License.

The GUI component displays data for a selected individual user or aggregated data for all users. A menu of tabs at the top of the GUI component is provided to change from viewing data for one user or all users, as shown in figure 6.3.

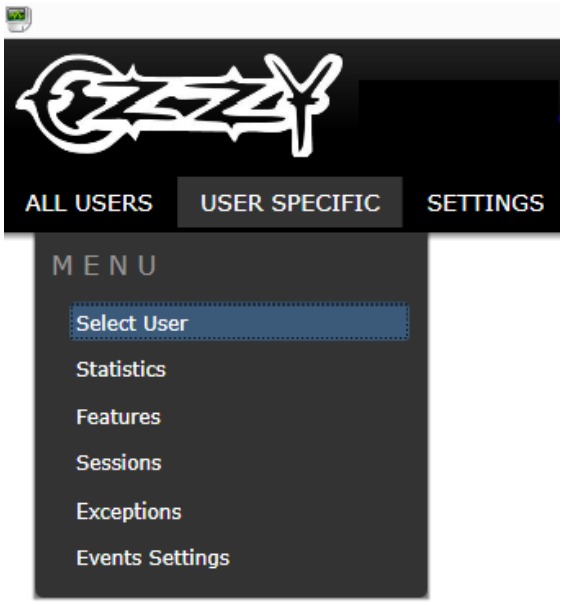

Figure 6.3: The menu for changing between viewing information for one user or all users

The *select user* sub-menu consists of a list of all users in the database along with a timestamp showing when that user was created. When a particular user has been selected, it is possible to navigate between the sub-menus of the *user specific* tab shown in figure 6.3.

If the *exceptions* sub-menu is selected, all the exceptions which have occurred in the target application while the user has used it are shown. Unhandled exceptions are shown separately from handled exceptions, and selecting a particular exception shows a tree of its inner exceptions. When an exception has been selected, a sequence of screenshots is shown which documents the actions taken by the user from 40 seconds before the exception occurred to 10 seconds afterwards. Associated with each screenshot is a list of method calls which were performed after the user action shown in the screenshot.

If the *sessions* sub-menu is selected, a window opens which makes it possible to select one of many sessions, each of which corresponds to the unique times when the selected user has used the application. For each session it is possible to see screenshots taken at strategic points which reveal user actions, as well as a list of method calls executed by the application after the action revealed by the screenshots. This is illustrated in figure 6.4.

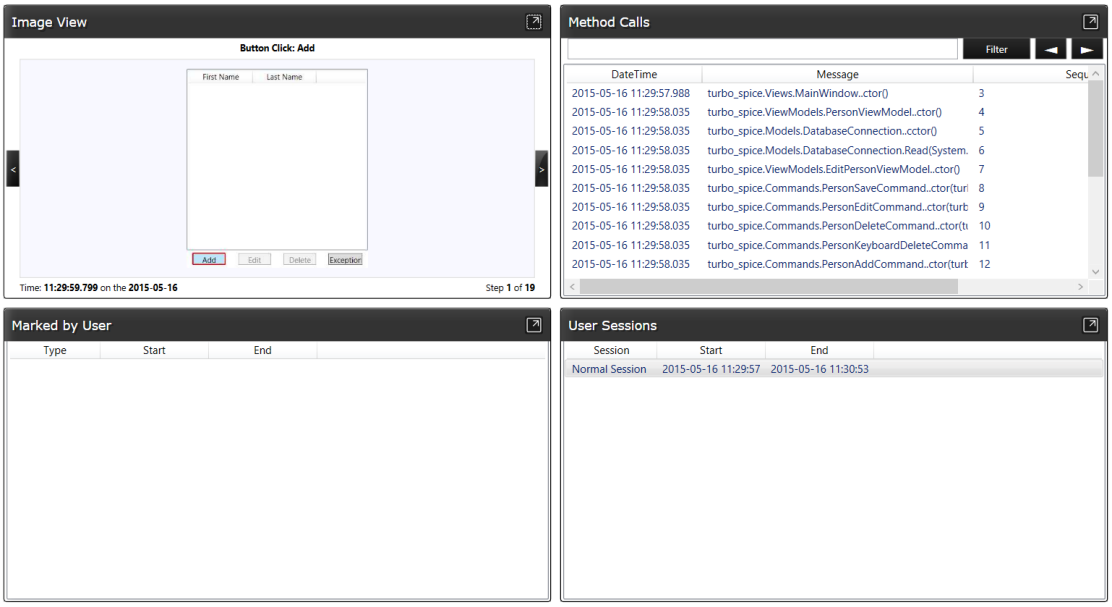

Figure 6.4: In the top left corner, a screenshot of a user clicking the Add button in shown. To the right, the associated method calls are shown.

The *statistics* and *features* sub-menus are available for both one selected user as well as in the form of aggregated data from all users under the *all users* tab. A figure of the statistics view for one users is displayed in figure 6.5. The statistics sub-menu contains information about the following:

- Most common exceptions
- Most used features
- Most called methods
- At what time of day the application is used

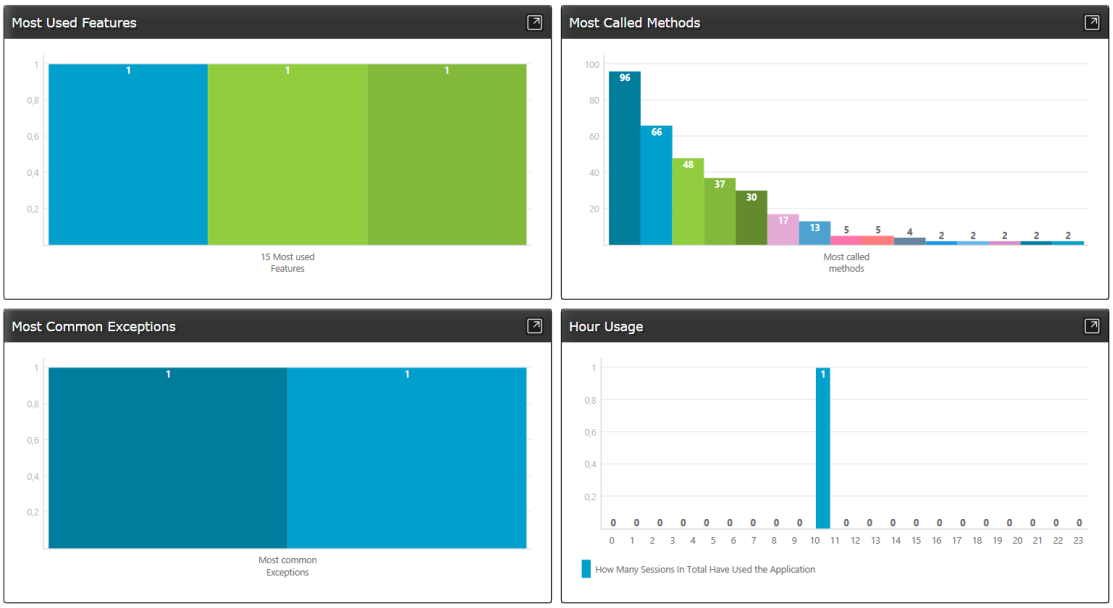

Figure 6.5: Statistics for most used features, method calls and most common exceptions are displayed. To see, for example, the name of a particular feature, the mouse can be used to hover over the bar in the chart to view the name. In the bottom right corner there is a view for displaying at what time of day the application has been used.

General statistics about an application are only shown under the *all users* tab. These include the following:

- Total number of users
- Average number of sessions per user
- Average sessions which crashed the application per user
- Total number of sessions which crashed the application
- Total number of sessions
- Average time for a session
- Average number of different flows used per session
- Average number of features used per session

Feature Definition

In order to be able to understand and analyze the relationship between the users of an application and its features, the application features need to be defined in some way. This is done through a *Settings* tab in the GUI component, in which functionality for defining and mapping what a feature consists of is provided. How the mapping process functions is illustrated in figure 6.6.

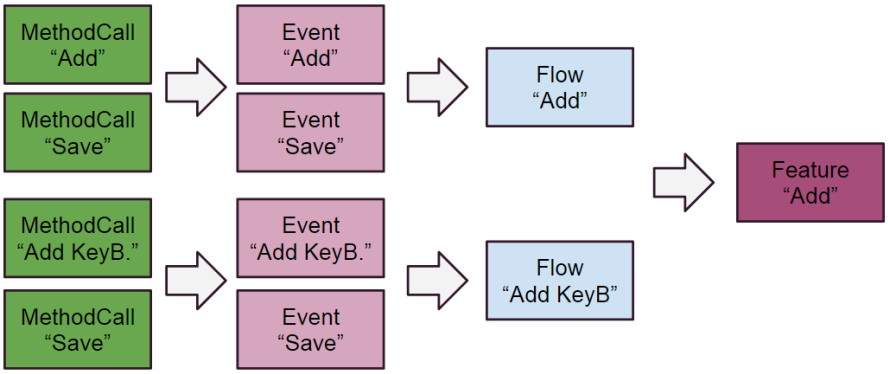

Figure 6.6: Showing the different steps in the mapping of a feature.

To give a concrete example of how this works in practice, an application that allows its users to save person names to a database and then read from that database can be considered. This application might enable its users to add a person to the database by pressing an *add* button, then entering the first and last names in respective input boxes, and finally clicking on a *save* button to transfer the information to the database. In this case, the *add* feature would not be used until the *save* button is pressed. Clicking these buttons will trigger methods to be called in the application. Since all method calls are logged by default, and each feature is mapped using these method calls, it is possible to keep the feature definition completely detached from the source code of the application that is being logged. This also makes it possible to re-define features at will, without affecting the underlying data. However, every time features are re-defined, the whole database needs to be read and data relating to the features re-calculated.

To module this, a sequence of method calls is mapped to a feature, e.g *add* and *save*. Then, for every feature, there can be several flows that will lead to the same feature being used. For example, a person could be added by using the *add* button or perhaps by using a keyboard shortcut. Each flow in itself is defined by one or several events. An event is a method call that has been defined by developers as *important* using the GUI component. An example of an event would be the method call triggered when clicking the *add* button. When executing the feature calculation, all method calls are mapped to the defined events. In case there is a defined event for a method call, the method call will be marked as an event. After all the events have been found, they are iteratively

mapped towards flows. If the events appear in a sequence, as defined by a flow, the flow is marked as having been used once — together with the feature that is represented by the flow. In this way it is possible to do calculate statistics about both feature usage, as well as the flows that constitute a feature.

After features have been defined in the GUI component, and the information on feature usage calculated, calculated data is then presented in the GUI component under the *feature* sub-menu. The feature view for all users is displayed in figure 6.7. For every feature, it is possible to view statistics for the average execution time, number of times it was used, number of session it was used in, how many session have used it, and the percentage of users that have used it. Information about the flows for the feature is available and provides statistics for how many times each flow have been used, how many users have used the flow and what is the average time for each flow. A similar view is available if a user is selected, but then no information is displayed about the relationship between the feature and how many users have used it.

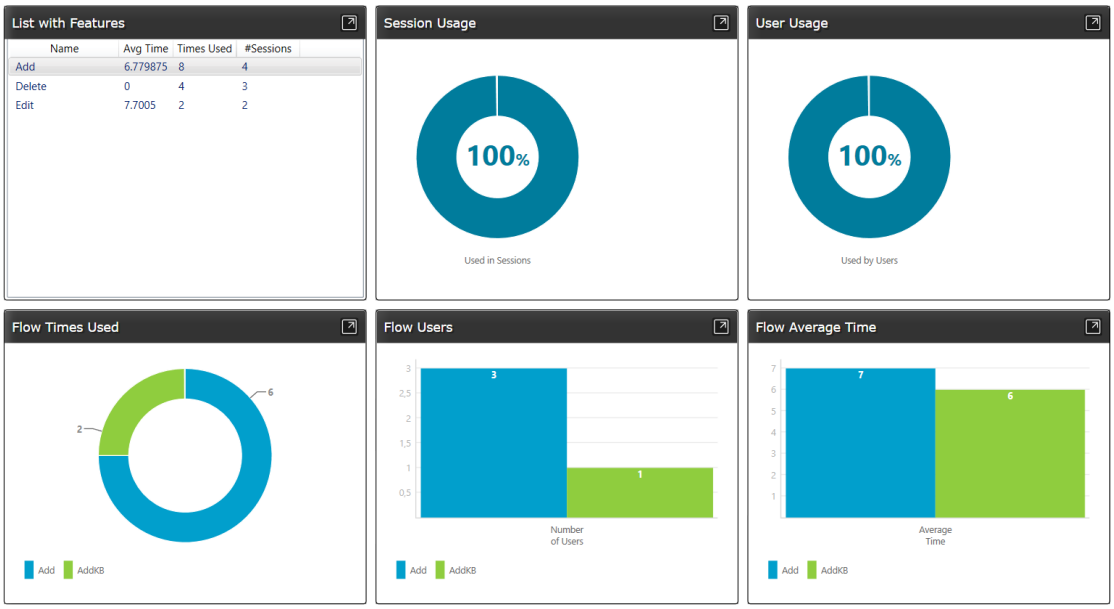

Figure 6.7: Statistics for features is displayed in this view. The bottom row shows information about the flows for the feature

Further screenshots of the GUI component as well as explanations of its various views and displays are shown in appendix A

6.2 Quantitative metrics

To evaluate the logging component several quantitative metrics were collected. The purpose was to investigate whether the integration of the logging component would lead to any large performance issues for the hosting application and to get an idea of how much data would be generated. The performance was assessed by conducting a test that measured execution time and collected information about the CPU usage of the logging component. Three different application were used during the measurements: PersonDatabase, ScreenToGif and Application X.

