Collecting Product Usage Data Using a Transparent Logging Component

Thorvaldur Gautsson, Jacob Larsson

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
e-mail: {gautsson, jacobla}@student.chalmers.se

Miroslaw Staron

Department of Computer Science and Engineering
University of Gothenburg
Gothenburg, Sweden
e-mail: miroslaw.staron@gu.se

***Abstract*—Continuous software engineering and experiments on released products have become very popular in modern software development. In this paper, we present a software component used to transparently log usage data from products in order to facilitate the use of customer-usage data by software developers. Such a component can aid with software maintenance and life-cycle management, but also provide help in software production and validation. We present a technical solution, the evaluation of its influence on the performance of a sample product, and an initial study on the acceptance of such a technology by regular users. Our results show that the mechanisms available in modern programming languages make it possible to integrate such a component without manual interventions in product code and that users are generally positive towards using this technology for logging the usage of work-related applications. We conclude that this type of technology can provide new possibilities for developers to adjust product planning based on the customer usage data.**

Keywords–Logging; usage patterns; features; data analysis.

I. INTRODUCTION

Modern software development emphasizes the need for rapid delivery of customer value and many software development companies use the principles of continuous software engineering to deliver on these needs [1][2][3]. The concept of continuous software engineering drives the development of technology towards continuous integration, continuous deployment and continuous feedback from customers. Continuous integration allows companies to decrease the internal feedback cycles on the quality of software by advocating quick delivery of small software increments and testing them directly after integration with the rest of the software product code. The continuous deployment philosophy prescribes methods and tools for delivering software without the need for manual installation (e.g., changing a web application) and finally the continuous feedback from customers is often realized as customer-experiments (also known as A/B testing [4]).

In this paper, we contribute to the area of rapid feedback from customers by developing a logging component which can be integrated with software products (e.g., desktop applications) without the need to modify the product code. The logging component collects data about the usage of features and functions — both per user and per feature. It also provides the possibility to store strategically taken screenshots of the GUI (*graphical user interface*) of an application, and enables analysing the status of applications before exceptions or crashes occur. Collecting this kind of data has the potential to speed up the development feedback substantially compared to the currently used data (c.f. [5]). The research question which we analyzed in our research 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?

We set off to design a component which could be integrated with existing products without manual intrusion in the product source code, though recompilation was allowed. We also defined the term *usage pattern* 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 occur. The logging component scrutinized in the research therefore provides developers with the flexibility to define the precise usage pattern that constitutes a feature.

Although there exists a body of research on both the logging of software applications as well as managing large quantities of data (c.f. [6][7]), there is still much to be researched. Our working hypothesis was that analyzing feature usage with the help of screenshots and unconstrained logging of method-calls can be highly valuable for developers. Such an approach has been found to help companies to remain innovative in the long run [8].

Our results show that automated logging of method-calls combined with pre-defined packaging of the sequences of method-calls into features can help developers to understand the usage patterns of an application. Based on a survey which was conducted, we also found that the logging of feature usage is generally met with a positive attitude for work-related applications, but skepticism for privately used software. The study led us to conclude that the largest developmental benefits are the combination of logging and pre-definition of features — which is a rather unique approach.

This paper is structured as follows: Section II presents some of the most related work in the field of customer data collection and continuous software engineering. Section III presents the design of our research. Section IV presents the logging component and its design. Section V presents the results from the evaluation of the logging component. Finally, Section VI presents the conclusions from our study.

II. RELATED WORK

Backlund et al [9] studied post-deployment data collection by conducting a case study on a web-based portal system.

 ISBN: 978-1-61208-458-9

They began by identifying which quantitative data needed to be collected, collected it and finally compared the collected data with survey answers from test subjects. The quantitative data used was collected through aspect-oriented programming and included various user actions, such as button clicks and task completion times. The authors 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 to perform.

Olsson et al. [5][10] studied patterns in post-deployment data collection in products from three companies in the embedded software development field. Their results show that the collected post-deployment data that the companies used came from the operating system, or concerned performance. They found that while feature-usage data is valuable, it is generally not collected. In our work, we address this challenge in the context of applications written in C#.

Lindgren et al. [4] studied the implications of using experiment systems in the development of software. Through a survey among companies, they found that there is still no consensus on how to collect customer feedback and how to use it during development. They also found that customer feedback is rarely used during development. Our work contributes to the ability to collect data automatically, thus presenting a way to use customer data during development.

