Designing an Unobtrusive Analytics Framework for Monitoring Java Applications

Sampo Suonsyrjä(✉) and Tommi Mikkonen

Department of Pervasive Computing, Tampere University of Technology, Korkeakoulunkatu 10, 33720 Tampere, Finland
{sampo.suonsyrja,tommi.mikkonen}@tut.fi

Abstract. In software development, attention has recently been placed on understanding users and their interactions with systems. User studies, practices such as A/B testing, and frameworks such as Google Analytics that gather data on production use have become common approaches in particular in the context of the Web, where it is easy to perform frequent updates as new needs emerge. However, when considering installable desktop applications, the situation gets more complex. While analytics facilities are still needed, they should address business logic, not generic traffic as is the case with many web sites. Moreover, analytics should be unobtrusive, and not have a high impact on the evolution of the actual application; thus, analytics should be treated as an add-on, as the target system may already exist. Finally, the instrumentation of features that are observed should be easy and flexible, but the provided mechanisms should be expressive enough for many use cases. In this paper, we examine different alternatives for implementing such monitoring mechanisms, and report results from an experiment with Vaadin, a web framework based on Java and Google Web Toolkit, GWT.

1 Introduction

The introduction of Agile methods [6] caused a paradigm shift in the development of software systems: instead of starting with a set of requirements that are all of the same value, software developers began to embrace a model where systems are first built with only a set of key features to be later extended into a more complete form. As more and more experience regarding the use of the system is gathered, developers write new versions of the system which satisfy user needs better. In fact, one can even claim that the core of iterative development is the ability to learn in each increment, which leads to improved products.

In the process of creating the software in the above fashion, input from users of the system can play a crucial role, given that adequate mechanisms for collecting the input are available. The most traditional way is to design questionnaires or other studies that the end users answer to guide the development, but in particular in the field of web systems, also more sophisticated forms of gathering information exist regarding users and the way the system is being used. For instance A/B testing, where different sets of users use a slightly different version of the software, helps in deciding between two ways to provide similar or the

© Springer International Publishing Switzerland 2015
A. Kobyliński et al. (Eds.): IWSM-Mensura 2015, LNBIP 230, pp. 160–175, 2015.
DOI: 10.1007/978-3-319-24285-9_11

same features. Moreover, analytics frameworks such as Google Analytics provide detailed understanding regarding how users interact with the system to perform more complicated tasks. In general, the ability to gather all this information is opening new possibilities for developers, because even the slightest deviations in user behavior can be tracked and reacted upon.

Although the field of web systems can nowadays be seen to have an edge in collecting post-deployment data, the same need is increasing in other contexts as well, as evidenced by [8]. In this paper, we investigate techniques for monitoring application-level user activity, as well as an option to extend the techniques to cover installable desktop applications, too.

The goal is to track actions at the level of user interface widgets, such as buttons, sliders, and text fields for instance. The work is based on using Java web framework Vaadin [5], where applications are first composed with Java, and then compiled into a form that can be deployed to the web, with the parts of the application that form the user interface being compiled with Google Web Toolkit [12]. As the concrete implementation mechanism for introducing analytics facilities, we experiment using aspect-oriented techniques [4] to bind an existing design to an external data analytics framework.

The rest of this paper is structured as follows. In Sect. 2, we introduce motivation and background of the study. In Sect. 3, we introduce our research questions. In Sect. 4, we describe our demonstrator application and how it has been constructed. In Sect. 5, we provide details of our implementation: showing how data is gathered in an unobtrusive fashion and describing the design of our analytics framework. In Sect. 6, we provide an extended discussion regarding our findings. Finally, in Sect. 7, we draw final conclusions.

2 Background

Analytics is used by businesses of all type to better understand customers. During the recent years, also software engineers and software engineering organizations have understood the opportunity to use more data for making constantly better decisions, but as even sporting teams have improved their performance with the help of analytics, the uses for analytics seem to be fairly general [1].

2.1 Software Analytics

Pachidi et al. [11] have developed the Usage Mining Method that enables conducting classification analyses, user profilings and clickstream analyses on logged operation data. Such data is beneficial for program understanding and reengineering [3]. In addition, as the size and complexity of software systems continue to grow, decision making is becoming even more difficult in the future and thus new solutions such as the use of analytics data are needed [2].