ScreenToGif revealed a performance limitation in the logging component by having a method that was called for every pixel of several images which the application used to generate a Gif image. This led to many hundred of thousands of method calls being executed in a short time frame. The result was not just a noticeable performance impact for the user, but also that more logs were added to the buffer than the logging component could transfer to the database. If nothing was done, this would eventually cause the logged application to run out of memory. To avoid this, the attribute `NoMethodCallLog` was used on the method to avoid logging it.

Execution time

The average time it took to run a pre-defined sequence of steps in ScreenToGif, PersonDatabase and Application X with and without the logging component integrated is summarized in table 6.1. The sequence of steps was executed 100 times using a GUI testing tool named SikuliX and the results were then averaged. The data gathered was then tested by using a t-test with α=0.05 to see if there were any significant difference between the mean time with and without the logging component. The complete data can be found in appendix D. For PersonDatabase and ScreenToGif there was a significant difference between the mean times, but not for Application X. That means that for PersonDatabase and ScreenToGif the difference observed is likely caused by the integration of the logging component, but for Application X it could as well be caused by something else. The data collected for Application X had a much higher variance than the data collected for the other two applications, which is the reason to why no significant difference could be confirmed.

	PersonDatabase	ScreenToGif	Application X
With the logging component	43.88 s	18.23 s	56.36 s
Without the logging component	42.64 s	17.44 s	55.87 s
Difference	1.24 s	0.79 s	0.49 s

Table 6.1: Average time to execute a pre-defined sequence of steps in seconds

CPU usage

Measurements of CPU usage were gathered using a tool in Visual Studio. PersonDatabase and ScreenToGif were executed together with the logging component while using the CPU measurement tool. The result is presented in table 6.2. To give a hint about performance bottlenecks in the logging component the CPU usage of the logging component was divided into three areas: MethodCall, Screen events and buffer & DB (DataBase). The logging component constituted 25.93% of the CPU usage of PersonDatabase and 11.91% of the CPU usage of ScreenToGif. The reason for the large difference between the CPU usage in PersonDatabase compared with ScreenToGif is that ScreenToGif requires more CPU computation in general just to run the application, for example when encoding GIF images. PersonDatabase is a small application with a relative low CPU usage. For this reason the CPU usage for logging method calls, gathering screen events and manage buffers becomes large relative to the CPU usage of the application.

	MethodCall	Screen Events	Buffer & DB	Total
PersonDatabase	0.55 %	14.01 %	11.37 %	25.93 %
ScreenToGif	0.2 %	4.45 %	7.26 %	11.91 %

Table 6.2: Percentage of how much CPU the logging component is using relative to the total CPU usage of each Application.

Database size

To measure the size of the data generated by the logging component, a built in measurement tool in MongoDB was used. The size of an individual image and of a log statement for a method call was calculated from the data. The results are presented in table 6.3 and table 6.4. *Count* defines how many logs the database contained and *Avg. Object Size* is calculated by dividing the *total size* with *count*. The results are further discussed in section 7.2.

Database Image Collection Size

Approx. Image Size	Count	Total Size	Avg. Object Size
Small (300x350)	560	7200 kB	12.86 kB
Medium (880x600)	200	12907 kB	64.5 kB
Large (1550x840)	206	26416 kB	128.23 kB

Table 6.3: Database size for storing images using the logging component

Database MethodCall Collection Size

Application	Count	Total Size	Avg. Object Size
PersonDatabase	17452	8656 kB	0.496 kB
ScreenToGif	6475	3217 kB	0,496 kB

Table 6.4: Database size for storing method calls using the logging component

6.3 Interviews

In order to evaluate the research question of whether an external logging component could be used to aid in the process of software development by providing developers with information about usage patterns, two workshops were held to obtain qualitative data. The workshops consisted of semi-structured group interviews, as described in section 5.2. The complete list of questions asked in the interview can be found in appendix B.

6.3.1 ScreenToGif

The participants in the first workshop were four students in the Software Engineering M.Sc. programme at Chalmers University of Technology. All of them considered software development to be their area of work and they all had previous industrial experience which ranged from 2 to 7 years. Three out of four participants had previous experience with Visual Studio, in addition to having used external logging components, such as Log4Net and log4j. Two participants had furthermore used custom made logging components in order to trace exceptions, and one participant had used a logging component which utilized aspect oriented programming by using PostSharp.

The workshop began by downloading the *ScreenToGif* application source code from SourceForge and importing it as a solution in Visual Studio. The NuGet package manager was then used to integrate the logging component with the solution. Before the

application was started, it was necessary to write `[NoMethodCallLog]` immediately preceding a method called `getPixel()` which returned the color of every pixel in a recorded video. Without this minor modification of the application source code, the method would be logged a disproportionate number of times compared to other methods, resulting in skewed statistics and a poorer performance of the logging component. The ScreenToGif application was thereafter started and a sequence of operations performed before the application was then closed.

The participants all thought that the integration process was easy and straightforward. One participant wondered whether modifications generally needed to be made, such as stating that the `getPixel()` method should not be logged. A discussion then ensued on the difficulty of building software which can be generalized to work on as many platforms as possible, as it is often difficult to predict corner cases. One method being called exceedingly more often than all others, as was the case with `getPixel()`, illustrates this quite well.

The participants opined that if they had access to videos showing how the integration process worked, along with good documentation about configuration options such as the `[NoMethodCallLog]` statement used in ScreenToGif, they would not have any qualms about integrating the logging component on their own without any more training. All participants stated that the integration looked easier than what they had expected prior to the start of the workshop.

After demonstration of a typical usage of the ScreenToGif application had been given, the GUI component was started to show how the logged data was visualized. A discussion on a range of topics then ensued. The participants e.g. wondered about performance issues, and especially how well the logging component would perform if used on an application with thousands of users. For most normal desktop applications, the logging component does not appear to affect performance to any noticeable degree. However, under some conditions this may not be the case. The previously mentioned `getPixel()` method was called so often for instance that it had a noticeable impact on performance. Generally speaking, applications with loops which call an methods an enormous amount of times are most likely to be problematic. Further work on analyzing performance on these kinds of applications would be needed in order to rectify the problem. Performance could also be affected if the logging component attempts to save too large amounts of data in the remote database for the bandwith of the internet connection to handle. The database additionally needs to have enough storage space if e.g. data for thousands of users is logged. Performance issues are further discussed in section 7.2.

The participants were adamant that the users of an application should have the right to know that their actions are being logged. One idea which was suggested to accomplish this was to inject a pop-up to the application the logging component is integrated with, which asks the users on the first application start whether they wish to send anonymous usage statistics to the developers or not.

The shared opinion of the participants was that they thought most users would be more comfortable knowing they were being logged at work but not in applications they use at home. They also made the distinction that the logging component only logs what users do in one application — not everything they do on their computer. Most users would therefore be more likely to consent to logging on applications which do not reveal much personal information about them. As an example of applications which the participants mentioned they had no problems with being logged using were project management software tools, drawing applications, spreadsheets. The main cases where they did not want to be logged were their Internet browser and applications where they did not know that they were being logged.

One participant mentioned that it might be possible to do less testing if the application was used. In his workplace, applications were often released to beta testers before being shipped, and by using the logging component at that stage to catch exceptions which occur it might be possible to cover more ground than were otherwise possible and spend more time in the beta testing phase instead of testing.

All participants thought that their companies would be interested in using an external logging component of the type that was developed. They all also opined that the logging component would be useful for them to understand how an application was truly used by its users and that such information would help to move development in the right direction. The participants were also of the opinion that the logging component would be beneficial for debugging an application. Three participants thought that the debugging functionality was more useful than seeing user flows, and one thought it was the other way around. One participant opined that the question was not relevant as the logging component was useful for both purposes and that being good at one thing did not exclude being good at another.

6.3.2 Application X

The second workshop was held at Diadrom and the participants were four developers with years of experience in developing, debugging and maintaining software applications. All participants had previous experience with Visual Studio and all had used logging tools of some kind at some point in their career. The second workshop followed the same structure as the first: the background of the project and the research question were presented, the integration of the logging component and the target application was shown, a live demonstration of of the application was given and the logged data was then shown. Several open-ended questions concerning various aspects of the logging component were asked throughout the process, but the conversation also diverged from the script several times. The final part of the workshop was used to ask the workshop participants what their general impressions of the logging component were and ask them whether they had anything to add.

The target application which the logging component was integrated with in the workshop was built for a Swedish aerospace company. For reasons of confidentiality, neither the application nor the company it is used at can be discussed further.

On the whole, the participants were impressed with the logging component. Their comments on the integration process were that it looked uncomplicated and clear-cut, which could greatly aid its adoption in industry. The comments on the data which the GUI component visualizes were that it could be helpful, especially the screenshots and information of steps users take before exceptions occur.

The participants however wondered how the logging component would work in applications whose source code is obfuscated at compile time. They also wondered whether the data generated was too much in cases where thousands of users use an application. They also had many concrete suggestions for how the logging component could further be developed. One participant said that it would be useful to add a view to the GUI component which enabled developers to select which namespaces, classes or methods should be logged and which should not. This would enable developers to log only relevant information for a given purpose, e.g. the debugging of a particular class in a program. Another participant mentioned that it might be useful for the logging component to present information about the state of the computer using an application, such as memory usage or CPU time. Other suggestions which were discussed include:

1. Splitting the logging component into different parts: one focused at developers and another one focused on managers.
2. Generating statistics to present which features lead to the most exceptions.
3. Being able to select a particular feature and see statistics for what users did before and after they used it.
4. Comparing different groups of users, e.g. seeing whether there is a difference between how people in Sweden and Denmark use an application.
5. Seeing what type of features are used at what hours of the day, which could help to e.g. determine when a system should undergo maintenance.
6. Presenting heatmaps of user clicks

Incorporating all these suggestions would take a great deal of time and would be out of scope for the thesis. The suggestions however illustrate the myriads of possibilities in which the data the logging component generates can be analyzed and presented. They also illustrate that even more types of data can be logged than what is currently done.

Finally, the participants said that they would not mind using an application which was logged by the logging component — as long as the data was not used to try to measure the

productivity or performance of an employee. They did not want the logging component to be use as a tool which upper management could use to monitor the employees of a company. As long as the logging component was used for debugging or developmental purposes, they opined that it would be a great tool for developers.

6.4 Survey

During the course of the research, the logging component was presented at four different companies, using the *PersonDatabase* application as an example target application with which the logging component was integrated. After each presentation, the survey in appendix C was handed out to those who were present. The total number of participants in the survey was 27 and the results are displayed in figures 6.8 to 6.17.