Börjesson and Feldt evaluated in 2012 two tools for automated visual GUI testing on a software system developed by a Swedish aerospace company. They found that visual GUI testing can perform better than manual testing practices and that it furthermore has benefits over manual GUI testing techniques. They stated however that visual GUI testing still had challenges which had not been addressed [11][12].

III. RESEARCH DESIGN

The research presented in this paper utilized two research methods: design science research for the development of the logging component and a case study for its evaluation.

A. Design science research

The design science research approach [13][14] 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.

In order to ensure industrial applicability, the logging component was developed in cooperation with Diadrom Systems AB (hereafter: Diadrom) 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 evaluation processes.

Before the development phase began, a prototype 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 fictitious company.

B. Case study

After the design science research phase was over, the logging component was further evaluated through a case study [15]. Two applications which had previously been developed by Diadrom were used in this phase. 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 consisted of the following steps:

1) Two suitable applications for evaluation were identified: an application from Diadrom (hereafter: Application X), consisting of ca. 40 000 LOC; and an open source application named *ScreenToGif* which allows users to record an area of their screen, manipulate, edit, and then save as a gif image file [16].
2) Qualitative data was collected through workshops and quantitative data by using the aforementioned applications.
3) The collected data was analyzed.

1) Interviews: 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. Both workshops followed the same structure. First, the project was introduced 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 logging component was discussed and questions about its benefits were posed. The remaining part of the workshop was then used to present open-ended questions.

2) Measurements: Several criteria were defined which the logging component had to fulfill, along with ways to measure them. 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 previously mentioned. 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.

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,

 ISBN: 978-1-61208-458-9

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 both with and without the logging component it was possible to investigate whether the it slowed down the application significantly.

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.

C. Analysis methods

The Student's t-test was used to see whether 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, as it performs better for data sets of unequal variance.

The null hypothesis was that there should not be a significant time difference in executing the sequence of operations dependent on whether the logging component was integrated with the application or not.

IV. The Logging Component

The purpose of the logging component was to log various user actions and program behavior and store those logs in a remote database. Among the user actions logged were keyboard input and mouse clicks. Handled and unhandled exceptions, as well as method-calls, were also logged. Screenshots were taken when a user interacts with the application — to make it possible to understand how the application behaved from the users' point of view.

The way the logging component operated was by weaving log 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 IV-A.

Immediately after the logging component started to generate data, the need for a 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. The development therefore resulted in three different modules:

- a logging module that logged usage patterns
- a GUI module for presenting the data
- a weaving module for the code injections

Together these three modules constituted a system which defines logging and presentation of data for C# WPF (Windows Presentation Foundation) applications.

A. Weaving Module

The Weaving module only had one purpose: to inject code into a target application in order to connect it with the logging module. 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 module that would automatically install Fody.

B. Logging Module

The logging module was developed using C# WPF. To store the large amount of data that the logging module produced, a NoSQL database called MongoDB was used due to its flexibility. The logging module provided an interface for logging information and timestamps for the following:

- Method-calls
- Handled and unhandled exceptions
- Mouse clicks and mouse scroll (start and stop)
- Keyboard button clicks
- Specified Keyboard Shortcuts

The logging module logged method-calls in order to track the data flow of an application. The method logs contain data concerning 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 exceptions is also important and an interface was provided for logging both handled and unhandled exceptions. To log handled exceptions, it was necessary to insert a log statement into every "catch"-block in an application, which is automatically done by the weaving module. To log unhandled exceptions, an event handler in C# WPF was used.

To capture user interaction, screenshots were taken for mouse clicks, mouse scroll, button clicks and specified keyboard shortcuts. All screenshots contained a timestamp to make it possible to follow the interaction between user and computer in a sequential order. All saved logs, except for the exception logs, were accompanied by a screenshot. The purpose of the screenshots was to allow developers to view what actions a user had taken; it made it for instance possible to view what a user did before an exception occurred or how a user used a certain feature.