Kristjansson and van der Schuur have formulated the concept Software Operation Knowledge [9]. They describe that to consist of knowledge of in-the-field

performance, quality and usage of software, and knowledge of end-user experience and end-user feedback. The researchers continue with stating how software vendors have a great interest in acquiring such knowledge, but that the systematic practice of gathering, analyzing and acting on such knowledge is still limited. Correspondingly, this kind of in-the-field knowledge could benefit usability studies as the lack of long-term data collection is considered as one of the challenges in measuring usability [7].

In general, it is possible to collect usage metrics by executing software applications, but this usually requires some sort of modifications to the source code of the target application. There are a few exceptions however. For example, the Patina system [10] uses Microsoft Active Accessibility API to collect accessibility data, and thus no altering of the source code is needed. The system creates a so-called heatmap, which visualizes the content and location of the user interface controls visible in the application. As a drawback, supporting the accessibility API usually requires some extra work from the application developers and so the coverage of the accessibility API can vary.

As for concrete implementations, one of the most commonly used analysis frameworks is Google Analytics (http://www.google.com/analytics/), which is presently being used by an increasing number of web sites. With it, the developers of a web site can track traffic of a monitored web site and view it in a form that is easy to interpret. The data provides information regarding visitors, their geographical locations, the time they remain on the site, what is the path that users take on the web site, and so on. Since the system operates in the Web, its operation can rely on web protocols that reveal these properties. For a generic desktop application, however, these facilities are not immediately available. Moreover, when considering installable applications, data to be collected is often application specific, not web traffic related as is the case with Google Analytics. However, the popularity of Google Analytics demonstrates that there is an increasing interest regarding user data, which can be made available in an unobtrusive fashion.

2.2 Aspect-Oriented Programming

Aspect-oriented software development provides means for capturing cross-cutting concerns and modularizing them as manageable units [4]. Tackling the issue of tangled code, aspect-oriented programming languages such as AspectJ provide means to insert additional operations to a target program in an unobtrusive fashion with a new construct, so-called *aspect*. Aspects in turn provide increased opportunities for advanced modularity.

At the implementation level, an AspectJ aspect always includes at least two parts: a pointcut and an advice, both of which are code snippets. The pointcut is used to describe the point where the execution of the target program is paused for inserting the additional code programmed in the advice part. Figure 1 provides a simple aspect code that introduces a simple logging facility that records the parameters and the return value of a method call. In this aspect, the pointcut is defined to take effect around the defined function of our example class,

MyClass::MyFunc. The Logger aspect takes effect as the function is called, and the aspect code is executed both before and after actually executing the original method in a fashion where its execution is not affected. The operations that are being executed before and after running the method can be arbitrary; however for the purposes of software analytics, these include data collection operations.

```
aspect Logger {
pointcut loggedFunction = call("void MyClass::MyFunc(...)");
advice loggedFunction:around() {
   // Log call and method parameters
   tjp->Proceed(); // Run MyClass::MyFunc
   // Log results
   }
}
```

Fig. 1. A sample aspect.

3 Research Questions

The research questions we formed to evaluate our usage data collection and analysis framework are the following.

RQ1: To What Extent can a Data Collecting Feature be Implemented Without Compromising the Evolution of the Target Program? As a starting point for our research, we have taken a view where the design and evolution of the target system, in other words the program from which usage data is to be collected, must remain as independent from data collection and analysis as possible. High priority of this independence is motivated by the fact that in the end analytics data leads to changes in the target program. Therefore, it is crucial that the target program can be under constant change and these data can still be collected from it. This leads to the selection of implementation techniques that are as unobtrusive as possible.

As the evolution of the target program results in data being collected from different versions of the target program, the approach used for collecting data has to ensure that these data are still comparable between the different versions. Thus, not only do we want to find out specific types of data that can be collected with our framework, but also if the data is adequate enough to be compared between different versions of the target program. Finally, as we aim at designing a data collection framework that is independent of the underlying target program, we also introduce an option to reuse the development effort invested in the framework in different setups, including desktop applications as well as web systems built using Java.

RQ2: What Types of Data can be Collected with the Given Approach? As with any technology, there are restrictions regarding the data that can be collected. In this paper, we are interested in interactions between the user and the application, and therefore we focus on data that is associated with user interactions only. Thus, interactions with e.g. external actors or machines are beyond our scope in this paper.

RQ3: How to Connect the Data Collecting Feature with an Analysis Framework? Being able to record data from a user interface is only a beginning in the way towards understanding how an application is being used. Therefore, it is necessary to load the resulting usage data to an analysis system, which can then be used to further process the data into a meaningful form.