1. My area of work is:

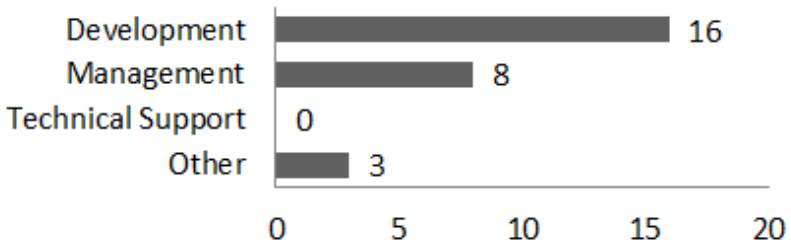

Figure 6.8: The participants' area of work

2. I have this many years of work experience:

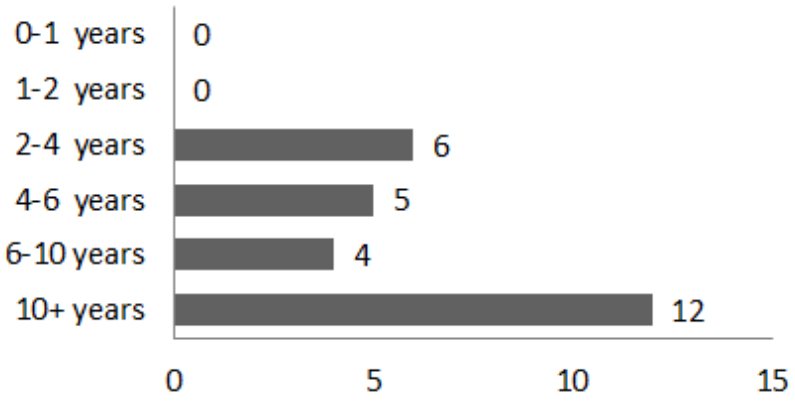

Figure 6.9: The participants' work experience

3. I would be comfortable if an application that I use for private matters is being logged by the logging component

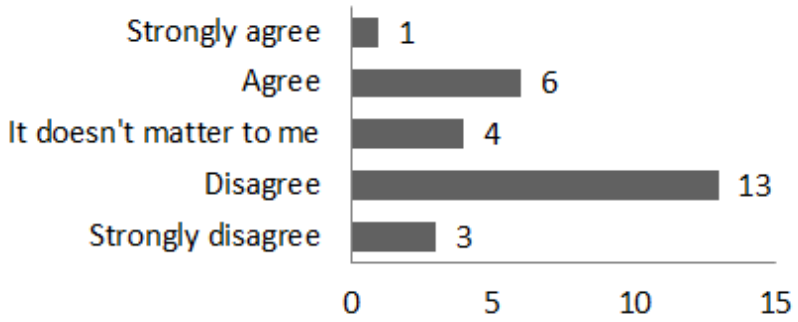

Figure 6.10: The participants' level of comfort with logging an application they use privately

4. I would be comfortable if an application that I use at work is being logged by the logging component

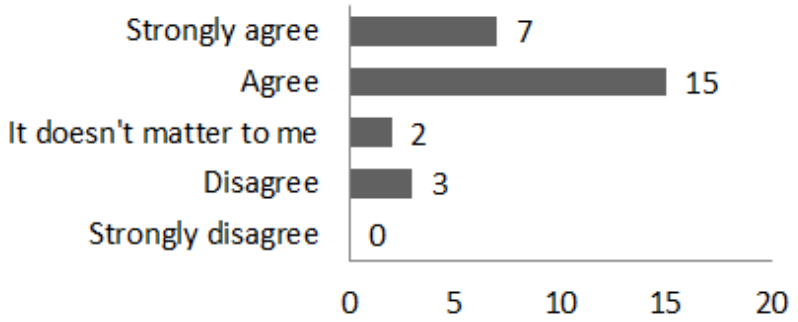

Figure 6.11: The participants' level of comfort with logging an application they use at work

5. I would be more comfortable using an application that is being logged at work rather than one I use for private matters

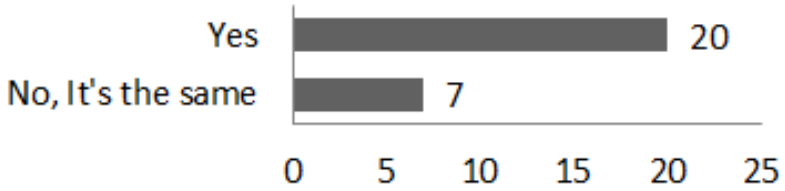

Figure 6.12: Comparison of level of comfort between applications used at home and at work

6. I know one or several projects that I have been part of where the logging component would have been useful

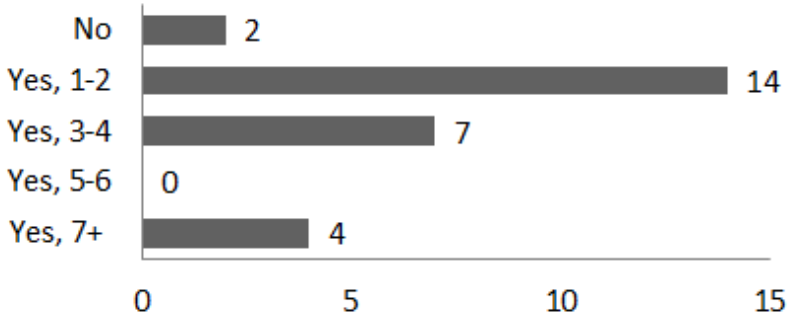

Figure 6.13: The number of projects the participants know of where the logging component would have been useful

7. I believe the logging component has potential to provide information that will facilitate the *debugging* of an application

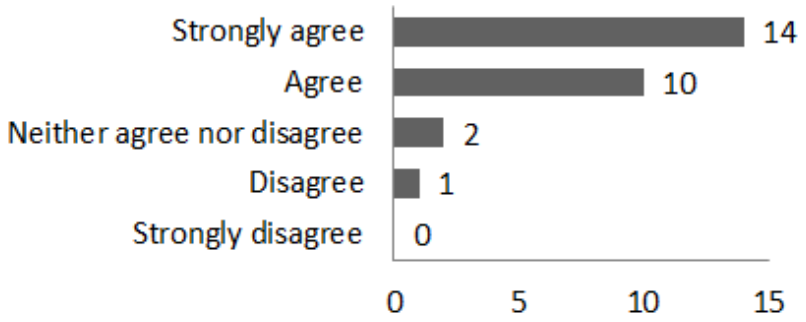

Figure 6.14: The potential of the logging component to help the debugging process

8. I believe the logging component has potential to provide information that could aid in *further development* of an application

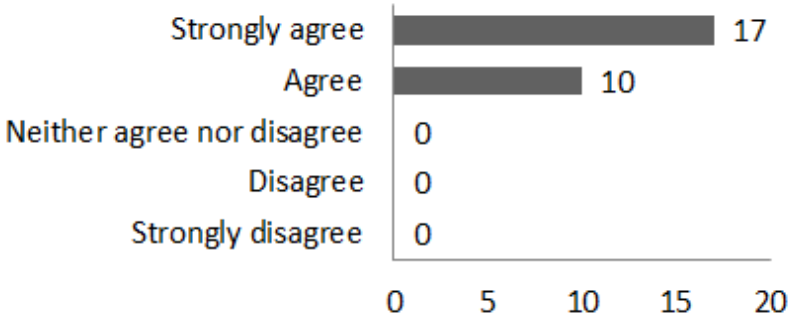

Figure 6.15: The potential of the logging component to help the development process

9. Of all the information provided by the logging component, the most important for me is:

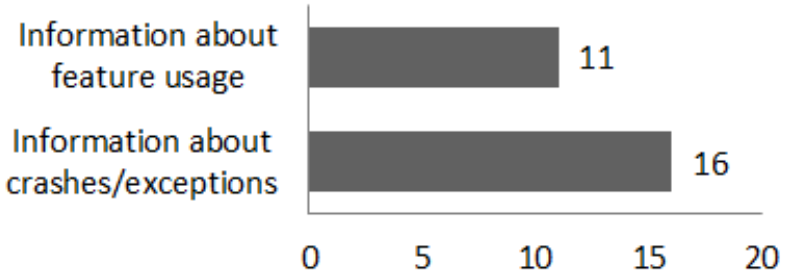

Figure 6.16: The most important type of information gained from the logging component

10. I believe my company would be interesting in investing money to get functionality similar to the one provided by the logging component

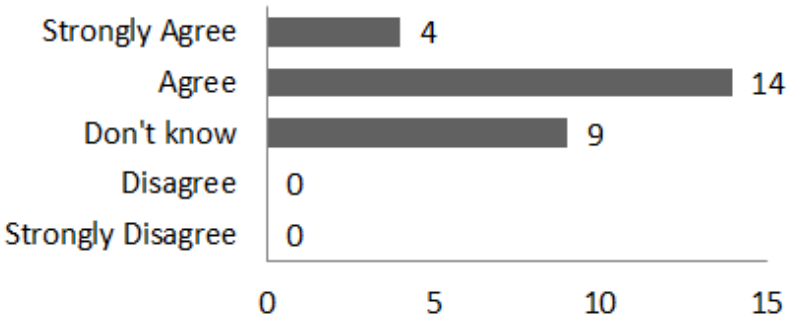

Figure 6.17: The participant's belief on whether their companies would be willing to pay for the functionality provided by the logging component

6.5 Analysis

The research question which was investigated during the course of the study was *how can an external logging component be used to aid in the process of software development by providing developers with information about usage patterns.* The data from the workshops and the survey supported the hypothesis that the logging component could aid in the process of software development. All participants in the workshops and over 90% of the respondents in the survey said that they knew of one or more projects they had worked on where the logging component would have been useful. The semi-structured interviews revealed two main ways in which the logging component could help:

1. By giving information about exceptions and crashes, to help developers to debug an application
2. By giving information about feature usage, to help developers to decide which features are important and which ones should not be developed further

Opinions were split on which one of these was more important. Around 60% of the survey respondents said that information about exceptions and crashes while 40% thought information about feature usage was more important. There was a statistically significant difference between the opinions of developers and managers, as shown in figure 6.18. Around 70% of the developers considered exception information to be more important than feature usage information, compared to under 40% of managers.

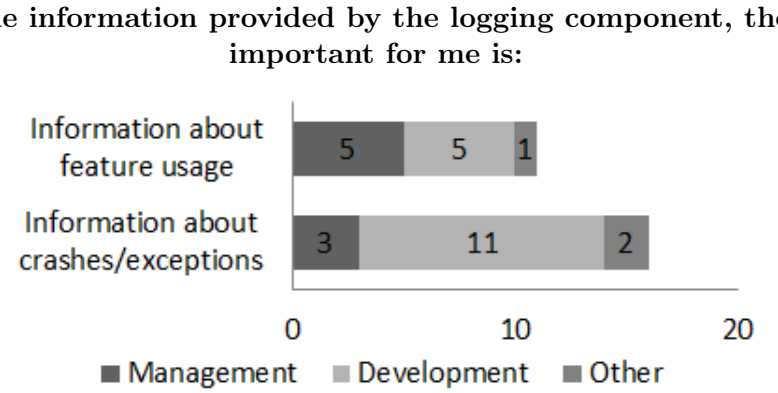

Figure 6.18: Most important information by profession

However, there seemed to be a correlation in the other direction between years of experience and preference between the aforementioned aspects, as can be seen in figure 6.19

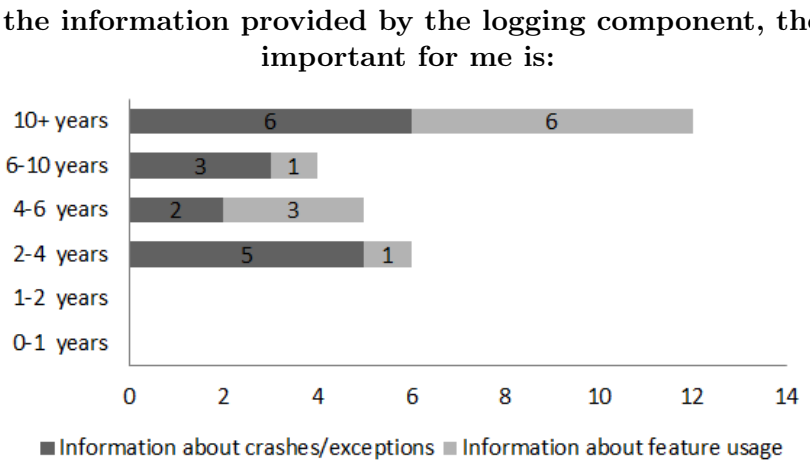

Figure 6.19: Most important information by experience

This was misleading however, since a disproportionate number of managers surveyed had high levels of experience, as shown in figure 6.20. Over 70% of managers had 10+ years of experience compared to under 40% for developers. After correcting for this, the correlation did not appear to hold any more. Since the focus of the research question in this study was on the developer point of view, the data suggests that information about exceptions and crashes is more important than information about feature usage.

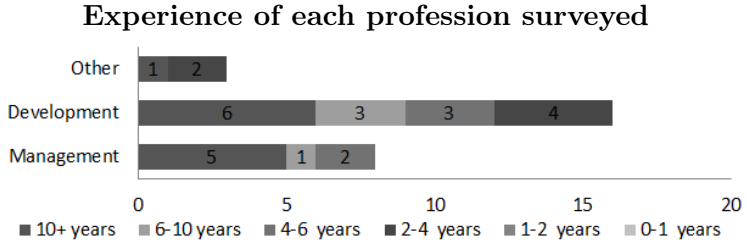

Figure 6.20: Work vs experience

However, whether the logging component is more useful at one of these two aspects does not exclude it being good at the other. The participants of the workshops were all of the opinion that the logging component could aid in developing an application through both of the aforementioned aspects. The survey yielded the same results, with 100% of respondents agreeing that the logging component has potential to provide information that could aid in further development of an application, and around 90% agreeing that it has potential to provide information that would facilitate the debugging of an application.

The conclusions drawn from the data therefore suggest that an external logging component can be used to aid in the process of software development by providing developers

with information about usage patterns by providing information about:

- Exceptions and crashes
- Feature usage

The performance measurements conducted in section 6.2 indicates that the logging component have some negative effect on the performance of an application. The performance impact of the logging component will always vary depending on the application, but by using the tools provided by the logging component it is possible for the developer to fine tune the logging to avoid cases where the logging component will decrease performance to a degree where it is noticeable by the users. The observed time difference for PersonDatabase would be almost impossible for a users to notice and after removing the logging from one method in ScreenToGif it would either not be possible for a user to notice a difference. Further discussion about performance issues are found in section 7.2.

The final question is then whether an external logging component of the sort which was built during this research is useful enough to warrant further development and analysis. In order to answer that question, it is instructive to view the results from figure 6.17. The most important test of whether an application is truly useful or not is often whether people are willing to pay for it. In this respect, the logging component also performed favorably, with around 65% of respondents believing that their company would be interesting in investing money to get functionality similar to the one provided by the logging component.