C. GUI Module

The GUI module was developed in C# WPF. The GUI module displays data for a selected individual user or aggregated data for all users. Special sub-menus for *statistics* and *features* sub-menus were available for both one selected user as well as in the form of aggregated data from all users. A figure of the statistics view for one users is displayed in Figure 1. 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

 ISBN: 978-1-61208-458-9

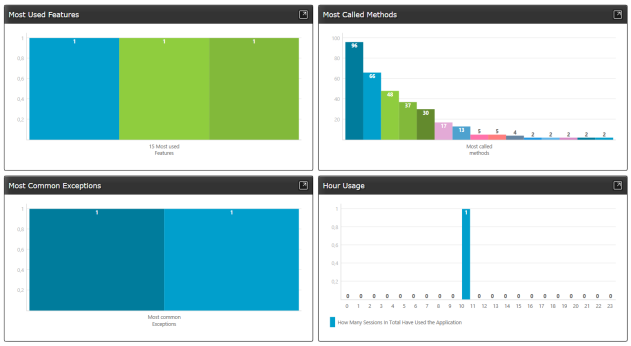

Figure 1. Statistics sub-menu showing: most called methods, when an application is used, most common exceptions and most used features

General information presented about an application included the following: (i) total number of users, (ii) average number of sessions per user, (iii) average sessions which crashed the application per user, (iv) total number of sessions which crashed the application, (v) total number of sessions, (vi) average time for a session, (vii) average number of different flows used per session, and (viii) average number of features used per session.

Defining features as sequences of method-calls

In order to know how the users of an application use its features, those features need to be defined in some way. This was achieved through the GUI module, which provided functionality for defining and mapping what method calls a feature consists of.

To give a concrete example of feature definition, we can consider the previously mentioned PersonDatabase application. This application enabled its users to add a person to a database by pressing an *add* button, then entering the first and last names in respective input boxes, and finally clicking on a *save* button. In this case, it seems reasonable that the *add* feature finishes when the *save* button has been pressed. The above scenario will trigger methods to be called in the application, and all method-calls are logged by default. The *add* feature can therefore be defined as a sequence of method calls that begins when the user clicks on the *add* button and ends when the user clicks on the *save* button. The feature definition is therefore kept 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.

In addition to a sequence of method-calls being mapped to a feature, there can be several *flows* leading to the same feature being used. For example, a person could be added to *PersonDatabase* by using an *add* button or perhaps by using a keyboard shortcut. Each flow is defined by one or several events, where an event is a method-call that has been defined by developers as *important* using the GUI module. An example of an event would be the first 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 calculate statistics about feature usage. How the mapping process functions is illustrated in Figure 2.

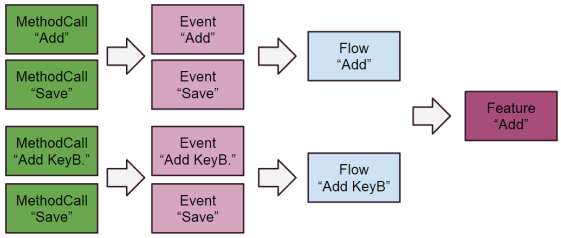

Figure 2. The different steps in the mapping of a feature.

After the features have been mapped using the method in Figure 2, various statistics can then be viewed in the GUI module. The feature usage view for all users is displayed in Figure 3.

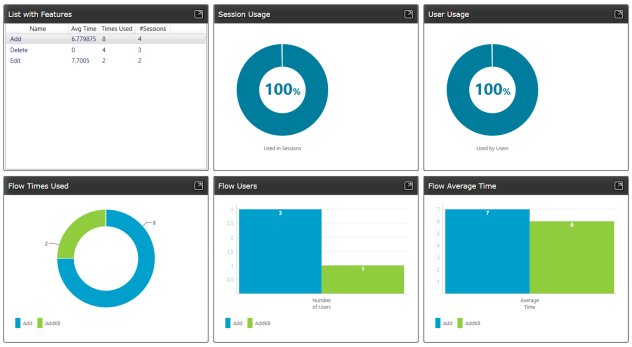

Figure 3. Feature statistics. The bottom row shows data on feature flows

For every feature, it is possible to view statistics for the average execution time, number of times it was used, 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 has been used, how many users have used the flow and what the average time for each flow is. A similar view is shown if a single user is selected.

V. Evaluation Results

After development had ceased and all data had been gathered, the results were evaluated and conclusions drawn.