4 Demonstrator Application

To answer the above research questions, we next describe a demonstrator application. First, we introduce the platform on top of which the system is built. Then, we describe the application. Finally, we show how manual instrumentation could be carried out for this application.

4.1 Vaadin Web Framework

Vaadin [5] is an open source framework that is used for developing Rich Internet Applications (RIA). Vaadin applications are written using Java, and they are transformed into AJAX applications with the facilities of Google Web Toolkit (GWT) [12]. The architecture of the system is illustrated in Fig. 2.

Vaadin applications are implemented similarly to Java Standard Edition desktop applications, with all the functionality written using Java. However, instead of using the usual Java UI libraries like AWT, SWT, or Swing, a specific set of Vaadin UI components is used. These components can be compiled into a form that is runnable inside the browser, following the development process of GWT. This process is illustrated in Fig. 3. In addition, new custom made UI components can be implemented when needed to create systems with different kinds of look-and-feel.

4.2 Demonstration Application

To evaluate the designed framework for usage data collection, we selected a Vaadin application, which is fully functional and already developed yet simple enough to be the first test application. The source code is available for download at https://github.com/vaadin/dashboard-demo, and a working demo is located at http://demo.vaadin.com/dashboard.

The target application, called *QuickTickets Dashboard Demo*, demonstrates how the Vaadin framework can be used to create a simple dashboard web application. The main dashboard view is initialized as an object of DashboardView class.

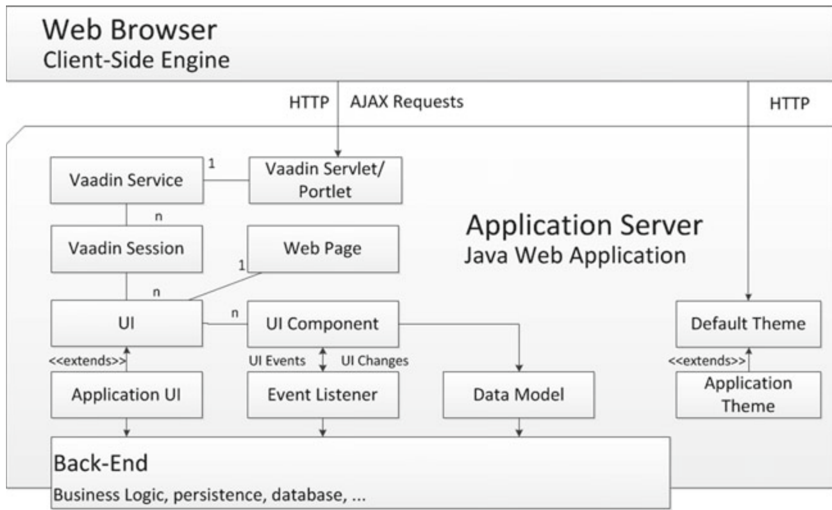

Fig. 2. Vaadin architecture. Image adapted from [5]

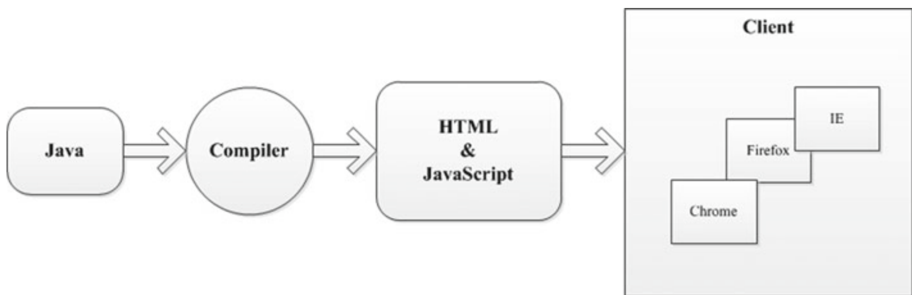

Fig. 3. GWT process of compiling Java to HTML and JavaScript [12]

During the initialization, several objects of HorizontalLayout class are instantiated and pushed to the view with an addComponent method. These include components such as a toolbar and several rows. Buttons are added correspondingly to these layout components in the same manner. In Fig. 4, we demonstrate the initialization of a dashboard object on code level along with a toolbar (a `horizontalLayout` object) and a notify button.