7

Discussion

The logging component which was built during the course of the research was designed to be generalizable and easy to implement. The semi-structured interviews that were conducted in two evaluation workshops fully supported the hypothesis that this objective had been achieved. In both workshops the total time to integrate the logging component took under five minutes and the participants said that the integration was *easy and straightforward* and that it looked uncomplicated. Such a swift and effortless integration is in most respects a positive feature, but it also has a potential drawback: the possibility of use as spyware for malicious purposes. This potential drawback, along with a host of other issues, is further discussed in this chapter.

7.1 Design decisions

Throughout the project development, decisions were taken which impacted the final version of the logging component in various ways. For instance, as the system was designed with the focus of being easy to integrate, some trade-offs regarding performance were unavoidable. How the *easy to integrate* design principle affected the final outcome is particularly well illustrated by two issues which came up.

1. The logging component logs all methods calls. This will inevitably decrease the performance of an application, especially if it calls methods unusually frequently. Another way to implement the logging would have been to log only those methods which have been specified by developers and nothing else. That would however have meant that the developers would need to spend time deciding which methods should be logged and marking them as such. To avoid that, every method call

is logged and a possibility of specifying which methods should not be logged is provided instead.

2. The way features of an application are defined is done in the GUI component. This means that when a feature is defined or re-defined, the whole database needs to be read through, data aggregated and results calculated. It would also have been possible to define features in the source code of the logged application instead, which would avoided spending as much time on feature calculation. However, this would have meant putting a great deal of calls to the logging component in the source code, making it less readable, and it would mean re-defining features after an application has shipped would not be possible.

Another important design decision was how the logged data should be visually presented. This was done by building a GUI component using C# WPF and employing two external libraries to complete the GUI. The data could just as well have been presented through other libraries or methods, or even written in a totally different framework, such as e.g. as a web application. If the logging component were extended to cover applications written in more programming languages than just C#, this would likely be desirable. If the logging component were however used by an organizations which have some internal data visualizing tools or an organization-wide intranet, the data visualization would likely fit better if added to these previously existing tools.

Just as some of the design decisions which were taken in order to develop the logging component affected the final result, so will the decision of other developers to use an external logging component affect their software applications. As the logging component injects functionality into applications at compile time, the final code released by developers would not be exactly the same code as what they wrote. This is no different from any other external library of framework however; using external libraries or products always represents some loss of control.

7.2 Performance issues

One of the requirements of the logging component was that it should not reduce the performance of an application to a degree where it affects the usage of an application. Whether this requirement has been fulfilled or not is a question still open for discussion. An application that has a method that is called thousands of times during a short interval will create latency issues and might even cause the logging component to run out of memory. This was a problem that appeared during the development phase and the current solution to the problem is to remove the logging of that method by using the provided `NoMethodCallLog` attribute or by specifying the method in the settings file. One could argue that the performance should be improved instead. Doing that is certainly

possible, but it would still not solve the problem. It would only move the boundary of when the issue will arise, making the problem appear less often. One noticeable issue is that this enormous amount of method calls is rarely caused directly by a GUI event. In the *ScreenToGif* application it was caused by the encoding of an image, which is a background calculation to produce the GIF image. To understand the usage patterns of an application, knowledge about all of those method calls is therefore often not necessary. It could however be useful to know the complete call hierarchy if an exception occurred, but the intention with the logging component was not to replace a normal error log. Instead it focuses on providing information about what the user did before the exception occurred, making it easier and faster for the developer to correct the error. Removing the log statements in certain performance critical methods was therefore seen as acceptable.

Another concern is the large volumes of data that the logging component will create if it is ever used in a released application. If an application for instance has 1000 active users and each user generates 1000 images per day, it would result in:

$$1000 \times 1000 \times 128.23kB = 128230000\,kB/day = 128.230\,GB/day$$

This is calculated using the size presented in table 6.3, where the size for the large images is used. Storing 128 GB of data every day will be both expensive and result in problems managing and querying the large volumes. No time has however been spent on minimizing the image size due to time constraints for the thesis. It is therefore most likely possible to greatly reduce the size for each image. In many cases there is also no need to store all the data after a couple of weeks or months, and by developing a script that removes unnecessary data could reduce large volumes of data. But with a growing number of users and applications that require extensive GUI interaction and are used during a major part of the day, another solution will be needed. One alternative could be to implement an on/off functionality for remotely controlling which users should be logged. By doing that it is possible to only monitor a smaller part of all the users, which would make logging more manageable. The users could be randomly selected, and then by using statistics estimates could be made for the whole population, or they could be specifically selected to eventually cover the whole population of users.

7.3 Ethical considerations

There are two main ethical aspects relating to the logging component which need to be considered. Firstly, there is the issue of potential malicious use as spyware which was previously mentioned. If the logging component were to be released as open source software, it would theoretically be easy for the developers of any C# WPF application to easily bundle it together with their application in order to gain access to personal information about users, their usernames and passwords, or their bank and credit card

information. Information about computer usage habits which users do not wish to disclose could likewise be obtained. Such usage of the logging component would not only be unethical, but also illegal. Unauthorized access to a computer or user data is illegal in most jurisdictions, as e.g. defined by the *Computer Fraud and Abuse Act* in the United States and the computer fraud section in the 4th chapter of the criminal code in Sweden.

However, there is not much that can be done to prevent this. There are many logging and monitoring tools which can be used for the same nefarious purposes, and if the logging component were not released because of these concerns that would in effect be a form of *security through obscurity*. In the academic security community, security through obscurity is generally considered to be bad practice [32]. In this context it is therefore the responsibility of the developers of an application to use the logging component responsibly and to notify users if the user actions are logged.

In the event that the logging component will be formally released, either as an open-source or proprietary software, code will be written which injects a pop-up into the application which the logging component is integrated with for the first time a user starts the logged application. This will be done to underscore that the users of an application need to be aware that they are logged. The pop-up would inform a user that he or she is being logged, and give instructions on how to turn the logging on or off. User confidentiality should be paramount in cases where user interactions are logged, and the logging component was not developed for unauthorized logging. As long as users of an application are aware that they are being logged, user confidentiality is not breached.

The second issue is that the logging component could be used for productivity measurements, i.e. to measure which employees of a company are the most productive and which ones are the least productive. This was an issue raised in one workshops which were held; participants said that they would not mind using a logged application as long as the logged data was not used for productivity measurements. This is a bit of an ethical grey area. It is not clear whether any ethical limits are crossed if all employees are aware and accepting of the fact that the logged data is analyzed in this fashion. Whether it crosses ethical limits can depend on cultural issues, such as how the concept of personal integrity is reconciled with the commitment made to an organization or company. There may also be local laws to consider which can vary from jurisdiction to jurisdiction.

However, the same applies as in the issue of unauthorized logging, namely the issue of original intent. The logging component was neither built for the purposes of unauthorized logging nor for logging productivity. The onus therefore lies with those who use it to proceed in an ethical and responsible fashion. Any tool can be misused, and the logging component is no different.

7.4 Threats to validity

In order for research to be considered scientific, its conclusions must be reproducible using the defined parameters in the research [33]. There are many aspects which can influence the reproducibility, and by extension the validity, of a research. In this section the threats to the validity of this study which were identified are discussed.

The external validity of a research refers to the extent to which it is possible to generalize research findings and to what extent the findings are of interest to people, tools, organizations or companies outside of the investigated case [34]. The logging component was evaluated with a survey, quantitative measurements on three different applications of which two were real-life applications in actual use, and workshops with semi-structured interviews. In order to obtain more generalizable results it would be necessary to test the logging component for a longer period on applications and analyze how they evolve. The logging component also only functions on applications written in C# WPF. There may possibly be factors in the language which could lead to biased results which are not reproducible in other languages.

Internal validity refers to the threat that unrelated factors which are not under investigation might cause the effect observed in the study [34]. Put differently, internal validity is the risk that a third factor is the cause of an observed effect — and not the factor which was under investigation. This can be particularly dangerous if researchers are not aware of other factors which could influence an outcome. The quantitative measurements made in this research only had one factor which differed — whether the logging component was integrated into the target application or not. The semi-structured interview used in the workshops consisted of questions which related solely to the benefits of the logging component and no third factor was ever involved. For these reasons no internal threats to validity were identified in the study.

Construct validity refers to whether the values measured in a research reflect the intended ones. For example, if the questions discussed in an interview are not interpreted the same way by the interviewer and the interviewee, there is a threat to construct validity [34]. The qualitative data used for evaluation the logging component used interviews. In order to minimize the construct validity threat, the interviews were in a semi-structured format, in which the questions could be discussed and the interviewers explain the questions in cases where the interviewees were uncertain. However, a survey was used in one part of the evaluation. In this case, it was not possible to expand upon the questions which were listed. There was therefore a certain threat to validity in that case, but the questions were worded as clearly and succinctly as possible in order to minimize it.

The conclusion validity of a research refers to whether it is possible to draw the correct conclusions, i.e. whether there is a statistical relationship between the factors under investigation. If other researchers later on conduct a similar study, they should be able

to reproduce the results [34]. This study was conducted by two researchers, which minimized the *single researcher bias*. All data collected was digitalized and documented and reviewed many times. Employees at Diadrom read the results and agreed with their interpretation. Conclusion validity was also minimized by strictly separating the collection and analysis of the data which was obtained in the two different target applications in the case study.

8

Conclusions

This study examined how a logging component could be used to aid in the process of software development by providing developers with information about usage patterns. The results of the study, which are presented in chapter 6, indicate that a logging component that can enable developers to obtain information about usage patterns has developmental value. All participants in the workshops that were held and over 90% of the respondents in the survey which was posed said that they knew of one or more projects they had worked on where the logging component would have been useful. The study revealed many developmental benefits for the developing organization of an application, but the most important of these benefits were the ability to obtain information about exceptions and crashes, as well as information about which features the users of an application use. Opinions were split on which one of these two was more important, with developers and managers in particular taking a different stance. Developers generally thought that information about exceptions and crashes was the most valuable, while managers thought that information about feature usage was of greater value.

The logging component developed during the course of the study can be integrated with applications developed in the C# language which use WPF. In order to obtain more general results which are valid for a greater range of applications, it might be pertinent to port the logging component so that it could work on applications written in other languages. The Java programming language uses a compact collection of numeric codes, constants and references called *bytecode* as an IL before java code is compiled to machine code. It is possible to inject functionality to the bytecode in a similar fashion as the IL injections used in the logging component. Recommended future work therefore involves porting the logging component to Java, and possibly other languages as well.

The evaluation of the logging component focused on developers and the benefit they can attain from it. As already stated however, there are two other possible focus groups for

the component: people in IT support and managers. Recommended future work includes analyzing the benefits of the logging component from their perspective.

The logging component was evaluated through both a survey as well as a case study in which it was integrated with two different applications. While the conclusions seem to be relatively robust, further evaluation could be conducted. It would be interesting to conduct a blind experiment in which two groups of developers were given the task of debugging an application and one group would have access to the logging component but the other one wouldn't.

A more long term experiment could also be conducted in which the evolutionary path of two different applications was compared, where one application was developed over time using information about features gained from the logging component and the other one was developed over time using traditional methods. This type of an experiment would be difficult to perform however due to the considerable amount of time and resources needed.

Bibliography

[1] I. Schnabel, M. Pizka, Goal-Driven Software Development, in: Software Engineering Workshop, 2006. SEW'06. 30th Annual IEEE/NASA, IEEE, 2006, pp. 59–65.

[2] F. P. Brooks, N. S. Bullet, Essence and Accidents of Software Engineering, IEEE computer 20 (4) (1987) 10–19.

[3] C. A. Mack, Fifty Years of Moore's Law, Semiconductor Manufacturing, IEEE Transactions on 24 (2) (2011) 202–207.

[4] E. W. Dijkstra, The Humble Programmer, Commun. ACM 15 (10) (1972) 859–866, turing Award lecture.

[5] N. Wirth, A Plea for Lean Software, Computer 28 (2) (1995) 64–68.

[6] A. Hejlsberg, S. Wiltamuth, P. Golde, The C# Programming Language, Adobe Press, 2006.

[7] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, L. Réveillere, Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects, in: Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual, IEEE, 2013, pp. 303–312.

[8] A. Troelsen, Pro C# 2010 and the .NET 4 Platform, Apress, 2010.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP'97—Object-oriented programming, Springer, 1997, pp. 220–242.

[10] N. Leavitt, Will NoSQL Databases Live Up to Their Promise?, Computer 43 (2) (2010) 12–14.

[11] R. Cattell, Scalable SQL and NoSQL Data Stores, ACM SIGMOD Record 39 (4) (2011) 12–27.

[12] M. Staron, W. Meding, J. Hansson, C. Höglund, K. Niesel, V. Bergmann, Dashboards for Continuous Monitoring of Quality for Software Product Under Development.

[13] S. Few, Information Dashboard Design, O'Reilly, 2006.

[14] D. Loshin, Business Intelligence: The Savvy Manager's Guide, Newnes, 2012.