A. Measurements

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 is summarized in Table I. The sequence of steps was automatically executed 100 times and the results then averaged. The data gathered was then tested by using a t-test with $\alpha = 0.05$ to see if there was any significant difference in execution time with and without the logging component. For PersonDatabase and ScreenToGif there was a significant difference, but not for Application X. For PersonDatabase and ScreenToGif the difference observed is therefore likely caused by the integration of the logging component. The data collected for Application X had a much

 ISBN: 978-1-61208-458-9

higher variance than the data collected for the other two applications, which is the reason to why no significant difference could be confirmed.

The performance measurements therefore indicate that the logging component has some negative effect on the performance of an application. How large the effect is depends on the nature of the logged application. For instance, an application which has a method that is called thousands of times during a short interval will likely create latency issues and might even cause the logging component to run out of memory. In those cases it would be necessary to disable logging for that particular method. What the performance measurements appear to show is that in most cases the time difference is within an acceptable range, as the observed time difference in all three cases would be quite hard for a regular user to notice.

TABLE I. AVERAGE TIME TO EXECUTE A SEQUENCE OF STEPS

	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

Measurements of CPU usage were gathered using a CPU measurement tool in Visual Studio by running PersonDatabase and ScreenToGif with the logging component. The results are presented in Table II. The CPU usage of the logging component was divided into three areas: method-call, 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 is that ScreenToGif requires more CPU computation in general just to run the application. PersonDatabase is a small application with a relative low CPU usage.

TABLE II. HOW MUCH OF THE TOTAL CPU POWER OF AN APPLICATION THE LOGGING COMPONENT USES

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

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 Tables III and IV. *Count* defines how many logs the database contained and *average object size* is calculated by dividing the *total size* with *count*.

TABLE III. DATABASE SIZE FOR STORING IMAGES

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 IV. DATABASE SIZE FOR STORING METHOD-CALLS

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

B. Interviews

In order to evaluate whether an external logging component could be used to aid software development by providing developers with information about usage patterns, two workshops were held to obtain qualitative data. The participants in the first workshop were four students in the Software Engineering M.Sc. programme at Chalmers University of Technology. This workshop was used as a pilot study before conducting the workshop at our industrial partner. All of the subjects considered software development to be their area of work and they all had previous industrial experience which ranged from 2 to 7 years. The main findings from the workshop were:

1) The integration process using weaving was perceived as straightforward as it required only a few mouse clicks in the Visual Studio GUI.
2) A set of situations when the logging component should be modified — e.g., a case of a function which crashed when being logged (because of buffers) — were noted.
3) The logging component was not found to hinder application performance. Furthermore, a suggestion was raised that logging component could be used to re-architecture an application to its improve performance.
4) It was considered essential to conduct a survey to find how users would perceive being logged.

The second workshop was held at Diadrom and the participants were four developers with years of experience in developing software applications. All participants had previous experience with Visual Studio and all had used logging tools of some kind at some point. The target application which the logging component was integrated with in the workshop was built for a Swedish aerospace company. The results were:

1) The practitioners perceived the integration process to be straightforward.
2) The practitioners suggested further uses — for instance customization of which design-time elements should be logged (e.g., namespaces).
3) The practitioners identified further development areas — e.g., a management view, adding temporal aspects or presentation of user clicks as a heat-map.

Finally, the participants concluded 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. As long as the logging component was used for debugging or developmental purposes, they found logging to be acceptable.

C. Survey

It was considered essential to conduct an initial survey to find how users of an application would perceive being logged. This was evaluated by presenting the logging component to employees at four different companies in Sweden, and afterwards handing out a survey. The total number of participants in the survey was 27, of which 16 were software developers, 8 worked in management, and 3 worked in other fields. No participant had less than 2 years of work experience, and nearly 50% had 10 or more years of work experience.

The participants were also asked about the developmental perspective. Over 90% of the participants said that they knew

 ISBN: 978-1-61208-458-9

of a project where the logging component would have been useful, and 100% of the participants thought that the logging component had potential to provide information that could aid in the further development of an application. Just under 90% of the participants thought that the logging component had potential to provide information that would facilitate the debugging of an application.

Three questions were asked to query how users would perceive being logged. The participants were first asked whether they would be comfortable using an application for their own private matters if they knew that it was being logged. As Figure 4 shows, 60% of the respondents said that they would not be comfortable with using a logged application for private matters. The participants were then asked whether they would be comfortable using an application at work if they knew that it was being logged. In this case, the results were very different. As Figure 5 shows, only 3 out of 27 participants – around 10% – answered that they would not be comfortable with this.