In the following, we demonstrate how collecting usage data works by focusing on buttons that can be pressed by users. While this obviously does not cover all the dimensions of software operation knowledge, this restriction simplifies the presentation to a form that is concrete enough to demonstrate at a detailed level how data collection works.

```
public DashboardView() {
   HorizontalLayout top = new HorizontalLayout();
   addComponent(top);
   Button notify = new Button('2');
   Notify.addClickListener(
      new ClickListener(){
         ...
      });
   top.addComponent(notify);
};
```

Fig. 4. Initialization of a dashboard object.

4.3 Manual Method as a Motivation

To show how the proposed automatic data collection feature simplifies developers' tasks, we first provide a manual implementation of the same function. To this end, we inserted data collecting features manually ourselves to specific places in the original source code of the target application. Thus, this approach is an intrusive one as it essentially changes the source code of the target application, which is built by someone else.

First, we developed a class called `DataLogger.java`. This class was used for two important tasks. On one hand, it included public method `logButtonClick` and on the other hand it stored these button clicks to a SQL type of a container. `Button` and `ClickEvent` objects were used as parameters for the method. It draws information about the button and its context and then stores it to the aforementioned temporary container. This information could be of course stored in some other way as well, but for our study case this was not seen important. However, some storing options are discussed in the future work section. This class itself was then included in the same java package with the target applications source code files. Up until this point the target applications source code was not altered.

In the unobtrusive part, the whole source code of the target application was then searched through to find each and every place where a new button was instantiated and added to the UI as seen in Fig. 4. As with every button there was also an instantiation of its `ClickListener`, we always inserted a call to our `logEvent` method within this instantiation. In Fig. 5, we provide a code snippet that elaborates how this implementation was done.

Clearly, we only used one intrusive insertion to the application, the call to method `Logger.logEvent`. However, even with this simple application, there were a total of 33 of this kind of button instantiations in the target application, all of which had to be extended with a similar call to our data logging method. While 33 insertions can be implemented once quite fast, the devil is in the complexity that most likely starts to build up when such implementation process is repeated for a while. Especially in a case where the target application is developed by a different person than the one implementing the usage data logging features, there

```
final Button signin = new Button("Sign In");
signin.addClickListener(
   new ClickListener() {
      public void buttonClick(ClickEvent e) {
         Logger.logEvent(signinEvent, e);
         ...
      }
   }
);
```

Fig. 5. Manual implementation of data collection.

is always the risk of forgetting to add these logging features to all the necessary places. Furthermore, even if a special script was developed to insert the logging features automatically to specific places, one would have to be very careful in developing such a script. Although this should reduce the risk of forgetting to log a button at all, any possible extra calls to the data logging methods would then again distort the data and its reliability as button clicks could be recorded not just once but twice or trice and so on.

In the regard of data comparability the manual approach, qualities depend greatly on the specific implementation. In this case study, our implementation gathered data only straight from the context of the target application. This included data types such as buttons caption, session id, and URI fragment. Although having all the data coming from the Vaadin frameworks context creates quite a reliable starting point for a further usage data analysis, target application evolution and changes in for example buttons captions might lead to inconsistencies in collected usage data.

In what comes to the flexibility of the manual approach and usefulness of the data it collects, we saw this approach performing understandably well. Making the application log new kinds of data types was as easy as making it log the first types of data. Of course in a case with a larger-scale application this might take more than a blink of an eye. However, the point in the flexibility criterion is to evaluate if the approach is able to collect also new kinds of data and the manual approach certainly has that as an advantage. Similarly, it collects just the types of data one wants and thus these data should be as useful as any.

5 Data Collection and Analysis Framework

To support usage data collection, we designed a framework where several already existing techniques and tools are used (Fig. 6). These key components are:

- AspectJ is used for creating an unobtrusive monitoring mechanism for the target application.
- Fluentd (www.fluentd.org) is used as the mechanism for unified data collection.

– Elasticsearch (www.elasticsearch.org/overview/elasticsearch) is used as a real-time storage for flexible searches.
– Kibana (www.elasticsearch.org/overview/kibana) is used for creating real-time visualizations and analytics.

This stack that combines Fluentd, Elasticsearch, and Kibana can be considered as an open source alternative to Splunk (www.splunk.com) log management software.