[15] W. Van Der Aalst, A. Adriansyah, A. K. A. de Medeiros, F. Arcieri, T. Baier, T. Blickle, J. C. Bose, P. van den Brand, R. Brandtjen, J. Buijs, et al., Process Mining Manifesto, in: Business process management workshops, Springer, 2012, pp. 169–194.

[16] N. Påhlsson, Aspect-Oriented Programming, Topic Report for Software Engineering (2002) 11–03.

[17] E. Borjesson, R. Feldt, Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry, in: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE, 2012, pp. 350–359.

[18] T. Ball, The Concept of Dynamic Analysis, in: Software Engineering—ESEC/FSE'99, Springer, 1999, pp. 216–234.

[19] F. Gabbay, A. Mendelson, Can Program Profiling Support Value Prediction?, in: Microarchitecture, 1997. Proceedings., Thirtieth Annual IEEE/ACM International Symposium on, IEEE, 1997, pp. 270–280.

[20] M. Staron, W. Meding, K. Palm, Release Readiness Indicator for Mature Agile and Lean Software Development Projects, in: Agile Processes in Software Engineering and Extreme Programming, Springer, 2012, pp. 93–107.

[21] M. Staron, J. Hansson, R. Feldt, W. Meding, A. Henriksson, S. Nilsson, C. Hoglund, Measuring and Visualizing Code Stability–A Case Study at Three Companies, in: Software Measurement and the 2013 Eighth International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2013 Joint Conference of the 23rd International Workshop on, IEEE, 2013, pp. 191–200.

[22] E. Alégroth, Random Visual GUI Testing: Proof of Concept, 2013.

[23] G. Liebel, E. Alégroth, R. Feldt, State-of-Practice in GUI-based System and Acceptance Testing: An Industrial Multiple-Case Study, in: Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on, IEEE, 2013, pp. 17–24.

[24] E. Backlund, M. Bolle, M. Tichy, H. H. Olsson, J. Bosch, Automated User Interaction Analysis for Workflow-based Web Portals, in: Software Business. Towards Continuous Value Delivery, Springer, 2014, pp. 148–162.

[25] V. Vaishnavi, W. Kuechler, Design Research in Information Systems.

[26] P. Runeson, M. Höst, Guidelines for Conducting and Reporting Case Study Research in Software Engineering, Empirical software engineering 14 (2) (2009) 131–164.

[27] R. H. von Alan, S. T. March, J. Park, S. Ram, Design Science in Information Systems Research, MIS quarterly 28 (1) (2004) 75–105.

[28] K. Peffers, T. Tuunanen, M. A. Rothenberger, S. Chatterjee, A Design Science Research Methodology for Information Systems Research, Journal of management information systems 24 (3) (2007) 45–77.

[29] R. Mason, Evaluation Methodologies for Computer Conferencing Applications, in: Collaborative learning through computer conferencing, Springer, 1992, pp. 105–116.

[30] J. G. Adair, The Hawthorne Effect: A Reconsideration of the Methodological Artifact., Journal of applied psychology 69 (2) (1984) 334.

[31] N. Manarin, ScreenToGif.
URL `https://screentogif.codeplex.com`

[32] J.-H. Hoepman, B. Jacobs, Increased Security Through Open Source, Communications of the ACM 50 (1) (2007) 79–83.

[33] S. M. Downing, T. M. Haladyna, Validity Threats: Overcoming Interference with Proposed Interpretations of Assessment Data, Medical Education 38 (3) (2004) 327–333.

[34] P. Runeson, M. Host, A. Rainer, B. Regnell, Case Study Research in Software Engineering: Guidelines and Examples, John Wiley & Sons, 2012.

A

Screenshots

In this section screenshots of some of the views of the GUI component are shown.

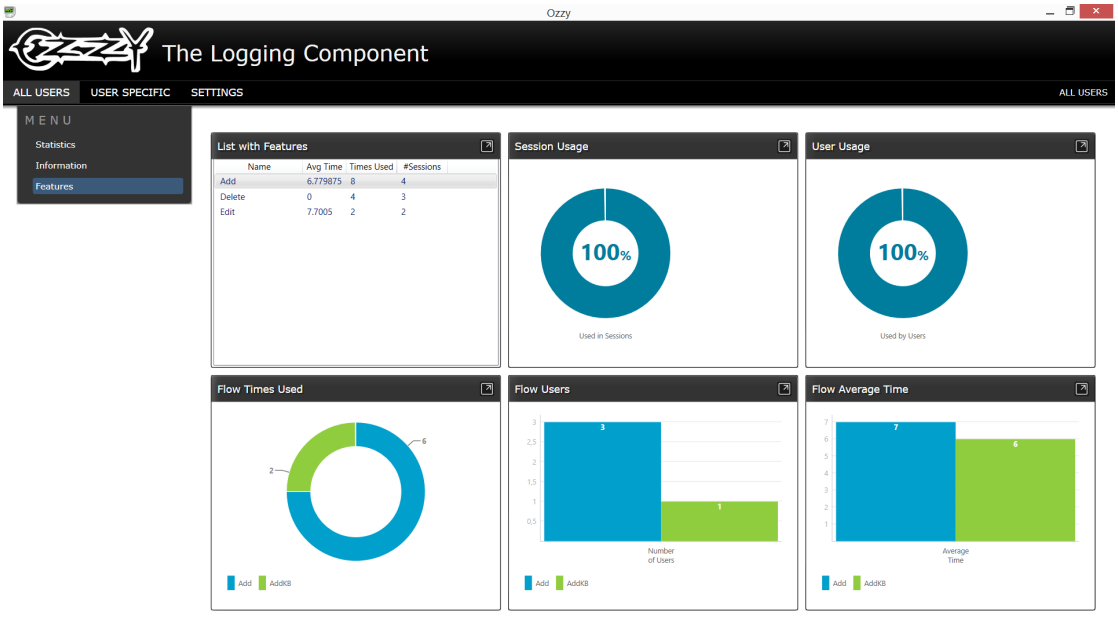

Figure A.1: The GUI component, showing data about features for all users of a logged application

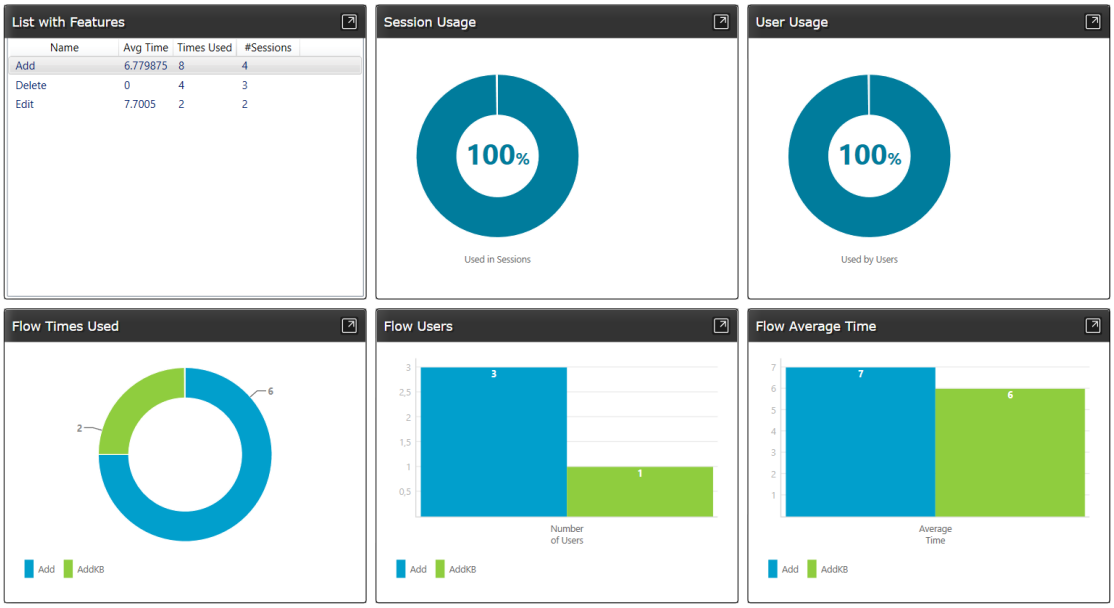

Figure A.2: The same view as in figure A.1, but with the surrounding border removed. The 3 views in the top shows statistic about a feature and the 3 bottom views show statistics for the different flows that can be used to perform a feature

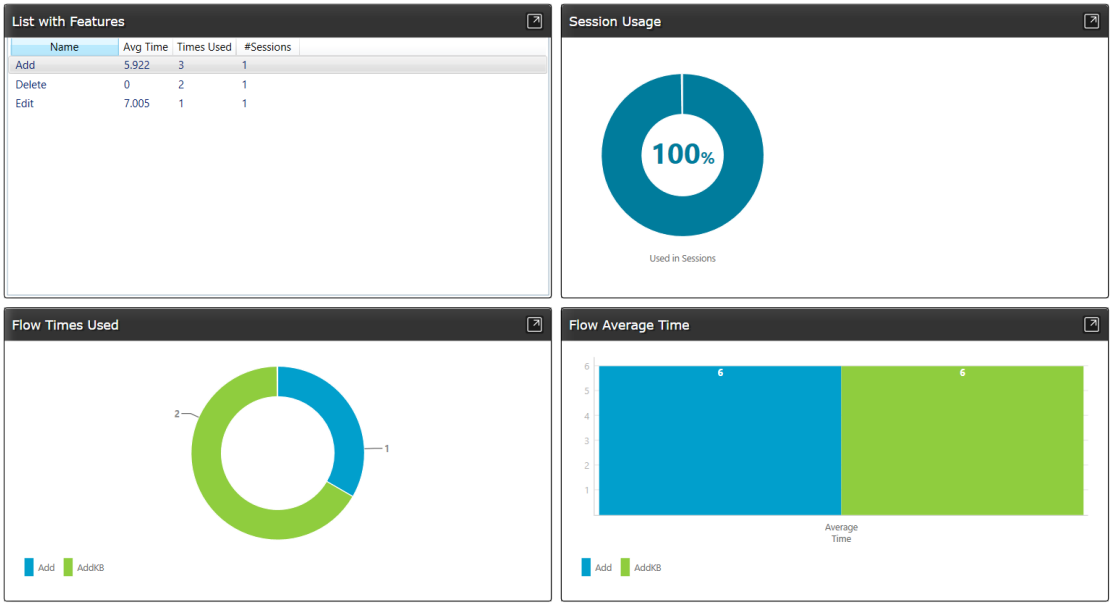

Figure A.3: The feature usage of one selected user

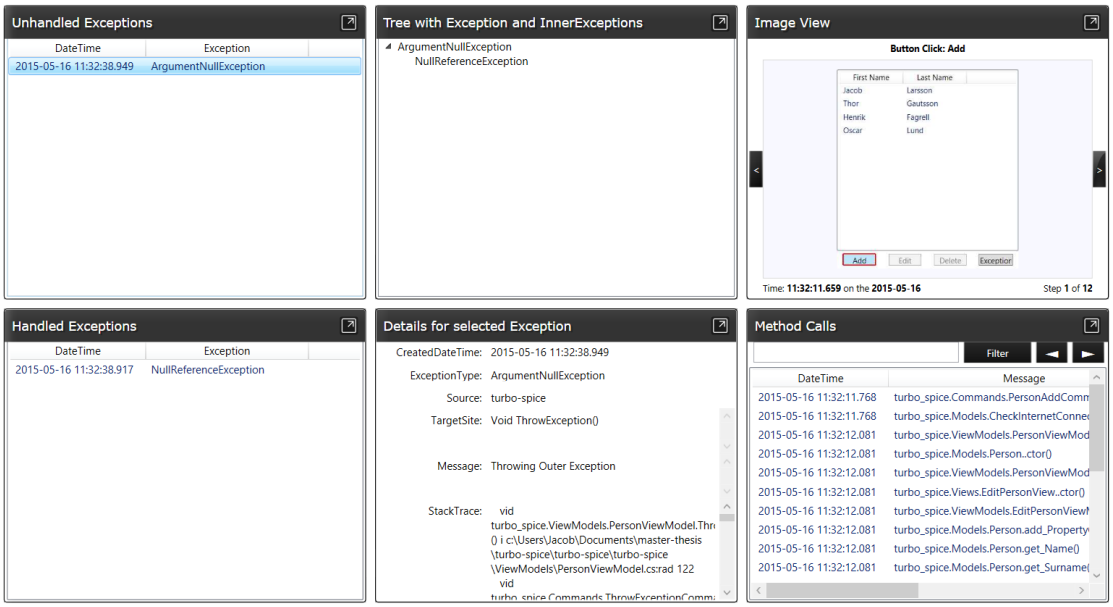

Figure A.4: A tree of exceptions, along with the exception stacktrace — as well as screenshots and method calls showing what the user did before the exception occurred