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

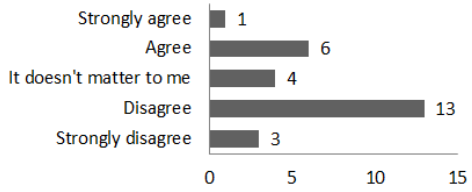

Figure 4. Comfort with using an logged application for private matters.

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

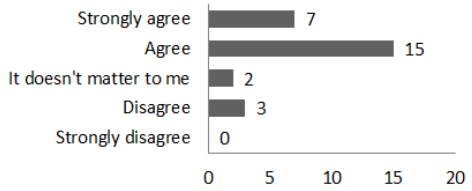

Figure 5. Comfort with using a logged application at work.

Finally, the participants were asked whether they would be more comfortable using an application that is being logged at work rather than one they use for private matters. 20 out of 27 answered that they would be more comfortable using a logged application at work than at home, while 7 participants answered that they felt that was no difference between the two. These results therefore suggest that the context in which the logging component is used can greatly affect the perception of users about whether logging is acceptable or not. Using a logged application at work, for instance, seems to be much more tolerable for most people rather than using a logged application at home.

VI. Conclusions

Continuous deployment, experiment systems and user-centered software engineering approaches have become very popular in modern software development. However, there are still challenges, such as how to add logging functionality to an application, what to log, and how to translate low-level logging data into knowledge about how product features are used by their users. In this paper, we contributed by developing and evaluating a logging component which could be integrated with a product without affecting its source code, and which added negligible performance penalties. It was found to provide developers with informative data on how a product is used by enabling them to define features and then visualize how they are used. Since this style of logging can lead to ethical issues, we conducted a survey with 27 participants at four different companies to initially assess the attitude of users to being logged. The results from the survey showed that users consider the logging process to be acceptable as long as they are informed of it, and if the logged application is not one that they use for private matters.

References

[1] M. Staron, W. Meding, and 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.

[2] B. Fitzgerald and K.-J. Stol, “Continuous software engineering and beyond: trends and challenges,” in Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. ACM, 2014, pp. 1–9.

[3] J. Bosch, Continuous Software Engineering. Springer, 2014.

[4] E. Lindgren and J. Münch, “Software development as an experiment system: A qualitative survey on the state of the practice,” in Agile Processes, in Software Engineering, and Extreme Programming. Springer, 2015, pp. 117–128.

[5] H. H. Olsson and J. Bosch, “Post-deployment data collection in software-intensive embedded products,” in Continuous Software Engineering. Springer, 2014, pp. 143–154.

[6] R. Veeraraghavan, G. Singh, K. Toyama, and D. Menon, “Kiosk usage measurement using a software logging tool,” in Information and Communication Technologies and Development, 2006. ICTD’06. International Conference on. IEEE, 2006, pp. 317–324.

[7] T. Menzies and T. Zimmermann, “Goldfish bowl panel: software development analytics,” in Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, pp. 1032–1033.

[8] A. Steiber and S. Alänge, “A corporate system for continuous innovation: the case of google inc.” European Journal of Innovation Management, vol. 16, no. 2, 2013, pp. 243–264.

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

[10] H. H. Olsson and J. Bosch, “The hypex model: From opinions to data-driven software development,” in Continuous Software Engineering. Springer, 2014, pp. 155–164.

[11] E. Borjesson and 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.

[12] R. Feldt, M. Staron, E. Hult, and T. Liljegren, “Supporting software decision meetings: Heatmaps for visualising test and code measurements,” in Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on. IEEE, 2013, pp. 62–69.

[13] V. Vaishnavi and W. Kuechler, “Design Research in Information Systems,” 2004.

[14] R. H. von Alan, S. T. March, J. Park, and S. Ram, “Design Science in Information Systems Research,” MIS quarterly, vol. 28, no. 1, 2004, pp. 75–105.

[15] P. Runeson and M. Höst, “Guidelines for Conducting and Reporting Case Study Research in Software Engineering,” Empirical software engineering, vol. 14, no. 2, 2009, pp. 131–164.

[16] Nicke Manarin. ScreenToGif. [Online]. Available: https://screentogif.codeplex.com (Retrieved: January 2016)

 ISBN: 978-1-61208-458-9