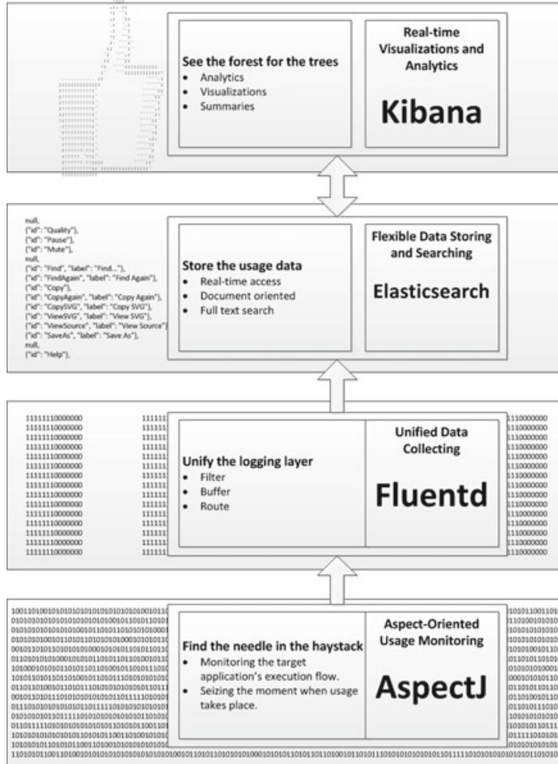

Fig. 6. The designed framework for unobtrusive analytics.

5.1 Aspect-Oriented Usage Monitoring

The aspect-oriented approach to inserting additional features into existing applications is unobtrusive by nature. As already mentioned, we demonstrate this facility by focusing on buttons. To this end, we wish to intervene in the execution every time a button is being added to a UI component (see Fig. 4 in Subsect. 4.2). To attach a pointcut and a logging advice to such call, aspect `AddComponentListener` was created as shown in Fig. 7.

In this aspect, pointcut called `addComponentCall` defines that each time method `addComponent` is called with a button as its parameter, the execution

```
public aspect AddComponentListener {
   // Button clicks are stored in this container.
   DataLogger dataCollector = new DataLogger();
   // To be executed when a button is added to the layout.
   pointcut addComponentCall(Button b):
      call(* *.addComponent(*)) && args(bb);
   // To be executed after a button has been added to layout.
   after(final Button b):
      addComponentCall(b) {
      // Clicks are listened to with a basic Vaadin ClickListener.
      b.addClickListener(
         new Button.ClickListener() {
         public void click(ClickEvent e) {
            dataCollector.logEvent(b, e);
         }
      });
   }
}
```

Fig. 7. Data collector aspect, its pointcut and advice.

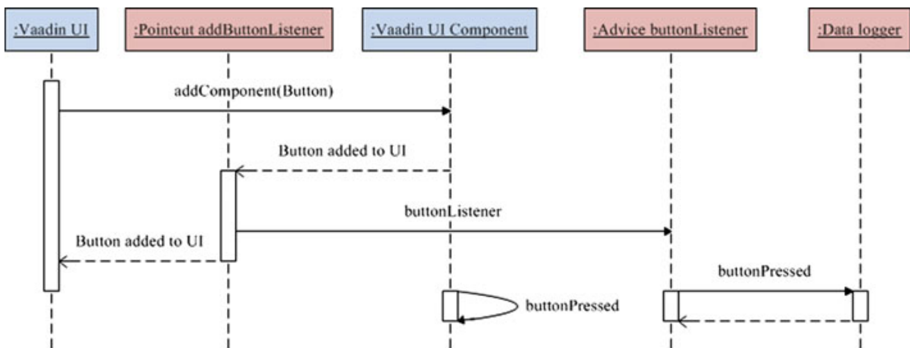

Fig. 8. Insertion of an additional click listener with an aspect.

can be cut for the corresponding advice part. This part will then define an additional click listener. This is shown in Fig. 8.

Finally, a remark must be made regarding the degree of unobtrusiveness of the approach. While the effect of AspectJ code is unobtrusive to the underlying target program, tooling is affected by AspectJ. To begin with, for the build process, a dependency to AspectJ must be inserted to the target application's project file. Additionally, the AspectJ tools must be included in the used IDE, in our case Eclipse.

5.2 Collecting Data with Fluentd

Fluentd was implemented in quite a similar fashion as the AspectJ for monitoring features. However, wherein AspectJ was used for unobtrusively monitoring the usage, Fluentd was used for collecting usage data from the usage points defined with AspectJ. Thus, the core idea of Fluentd is to be the unifying layer between different types of log inputs and outputs. This is illustrated in Fig. 9, in which a box is a component and the arrows describe the data flow.