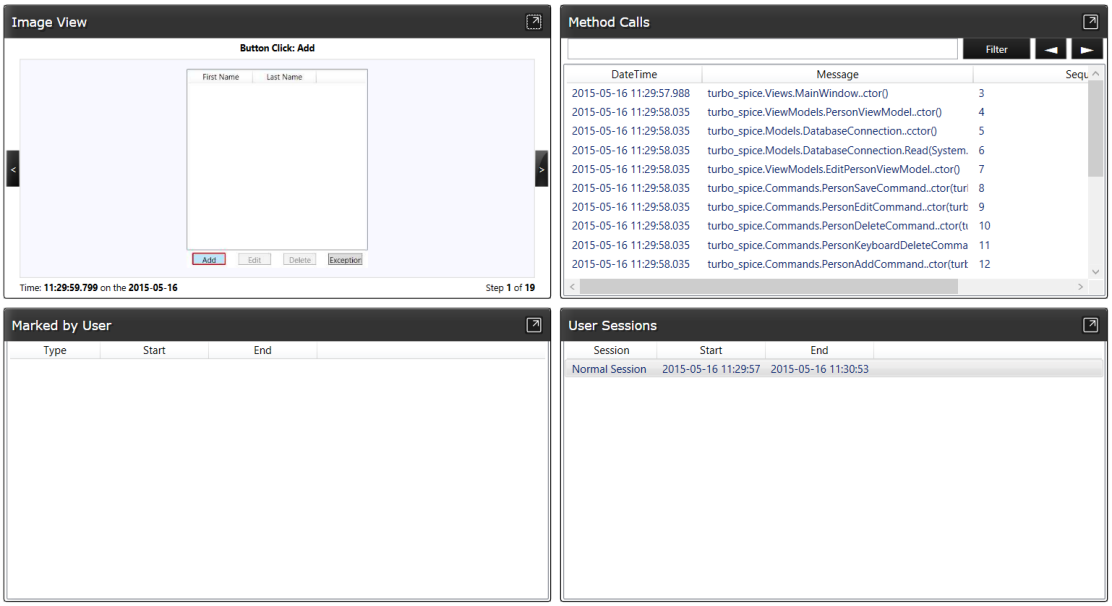

Figure A.5: Showing all the sessions for a selected user together with screenshots and method calls

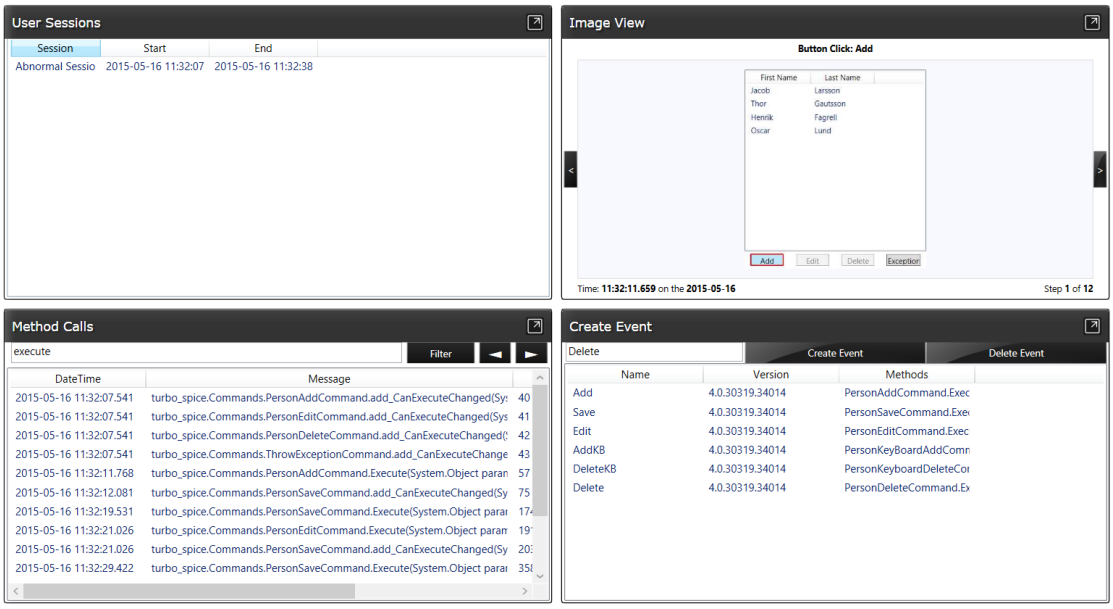

Figure A.6: Mapping method calls to an *event*. (An *event* is just an intermediate step in the mapping of flows and features)

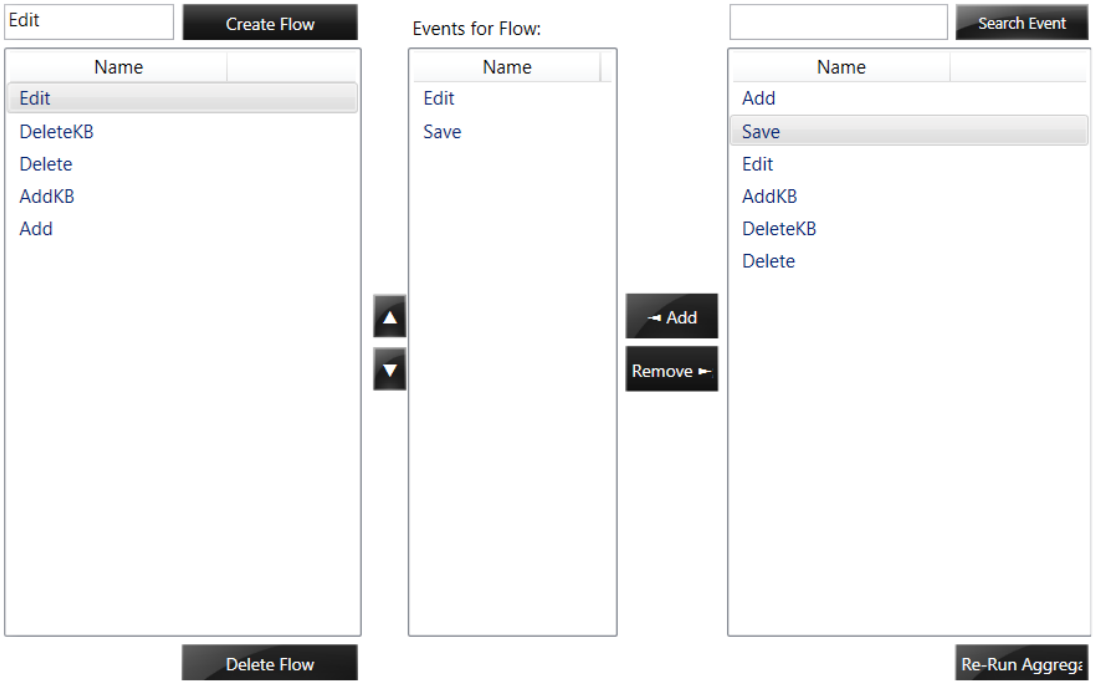

Figure A.7: Mapping *events* (method calls) to flows

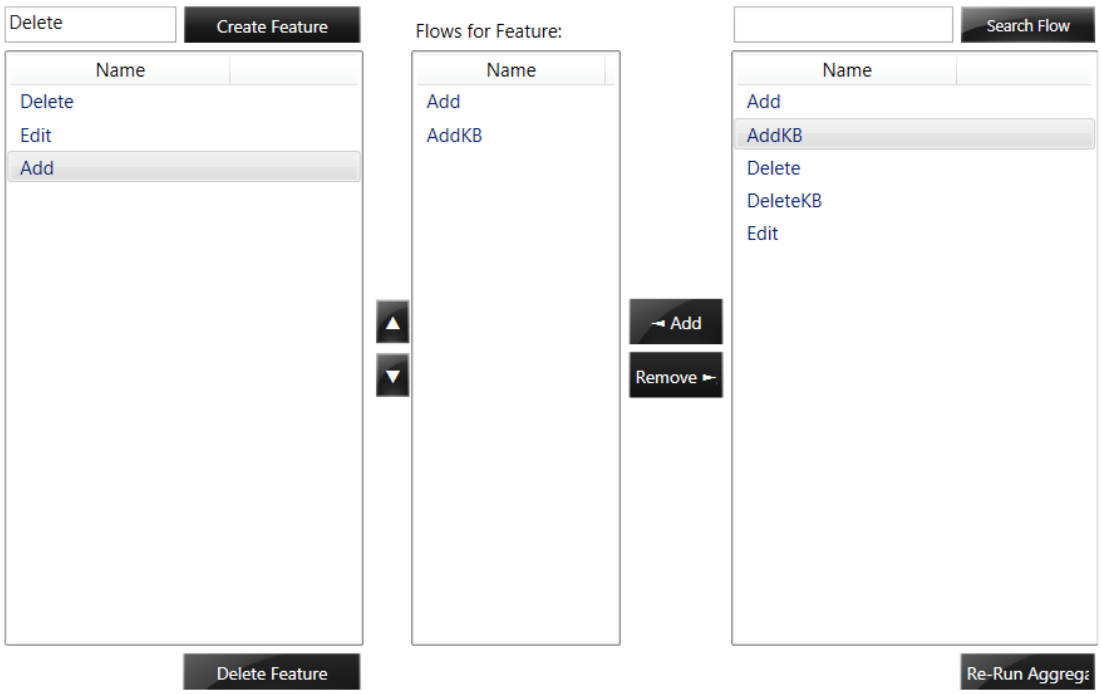

Figure A.8: Mapping flows to a given feature

B

Interview Questions

Introduction

The workshop was started by outlining the purpose of the study. This was done by first introducing the research question, giving relevant background information to the project, and then briefly discussing technical aspects such as e.g. intermediate languages. Thereafter, the participants were guaranteed anonymity and finally asked for their consent about recording the workshop.

Background

The purpose of the background questions was to collect information about the participants.

1. Is your area of work within the field of software development?
   (a) If not, what is your area of work?
   (b) If yes, for how long have you been doing work related to software projects?
2. Do you have any experience with Visual Studio?
3. Do you have experience with any logging tools?

The integration process

First a demonstration was given to show how the logging component (in the form of a NuGet) can be integrated with the target application. The following questions were then asked.

4. Would you say that the integration process is easy or difficult?
   (a) Why was it easy or difficult?
5. Would you trust yourself to integrate the logging component to an application in Visual Studio right now?
   (a) If not, why not?
6. Having seen how the integration process can look like, would you be more or less willing to use a similar logging component?

The target application

Next a demonstration of the target application (i.e. the one which the logging component was going to be tested on) was given. The following questions were then asked in order to see whether the workshop participants saw potential with a logging component prior to seeing how it presented data.

7. What are your first impressions of the application which was shown?
8. Do you think that an external logging component could help to find out which features of this application are used the most?
9. Do you think that an external logging component could help to debug this application by finding exceptions which occur?
10. Which functionality do you believe to be more valuable: Information about features, or about exceptions and crashes?

The logging component

Next the GUI component was opened to show how the logging component shows the data gathered from the previously performed demonstration of the target application.

11. After having seen how the logging component works, do you think that it gives useful information about the application you saw it used on?
    (a) If no, why?
12. What is the most useful feature of the logging component with regards to the application you saw it used on?
13. Judging from what you saw, do you think that the logging component can help to find faults in an application?
14. Do you think the logging component could help you to debug the application which was shown?
    (a) Do you think that it could help you generally to debug applications?
15. Do you think the logging component could help you to figure out which features of this application are important and which ones are not important?
    (a) Do you think that it could help you generally to do this?
16. After having seen all of this, which functionality do you think is more valuable: Information about exceptions or features?

Final thoughts

Finally, questions were posed which related to ethical aspects, the possibility of further features which could be added, the novelty of the logging component, and whether the participants would used it or companies be willing to pay for it.

17. What is your general feeling about this level of logging. Would you be comfortable with an application that you use at home being logged like this?
18. What about an application you use at work?
19. Does it matter whether it is an application you use in your free time or an enterprise application used at work?
20. Have you used any similar tools?
    (a) If yes, which ones?
21. Is there any important information which you feel that the logging component does not show?
22. Would you want to use the logging component in your own projects?

   (a) If yes, which of its functionalities would you likely use the most?

23. Do you think a company which you worked at would be interested in investing money to get functionality similar to the one provided by the logging component?