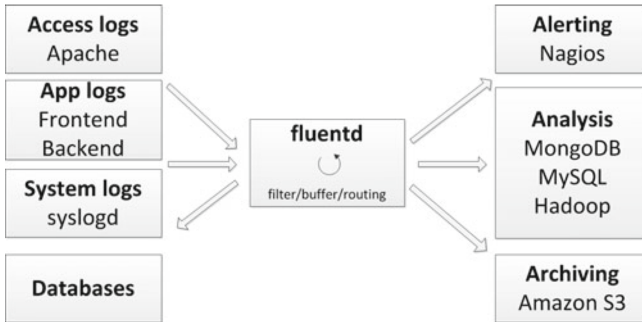

Fig. 9. Architecture of Fluentd and its plugins. Image adapted from [fluentd.org/architecture]

Figure 9 illustrates the architecture of Fluentd. Its various plugins for data input make it easier to unify the logging layer of an application or even an application ecosystem. There are a number of different input plugins available for several programming languages. In this study, we obviously used an input plugin for Java applications. However, Fluentd supports inputs not only from different language applications but also from entirely different kinds of inputs. These include for example access and error logs from web servers and system logs.

The concrete implementation of Fluentd into the target application required that a Fluentd dependency was inserted into the source code of the target application. This was done similarly as with the AspectJ facilities. Additionally, we installed and ran Fluentd on the same machine with the target application. As these requirements are met, the Fluentd process is able to receive the inputs described in Fig. 10. As described with the usage monitoring aspect in Fig. 7, `logButtonClick` method is called whenever a button is clicked.

Similar to the input plugins of Fluentd, its plugins for storing data standardize that front. Depending on the use case, data can be stored in different formats for archiving and analysis, for example. In this study, we used Fluentd for parsing the usage data into JSON and then forwarding them for analysis in Elasticsearch. As seen in Fig. 10, there were different types of usage data related to a button click, its context, and the button itself. These data were first stored in a temporary Java Hashmap object but then forwarded to Fluentd for its filtering, buffering, and rerouting processes.

```
public class DataLogger {
    private static FluentLogger LOG =
       FluentLogger.getLogger("button.click");
    public void logButtonClick(Button b, ClickEvent event){
       Map<String, Object> data = new HashMap<String, Object>();
       data.put("Uri Fragment", Page.getCurrent().getUriFragment());
       data.put("Page", Page.getCurrent().toString());
       data.put("Button Caption", b.getCaption());
       data.put("Button ID", b.getId());
       ...
       data.put("Click X", event.getClientX());
       data.put("Click Y", event.getClientY());
       LOG.log("click", data);
   }
}
```

Fig. 10. Collecting data from a Java application with Fluentd.

5.3 Elasticsearch and Kibana

In our study setup, we used Fluentd and Elasticsearch on the same localhost. Fluentd sent the collected usage data to Elasticsearch, which stored them into its document oriented database without any pre-configurations. As the data was already formatted in JSON, the field names were already there. This in combination with the full-text search abilities made analyzing facilities easily accessible. In addition, Elasticsearch supports real-time access to exploring the stored data.

However, Elasticsearch is only storing the data and making it searchable. Therefore, Kibana was used as a dashboard for displaying the data from Elasticsearch. Through this dashboard, one can make queries and then visualize the results in various different forms. An example visualization is shown in Fig. 11. In the visualization there is a pie chart illustrating how many times a specific button has been clicked.

6 Discussion

To discuss our findings, we next revisit our research questions one by one. In addition, we will also provide some directions for future research.

6.1 Research Questions Revisited

Based on our experiences with the proposed framework, we revisit the paper's questions as follows.

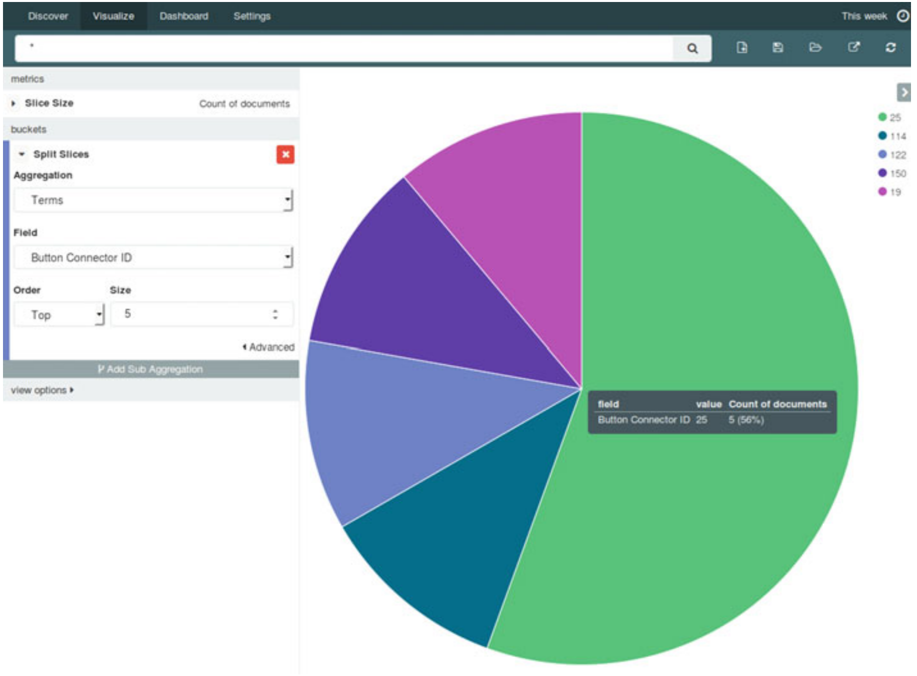

Fig. 11. Screenshot of a Kibana visualization.

RQ1: To What Extent can a Data Collecting Feature be Implemented Without Compromising the Evolution of the Target Program? Aspect-oriented approach to inserting additional features is quite unobtrusive by nature, which is supported by the code snippets. Also in this case, the usage monitoring facilities were inserted without changing the source code of the target application. The only parts which needed some modifications were the dependency addition to a build file and an insertion of an AspectJ file.

As these modifications were not altering the source code itself, the target application's evolution was not compromised nearly as much as with the manual approach. In this sense, if the target application's next version was to include new buttons, the aspect-oriented monitoring would notice them just as they did with all the rest. Therefore, the approach allows the target application to scale in that way without any additional efforts needed to include to the additional buttons as new data collecting points.

However, if the target application was to be changed in the way its buttons are instantiated, the aspect-oriented monitoring needs to be changed correspondingly. Even in this kind of a case though, the modification to the monitoring pointcut would most likely have to be done only once.

RQ2: What Types of Data can be Collected with the Given Approach? With an aspect-oriented monitoring approach, pointcuts could be made on a vast variety of different points in the execution flow. For instance, we could have associated the pointcuts with the initialization of objects of a particular class, as well as any other public method. The same goes for advices, which can contain almost arbitrary code that is needed for monitoring.

Additionally, aspect-oriented techniques support various different types of data that can be collected. Software operation knowledge in general includes information such as in-the-field performance, quality and usage of software, and knowledge of end-user experience, and end-user feedback, and to some extent this is necessarily platform-specific. In our case, the Vaadin framework provides an API to get such data directly from the platform. For instance, there are straightforward methods to get information on timestamps, URI fragments, button captions, and so on. With such information, it is possible to gain knowledge for example about the clickstream a user leaves behind, the average time they spent on a specific page, or what kind of errors are logged the most.

All in all, the aspect-oriented approach provides us with the same flexibility in gathering different types of data as the manual approach did. With such arbitrary data types, the problems of analytics are more about asking the right questions than getting enough data.

RQ3: How to Connect the Data Collecting Feature with an Analysis Framework? Although collecting data can be done in most cases in a various ways, further exploring and analyzing of data might turn out more difficult. The use of a standardized analysis framework might require the data to be in a specific format. In this regard, the data logging tool's ability to unify the data it collects becomes important. In this study, Fluentd was used for collecting data, and it also performed the unifying by turning the data into the JSON format. This again was a format that the data storing solution supported and the visualization tool had an access to. Thus, the data collection tool's unifying feature enabled us to form an end-to-end analytics framework starting from the usage monitoring and peaking in the visualizations.

In circumstances such as these, general collection frameworks can provide a way to standardize parts of the logging even if data inputs and outputs varied from time to time. This becomes especially important when the aim is to combine data from different kinds of sources such as access, error and application logs.

6.2 Future Work

The work reported in this paper is only the very beginning of research regarding using aspects as a tool for analyzing user interactions. As already pointed out, at present we have a mechanism for collecting the data, and next challenge is to figure out which part of the data is truly meaningful, and how should the gathered data be used. Some of the directions for future work are listed below.