24. Is there anything you would like to add?

C

Survey Questions

1. My area of work is:
   ☐ Development
   ☐ Management
   ☐ Technical support
   ☐ Other (please specify) _____

2. I have this many years of work experience:
   ☐ 0-1 years
   ☐ 1-2 years
   ☐ 2-4 years
   ☐ 4-6 years
   ☐ 6-10 years
   ☐ 10+ years

3. I would be comfortable if an application that I use **for private matters** is being logged by the logging component
   ☐ Strongly agree
   ☐ Agree
   ☐ It doesn't matter to me
   ☐ Disagree
   ☐ Strongly disagree

4. I would be comfortable if an application that I use **at work** is being logged by the logging component
   ☐ Strongly agree
   ☐ Agree
   ☐ It doesn't matter to me

☐ Disagree
☐ Strongly disagree

5. I would be more comfortable using an application that is being logged at work rather than one I use for private matters
   ☐ Yes
   ☐ No, it's the same

6. I know one or several projects that I have been part of where the logging component would have been useful
   ☐ No, I don't
   ☐ Yes, 1-2 projects
   ☐ Yes, 3-4 projects
   ☐ Yes, 5-6 projects
   ☐ Yes, 7+ projects

7. I believe the logging component has potential to provide information that will facilitate the **debugging** of an application
   ☐ Strongly agree
   ☐ Agree
   ☐ Neither agree nor disagree
   ☐ Disagree
   ☐ Strongly disagree

8. I believe the logging component has potential to provide information that could aid in **further development** of an application
   ☐ Strongly agree
   ☐ Agree
   ☐ Neither agree nor disagree
   ☐ Disagree
   ☐ Strongly disagree

9. Of all the information provided by the logging component, the most important for me is:
   ☐ Information about feature usage
   ☐ Information about crashes/exceptions

10. I believe my company would be interesting in investing money to get functionality similar to the one provided by the logging component
    ☐ Strongly agree
    ☐ Agree
    ☐ Don't know
    ☐ Disagree
    ☐ Strongly disagree

D

Quantitative measurements

The quantitative data collected provided information about time to execute a sequence of operations in an application with and without using the logging component.

SikuliX script

Three applications where used with and without the logging component. For each application a number of operations where defined using a GUI-testing tool called SikuliX. For each application the script where used with and without the logging component. The sequence of operations where ran 100 times and the mean where then calculated. The SikuliX code used on PersonDatabase can be seen in code snippet D.1. Due to the length of the python code which SikuliX iterated through, only the code for PersonDatabase is presented in here. The code for the other two applications was structured similarly, but used features native to each application.

```
from collections import deque

times = []
i = 0
totalTime = time.time()

while (i < 100):
    firstNames = deque(["Zula","Cinda","Terrence","Tatyana","Alba","Lemuel"
        ,"Brandon","Mao","Clarice","Manie","Lars","Finn"])
    lastNames = deque(["Pitt","Chandler","Jones","Sims","Ward","Maynard","
        Bryant","Kinney","Preston","Wiggins","Kim","Fiol"])
    numberOfDownClicks = deque([1, 4, 2, 2, 2, 2, 2, 2, 2, 2])
```

```
    # Start taking time and open application
    s = time.time()
    click("logo.png")
    wait("titlebar.png",100)

    # Add 5 persons
    for names in range(0,5):
        click("add.png")
        type(Key.TAB)
        type(firstNames.popleft() + Key.TAB + lastNames.popleft())
        type(Key.TAB + Key.ENTER)
        type(Key.TAB + Key.TAB)

    # Edit 2 persons
    for name in range (0,2):
        type(Key.DOWN)
        if name == 1:
            type(Key.DOWN)
        click("edit.png")
        type(Key.TAB)
        type('a', KeyModifier.CTRL)
        type(Key.BACKSPACE)
        type(firstNames.popleft())
        type(Key.TAB)
        type('a', KeyModifier.CTRL)
        type(Key.BACKSPACE)
        type(lastNames.popleft())
        type(Key.TAB + Key.ENTER)

    # Delete 3 persons
    for names in range(0,3):
        for y in range(0, numberOfDownClicks.popleft()):
            type(Key.DOWN)
            click("delete.png")

    # Add 5 persons
    for names in range(0,5):
        click("add.png")
        type(Key.TAB + firstNames.popleft() + Key.TAB + lastNames.popleft()
            + Key.TAB + Key.ENTER)
        type(Key.TAB + Key.TAB)

    # Delete 5 persons
    for names in range(0,5):
        for y in range(0, numberOfDownClicks.popleft()):
            type(Key.DOWN)
            click("delete.png")

    # Delete the last person
    clickLocation = find("1430747830097.png").below(10)
    click(clickLocation)
    click("delete.png")
```

```
64      # Close the application and append the values
        click("close.png")
66      times.append(time.time()-s)
        i += 1
68
print "Each iteration took: " + str(times)
70  print "Average time: " + str(reduce(lambda x, y: x + y, times) / len(times)
        )
    print "Total loop time: " + str(time.time() - totalTime)
72  print "Total time divided by 100: " + str((time.time() - totalTime)/100)
```

Code snippet D.1: Python code executed by SikuliX

Application execution time

The data collected using PersonDatabase is presented in table D.1. The result is then analyses using a two sample t-test for comparing the means (with and without the logging component). The data is to be found in table D.2 and have been plotted to verify that it follows the normal distribution. The result from the student's t-test shows that using the logging component has an affect on the execution time for the application. In average the execution takes 1.24 s longer when having the logging component integrated. The result gathered by using ScreenToGif yields a similar result, there is a significant difference between the means. Here the difference is 0.79 s instead of 1.24 s. The analyse on Application X provided less precise result. The variance of the samples varied and therefore Welch t-test for unequal variance where used. Also the variance within each sampling were larger. Therefore it was not possible to reject the null hypothesis of equal means. That means that the time difference of 0.49 s does not necessarily have to be caused by the logging component.

With the logging component			Without the logging component		
43.2630000114	43.7130000591	44.0799999237	42.1959998608	42.4460000992	42.8459999561
43.3470001221	43.7139999866	44.0950000286	42.2460000515	42.4630000591	42.8470001221
43.3650000095	43.7149999142	44.1130001545	42.246999979	42.4650001526	42.881000042
43.3819999695	43.7289998531	44.114000082	42.3110001087	42.4789998531	42.8949999809
43.4140000343	43.7459998131	44.114000082	42.3129999638	42.4790000916	42.8949999809
43.4309999943	43.7470002174	44.114000082	42.3139998913	42.4790000916	42.8959999084
43.5130000114	43.7589998245	44.1430001259	42.3169999123	42.4790000916	42.8959999084
43.5150001049	43.7630000114	44.1459999084	42.3289999962	42.4800000191	42.9140000343
43.5289998055	43.7790000439	44.1480000019	42.3289999962	42.4949998856	42.9140000343
43.5290000439	43.7809998989	44.1480000019	42.3289999962	42.4959998131	42.9140000343
43.5310001373	43.7810001373	44.1640000343	42.3289999962	42.496999979	42.9190001488
43.5320000648	43.7969999313	44.1789999008	42.3299999237	42.5089998245	42.9299998283
43.5470001698	43.7979998589	44.1800000668	42.3300001621	42.513999939	42.9300000668
43.5629999638	43.8120000362	44.1840000153	42.3310000896	42.5249998569	42.9300000668
43.5659999847	43.8469998837	44.1969997883	42.3450000286	42.5299999714	42.9300000668
43.5810000896	43.8629999161	44.2139999866	42.3459999561	42.5310001373	42.9620001316
43.5959999561	43.9129998684	44.2150001526	42.3459999561	42.5480000973	42.9639999866
43.6129999161	43.9130001068	44.2299997807	42.3619999886	42.5780000687	42.9639999866
43.614000082	43.9300000668	44.243999958	42.3619999886	42.6130001545	42.9659998417
43.629999876	43.9649999142	44.2460000515	42.3630001545	42.6789999008	42.9800000191
43.6300001144	43.9679999352	44.2460000515	42.3630001545	42.6970000267	42.9800000191
43.631000042	43.9800000191	44.246999979	42.364000082	42.7620000839	42.9800000191
43.6460001469	43.9819998741	44.2479999065	42.3790001869	42.7799999714	43.0119998455
43.6470000744	43.9960000515	44.263999939	42.3800001144	42.7809998989	43.0149998665
43.6630001068	43.9970002174	44.2799999714	42.381000042	42.7810001373	43.0310001373
43.6639997959	44.0289998055	44.2969999313	42.3959999084	42.7960000038	43.0460000038
43.6639997959	44.0290000439	44.2969999313	42.3960001469	42.7969999313	43.0469999313
43.6640000343	44.0299999714	44.2969999313	42.4119999409	42.7969999313	43.0629999638
43.6750001907	44.0469999313	44.2969999313	42.4130001068	42.8090000153	43.0629999638
43.6789999008	44.0470001698	44.2970001698	42.4299998283	42.8130002022	43.0799999237
43.6970000267	44.0629999638	44.3819999695	42.4299998283	42.8289999962	43.1289999485
43.6970000267	44.0639998913	44.4769999981	42.4299998283	42.8299999237	43.1629998684
43.6979999542	44.0640001297		42.4300000668	42.8299999237	
43.7090001106	44.0789999962		42.4309999943	42.8459999561	

Table D.1: Time to execute code for PersonDatabase

	With	Without
Mean	43.88762	42.64425
Variance	0.081394959	0.073413477
Observations	100	100
Degrees of freedom	198	
t-value	31.60116347	

α=0.05	One sided	Two sided
P(T<=t)	1.38859E-79	2.77719E-79
t-critical	1.652585784	1.972017478

Table D.2: Statistics for PersonDatabase using student's t-test for equal variance

With the logging component			Without the logging component		
17.4310002327	17.7460000515	18.5690000057	16.5810000896	17.5009999275	17.5829999447
17.5410001278	17.746999979	18.5750000477	16.5880000591	17.5039999485	17.5879998207
17.5680000782	17.7569999695	18.5759999752	16.5900001526	17.5039999485	17.5920000076
17.5710000992	17.7599999905	18.5820000172	16.5939998627	17.5069999695	17.5930001736
17.5770001411	17.7709999084	18.5939998627	16.6019999981	17.5199999809	17.5970001221
17.6000001431	17.7899999619	18.5939998627	16.6070001125	17.5250000954	17.5980000496
17.6129999161	17.9580001831	18.5999999046	16.638999939	17.5290000439	17.5989999771
17.6330001354	18.0500001907	18.6019999981	16.6770000458	17.5309998989	17.6019999981
17.6370000839	18.0550000668	18.6150000095	16.6849999428	17.5350000858	17.6080000401
17.6400001049	18.114000082	18.623000145	16.6909999847	17.5360000134	17.6119999886
17.6460001469	18.1199998856	18.6349999905	16.6970000267	17.5379998684	17.6180000305
17.6460001469	18.1210000515	18.6429998875	16.7049999237	17.5389997959	17.6190001965
17.6480000019	18.1589999199	18.6519999504	16.7090001106	17.5390000343	17.625
17.6540000439	18.1610000134	18.6789999008	16.7309999466	17.5400002003	17.6289999485
17.6730000973	18.2269999981	18.6870000362	16.7660000324	17.5429999828	17.6349999905
17.6769998074	18.2780001163	18.7100000381	16.7730000019	17.5439999104	17.638999939
17.6790001392	18.3150000572	18.7109999657	16.8680000305	17.5479998589	17.6400001049
17.6839997768	18.379999876	18.7589998245	17.3059999943	17.5480000973	17.6410000324
17.6949999332	18.4259998798	18.7820000648	17.3110001087	17.5490000248	17.6419999599
17.6970000267	18.4849998951	18.9749999046	17.4039998055	17.5520000458	17.6419999599
17.6989998817	18.4919998646	18.9889998436	17.4079999924	17.5529999733	17.6430001259
17.7000000477	18.5039999485	19.0209999084	17.4149999619	17.5529999733	17.6510000229
17.7019999027	18.5050001144	19.0230000019	17.4390001297	17.5559999943	17.6519999504
17.7030000687	18.5069999695	19.0299999714	17.4409999847	17.5600001812	17.6519999504
17.7100000381	18.5120000839	19.0510001183	17.4500000477	17.5600001812	17.6800000668
17.7109999657	18.5169999599	19.0759999752	17.4659998417	17.5659999847	17.6829998493
17.7119998932	18.518999815	19.0780000687	17.4689998627	17.5679998398	17.6970000267
17.7200000286	18.5199999809	19.0920000076	17.4800000191	17.5699999332	17.7460000515
17.7219998837	18.5250000954	19.0999999046	17.4819998741	17.5699999332	17.7920000553
17.7290000916	18.5250000954	19.1199998856	17.4909999371	17.5750000477	17.9750001431
17.7349998951	18.5410001278	19.131000042	17.4940001965	17.5769999027	18.4909999371
17.7369999886	18.5450000763	19.1600000858	17.4989998341	17.5769999027	20.2509999275
17.7369999886	18.5539999008		17.4990000725	17.5789999962	
17.7389998436	18.5620000362		17.4990000725	17.5829999447	

Table D.3: Time to execute code for ScreenToGif

	With	Without
Mean	18.23348	17.4485
Variance	0.265598756	0.211632487
Observations	100	100
Degrees of freedom	198	
t-value	11.36303013	

α=0.05	One sided	Two sided
P(T<=t)	1.16057E-23	2.32115E-23
t-critical	1.652585784	1.972017478

Table D.4: Statistics for ScreenToGif using student's t-test for equal variance

With the logging component			Without the logging component		
51.4779999256	54.5720000267	57.4149999619	44.8919999599	53.3359999657	58.6900000572
51.7809998989	54.6960000992	57.7279999256	45.0269999504	53.5180001259	58.7859997749
51.8900001049	54.8510000706	57.7750000954	45.0279998779	53.5379998684	58.8129999638
52.1290001869	54.9200000763	57.8370001316	45.2820000648	53.7590000629	58.8289999962
52.137999773	55.0230000019	57.9379999638	45.7630000114	54.1779999733	59.0440001488
52.2029998302	55.0280001163	57.9679999352	46.5900001526	54.375	59.0480000973
52.2710001469	55.0490000248	58.0069999695	46.7129998207	54.5420000553	59.0859999657
52.3289999962	55.0590000153	58.0339999199	47.6480000019	55.1490001678	59.0889999866
52.3929998875	55.1979999542	58.2049999237	48.7650001049	55.2949998379	59.1279997826
52.5970001221	55.2249999046	58.3119997978	49.1840000153	55.3660001755	59.1510000229
52.8280000687	55.2449998856	58.4779999256	49.7860000134	55.3679997921	59.2110002041
52.8680000305	55.3090000153	58.6890001297	49.8819999695	55.3770000935	59.4049999714
52.8919999599	55.4300000668	59.4700000286	50.4190001488	55.7369999886	59.5429999828
52.9299998283	55.5680000782	59.5409998894	50.4819998741	55.9119999409	59.5710000992
53.0549998283	55.6399998665	59.763999939	50.496999979	55.9200000763	60.242000103
53.2269999981	55.6659998894	59.9319999218	50.9630000591	56.0900001526	60.2620000839
53.2339999676	55.7750000954	60.0359997749	50.9879999161	56.378000021	60.2869999409
53.2540001869	55.8129999638	60.4670000076	51.0069999695	56.4019999504	60.3910000324
53.2599999905	55.868999958	60.5069999695	51.0950000286	56.8429999352	60.5469999313
53.3070001602	55.9160001278	60.9560000896	51.4470000267	57.0940001011	60.6649999619
53.3230001926	56.0720000267	61.0609998703	51.4819998741	57.2320001125	60.7300000191
53.368999958	56.3970000744	61.1990001202	51.5099999905	57.254999876	60.8090000153
53.385999918	56.4070000648	61.2269999981	51.7870001793	57.3230001926	61.5989999771
53.4579999447	56.4169998169	61.4119999409	51.8199999332	57.4419999123	62.1669998169
53.518999815	56.5230000019	61.4769999981	52.3629999161	57.4560000896	62.8480000496
53.5550000668	56.6159999371	61.4800000191	52.4869999886	57.75	63.1330001354
53.5850000381	56.6910002232	61.5279998779	52.4990000725	57.7699999809	65.1800000668
53.7180001736	56.7779998779	62.3080000877	52.753000021	57.8880000114	65.381000042
53.7380001545	56.8870000839	63.371999979	52.8579998016	58.4149999619	65.9210000038
53.7839999199	56.9010000229	64.6410000324	52.8800001144	58.4479999542	67.6160001755
54.1619999409	57.0969998837	66.4070000648	53.0650000572	58.4539999962	68.8240001202
54.228000164	57.2599999905	68.0440001488	53.0729999542	58.4699997902	71.3499999046
54.2430000305	57.2739999294		53.253000021	58.6679999828	
54.4190001488	57.3380000591		53.2790000439	58.6739997864	

Table D.5: Time to execute code for Application X

	With	Without
Mean	56.36276	55.87335
Variance	11.38949304	27.81692938
Observations	100	100
Degrees of freedom	168	
t-value	0.781617419	

α=0.05	One sided	Two sided
P(T<=t)	0.217769789	0.435539578
t-critical	1.653974208	1.974185191

Table D.6: Statistics for Application X using Welch t-test for unequal variance

E

User Stories

The user stories are grouped into different viewpoints, depending on which target group they relate to the most. The viewpoints are those of:

- A user of an application which have the logging component integrated (id: UL)
- A developer working on an application that is logged by the logging component (id: DL)
- A user of the Ozzy GUI component (id: UG)

A user of a logged application

ID:	UL1	**Dependency:**
Origin:	User of logged application	
User Story:	Users should not have to specify a username for the logging component to use	
Description:	The logging component should not require the user to enter any data manually in order to get started using the applications, except for accepting that the logging component is used	
Rationale:	The logging component should not require any data to be entered by the users since that would have a negative impact on the user experience	

ID:	UL2	**Dependency:**
Origin:	User of logged application	
User Story:	The logging component should not slow down the application	
Description:	A logging component is only useful if it does not negatively affect performance and latency of an application to a degree where it is possible for the user to notice a difference	
Rationale:	Performance is important since the user experience should not be affected	

ID:	UL3	**Dependency:**
Origin:	User of logged application	
User Story:	The logging component should notify the user that they are being logged	
Description:	By asking the user for their permission when the application start for the first time	
Rationale:	To make the user aware that they are being logged	

A developer working on an application that is logged by the logging component

ID:	DL1	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The logging component should be possible to install through the NuGet Package Manager in Visual Studio	
Description:	Visual Studio and the NuGet Package Manager is described in section 2.5	
Rationale:	To shorten the time it takes to get started using the logging component	

ID:	DL2	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The logging component should be completely separate from the presentation layer (GUI component) of the logged data	
Description:	Create a separate component for the GUI	
Rationale:	No user of the application that is using the logging component should have access to the presentation layer	

ID:	DL3	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The logging component should be able to integrate by using code injections	
Description:	By using code injections the required code for the application that want to use the logging component can be automatically added to an application during compilation	
Rationale:	To shorten the time required for integration and to keep the original application code more pristine	

ID:	DL4	**Dependency:** DL3
Origin:	Developer using the logging component	
User Story:	The functionality for the code injection should be separated from the logging component	
Description:	Create a separate component for the code injections	
Rationale:	To allow the developers to select whether they want to use the code injection or not. If not, they can add the log statements themselves	

ID:	DL5	**Dependency:** DL3
Origin:	Developer using the logging component	
User Story:	The logging component should provide a way for the developers to manually select which methods to inject logging statements to	
Description:	By enabling the developers to manually remove the logging of certain methods, the logging component can better be adapted to fit more applications	
Rationale:	Performance problems can arise when a method, that is being logged, is executed thousands of times in a short time slot	

ID:	DL6	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The logging component should provide a way for the developers to separate their own code from the code for the logging component	
Description:	By providing code injections and a separate settings file to specify how the logging component should behave, there is no need for the developers to modify their own code	
Rationale:	Some developers do not want to "contaminate" their code with for example log statements	

ID:	DL7	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The data should be stored in a MongoDB database	
Description:	MongoDB is described in section 2.6	
Rationale:	MongoDB is easy to get started with since it does not require a predefined database structure and it has ability to scale horizontally	

ID:	DL8	**Dependency:**
Origin:	Developer using the logging component	
User Story:	The logging component should function even if the connection to the remote database is lost	
Description:	By storing data locally for a defined time period, data can still be collected without constant connection to the remote database. In order to not consume large amount of hard drive space, the data will be erased after a defined time period	
Rationale:	It is common e.g that people use applications offline or while they are connected to a protected network, where it is not possible to reach the remote database	

A user of the Ozzy GUI component

ID:	UG1	**Dependency:**
Origin:	User of the Ozzy GUI component	
User Story:	The GUI component should present parts of the data using dashboards	
Description:	Present data using a dashboard. Dashboards are described in section 2.7	
Rationale:	By using a dashboard it is possible to quickly get an overview of the data without having to navigate through many views	

ID:	UG2	**Dependency:** UG1
Origin:	User of the Ozzy GUI component	
User Story:	The GUI component should be able to maximize and minimize different views of the dashboards	
Description:	By having a minimize and maximize button in the top right corner of every small view in the dashboard	
Rationale:	Some views in the dashboards might be difficult for the user to read or understand because of the limited space in the dashboard	

ID:	UG3	**Dependency:**
Origin:	User of the Ozzy GUI component	
User Story:	The GUI component should be able to present logged data for a selected user	
Description:	Selecting a user from a list and then presenting the screenshots and exceptions for that user	
Rationale:	To better understand how a user interacts with the applications and to see what the user did before an error occurred	

ID:	UG4	**Dependency:**	UG3
Origin:	User of the Ozzy GUI component		
User Story:	The GUI component should present clearly which user is selected or if statistics for all users are being displayed		
Description:	By always displaying, somewhere in each view, whether "all users" are selected or whether a particular user is selected. If one user is selected it will display the user ID for that user		
Rationale:	To help users navigate the GUI		

ID:	UG5	**Dependency:**	UG1
Origin:	User of the Ozzy GUI component		
User Story:	The dashboards of the GUI component should provide a way to move the different views within each dashboard around		
Description:	My making it possible to drag and drop the different views in the dashboard		
Rationale:	To view related data in the way that is best for the users		

ID:	UG6	**Dependency:**	UG7, UG8
Origin:	Developer using the GUI component		
User Story:	The GUI component should show the most common exceptions in the application		
Description:	Showing a chart of the most common exceptions		
Rationale:	By knowing which exceptions that is most common the developers can prioritize which exceptions to fix first		

ID:	UG7	**Dependency:**	
Origin:	Developer using the GUI component		
User Story:	The GUI component should show a list of all unhandled exceptions for a user		
Description:	Viewing a list with the name of the exception and the time when it occurred		
Rationale:	Providing a way for the developer to view the exceptions that have crashed the application		

ID:	UG8	**Dependency:**
Origin:	Developer using the GUI component	
User Story:	The GUI component should show a list of all handled exceptions for a user	
Description:	Viewing a list with the name of the exception and the time when it occurred	
Rationale:	Sometimes an unhandled exception is caused by a handled exception. In order to fix the unhandled exception, information about the handled exception could therefore be necessary	

ID:	UG9	**Dependency:** UG7, UG8
Origin:	Developer using the GUI component	
User Story:	The GUI component should show information about each exception	
Description:	For each exception the following information should be presented: stacktrace, exception type, message, timestamp, target site, source and inner exceptions	
Rationale:	To give the developers the data necessary to troubleshoot the application	

ID:	UG10	**Dependency:** UG7, UG8
Origin:	Developer using the GUI component	
User Story:	The GUI component should provide a way to browse through screenshots of all the steps performed by the user up to 40 seconds before an exception occurred	
Description:	Browse through screenshots for a selected exception	
Rationale:	To help the developers to understand what the user did before the exception occurred	

ID:	UG11	**Dependency:**
Origin:	User of the Ozzy GUI component	
User Story:	The GUI component should provide a way to show which features of an application are used the most/least and the flow that where used to access the features	
Description:	By mapping method calls to flows and then mapping flows to features it should be possible to get statistics about features and information about the flow that where used to access the feature	
Rationale:	To help the developing organization to better understand how their application is used	

ID:	UG12	**Dependency:**
Origin:	Developer using the Ozzy GUI component	
User Story:	The GUI component should provide a way to show which methods of an application are called most often	
Description:	By mapping method calls to features it should be possible to get statistics about features	
Rationale:	To help the developers to better understand where time could be spent to improve the code	

ID:	UG13	**Dependency:**
Origin:	Developer using the Ozzy GUI component	
User Story:	The GUI component should list all the sessions	
Description:	A list with all the sessions for a selected user containing information about start time, end time and if the sessions ended with an exception	
Rationale:	Enables the developers to view what the user did in a session to help them debug exceptions or to better understand how the users use the application	

ID:	UG14	**Dependency:**	UG13
Origin:	Developer using the Ozzy GUI component		
User Story:	The GUI component should provide a way to browse through all the screenshots for a session		
Description:	To be able to go through, step by step, what the user did during the session by showing screenshots accompanied by a timestamp		
Rationale:	Enables the developers to see what the user did		

ID:	UG15	**Dependency:**	UG14
Origin:	Developer using the Ozzy GUI component		
User Story:	The GUI component should display all the method calls from the application code that corresponds to the action associated with the screenshot		
Description:	For each screenshot a list with all the method calls should be displayed. The list should contain information about the namespace, the class name and a timestamp for the method		
Rationale:	By viewing method calls it is possible for developers to follow the execution of the code		

ID:	UG16	**Dependency:**	UG11
Origin:	Manager using the Ozzy GUI component		
User Story:	The GUI component should display a list with all the features used by the users of the application that is being logged		
Description:	A list with all the features that have been used in the application and information about the feature should be possible to view		
Rationale:	Gain better understanding about the application and to e.g better prioritize further development		

ID:	UG17	**Dependency:**
Origin:	Manager using the Ozzy GUI component	
User Story:	The GUI component should display at what time of day the application is used the most	
Description:	By plotting a chart showing at what hours the application is used	
Rationale:	To e.g. know when to schedule for service downtime	

ID:	UG18	**Dependency:**
Origin:	Manager using the Ozzy GUI component	
User Story:	The GUI component should display general statistics for the application	
Description:	Total number of users, average number of sessions for a user, total number of sessions, average time for a session and average number of features used per session	
Rationale:	To better understand the application usage	

ID:	UG19	**Dependency:** UG11
Origin:	Manager using the Ozzy GUI component	
User Story:	The GUI component should display general statistics for a feature	
Description:	Average time to used the feature, how many times the feature have been used, percentage of how many users who use the feature, percentage of how many sessions have the feature been used in, which flows have been used for this feature, how many users have used each flow and what is the average time for a flow	
Rationale:	To better understand the application usage	