Extending the Measurement Point Set. In addition to collecting straightforward data on user actions, broadening the focus to cover attributes such as in-the-field performance or end-user feedback can turn out as helpful opportunities for various different fields. For instance, a short user experience survey could be injected as an aspect into a specific point of execution flow, an error log could be sent to developers when a system crashes, or a sorry-note could be shown to the user in case of system performing under a specified level. Being able to perform this in a non-intrusive fashion could improve user experience considerably, with no risk to the future evolution of the system.

Experimenting with Real-Life Apps. Obviously, the feasibility of the above data collection approaches is domain dependent, and the type of the application as well as the setup created for testing has an impact on whether or not operations are offline or real time. Therefore, experimenting the different approaches with real-life applications and developer needs forms an important part of future work. Our present strategy is to execute these experiments together with Vaadin and the associated developer community. In addition, once we reach a maturity level where the analysis framework can be used in production use, we wish to study how interaction data that has been automatically collected relates to user studies executed in more conventional fashion.

7 Conclusions

Fueled by the opportunities provided by the web and associated tools, analytics regarding the use of software applications have become a central aspect in software development. The rationale is that data regarding the fashion a software system is used helps in understanding the true needs of end users. This in turn enables the design of more satisfying software applications, with improved performance, simplified interactions, and superior user experience. However, gathering data on real-life use of applications is sometimes difficult, in particular when considering installable applications that cannot be easily updated remotely. Moreover, creating practical tools for analysis commonly requires application specific attention.

In this paper, we are experimenting how analytics facilities similar to web applications can be introduced to desktop and Rich Internet Applications written in Java. To keep the application intact from analytics facilities, we are using AspectJ as the implementation technique for introducing application-level monitoring, which allows us to hook analytics facilities to user interface events in a non-intrusive fashion. As for analysis, we are using an already existing tool set, where open source systems play a key role. The implementation we have created is concise, and it can be easily generalized to other applications if needed.

Based on our experiences reported in this paper, we find aspects a technique that is well-suited for creating data extraction features for already existing applications. In particular, given that the applications follow certain conventions, it appears to be relatively straightforward to create join points that are easily

repeatable. Since we wish to track user actions, starting with user interface widgets is the natural starting point and almost all user interaction mechanisms in modern programs follow certain patterns, we believe that the results we have obtained can be generalized to many other environments, too. Moreover, already existing analysis tools provide support for filtering, analysing and visualizing data at real-time.

Acknowledgment. The authors wish to thank Digile Need4Speed program (http://www.n4s.fi/) for its support for this research.

References

1. Begel, A., Zimmermann, T.: Analyze this! 145 questions for data scientists in software engineering. In: Proceedings of the 36th International Conference on Software Engineering, pp. 12–23. ACM (2014)
2. Buse, R.P., Zimmermann, T.: Information needs for software development analytics. In: Proceedings of the 34th International Conference on Software Engineering, pp. 987–996. IEEE Press (2012)
3. El-Ramly, M., Stroulia, E.: Mining software usage data. In: Proceedings of 1st International Workshop on Mining Software Repositories (MSR 2004), pp. 64–68 (2004)
4. Filman, R., Elrad, T., Clarke, S.: Aspect-Oriented Software Development. Addison-Wesley Professional, Reading (2004)
5. Grönroos, M.: Book of Vaadin. Uniprint, Turku (2011)
6. Highsmith, J.: Agile Software Development Ecosystems. Addison-Wesley Longman Publishing Co. Inc., Boston (2002)
7. Hornbaek, K.: Current practice in measuring usability: challenges to usability studies and research. Int. J. Hum. Comput. Stud. **64**, 79–102 (2006)
8. Juergens, E., Feilkas, M., Herrmannsdoerfer, M., Deissenboeck, F., Vaas, R., Prommer, K.: Feature profiling for evolving systems. In: Proceedings of the 19th International Conference on Program Comprehension, pp. 171–180. IEEE (2011)
9. Kristjánsson, B., van der Schuur, H.: A Survey of Tools for Software Operation Knowledge Acquisition. Department of Information and Computing Sciences, Utrecht University, Technical report UU-CS-2009-028 (2009)
10. Matejka, J., Grossman, T., Fitzmaurice, G.: Patina: dynamic heatmaps for visualizing application usage. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 3227–3236. ACM, April 2013
11. Pachidi, S., Spruit, M., van de Weerd, I.: Understanding users behavior with software operation data mining. Comput. Hum. Behav. **30**, 583–594 (2014)
12. Perry, B.W.: Google Web Toolkit for Ajax. OReilly Short Cuts. OReilly (2007)