arXiv:1311.5270v1 [cs.SE] 21 Nov 2013

1

Automated Feature Identification in Web Applications

Sarunas Marciuska, Cigdem Gencel, and Pekka Abrahamsson

Free University of Bolzano-Bozen,

Summary. Market-driven software intensive product development companies have been more and more experiencing the problem of feature expansion over time. Product managers face the challenge of identifying and locating the high value features in an application and weeding out the ones of low value from the next releases. Currently, there are few methods and tools that deal with feature identification and they address the problem only partially. Therefore, there is an urgent need of methods and tools that would enable systematic feature reduction to resolve issues resulting from feature creep. This paper presents an approach and an associated tool to automate feature identification for web applications. For empirical validation, a multiple case study was conducted using three well known web applications: Youtube, Google and BBC. The results indicate that there is a good potential for automating feature identification in web applications.

Key words: feature creep, feature expansion, feature identification; feature reduction; feature location; feature monitoring; software bloat

1.1 Introduction

Feature creep [1, 2] (i.e. addition or expansion of features over time) has become a significant challenge for market-driven software intensive product development. Today's software intensive products are overloaded with features, which have led to an uncontrollable growth of size and complexity. A recent study [3] revealed that most of the software products contain from 30 to 50 percent of features that have no or marginal value.

One of the major consequences of feature creep is feature fatigue [4], when a product becomes too complex and has too many low value features. Users then usually switch to other, simpler products. Moreover, feature creep can also result in software bloat [5] that makes a computer application slower, which requires higher hardware capacities, and increases the cost of maintenance. One of the most recent example of software bloat is Nokia Symbian 60

smartphone platform [6]. The system grew so much that it was too expensive to maintain it, and therefore it was abandoned.

Currently, lean start-up [7] software business development methodology tackles the feature creep problem by finding a minimum viable product that contains only essential and the most valuable features. However, not all lean-start up companies start development from scratch and can easily determine the minimum viable product as they already have complex systems. For example, by understanding how users are using the features, a company might discover that a set of features maintained by the company are actually not too much valuable for their customers. Thus, decision makers could analyse if removal of such features would bring any long term benefits for the company as there would be less features to maintain. Therefore, there is a need to monitor and identify the features that are not too valuable in order to systematically remove them from the product [6].

To start with the feature reduction process it is crucial to identify the complete set of features. Features should be identified automatically in order to reduce feature reduction process cost. After identifying the features, they can be monitored by the company to detect how their values change in time. For example, feature usage monitoring could indicate that the usage of some features is decreasing, and thus such features might become candidates for feature reduction. In our previous work, we showed that low feature usage is a good indicator for the features that are potentially losing customer value [8].

There exists some approaches that tackle with feature monitoring and identification problem (see Section 1.2). For example, a number of methods aim to locate features from the source code [9], but they still lack precision. Others aim to monitor system changes by observing user activity [10, 11]. However, such approaches collect too much irrelevant information (i.e. random mouse clicks, mouse scrolling, all key strokes) and fail to monitor the system on a feature level. Another set of approaches [12, 13] monitor system usage on too high level of abstraction (i.e. page usage, but not a feature level usage).

The focus of this paper is to address the problem of automated feature identification for web applications for feature reduction purposes. We set our research questions as follows:

- RQ1 : What constitutes *a feature* in web applications for the purposes of feature reduction?
- RQ2 : How well features could be identified in an automated manner in the context of web applications?

Here, we present our approach and an associated tool that identifies elements of a web application (based on HTML5 technologies) that correspond to features. To this end, first, we investigated definitions of *a feature* by making a literature survey. Then, we defined formally what constitutes a feature in web applications. Finally, we developed a tool, which implements the rules for automatically identifying features in web applications.

To evaluate the performance of our tool, we conducted a multiple case study. We selected three well known web sites as the cases: Google, BBC, and Youtube. The features for these web applications were first identified manually by the participants of the case study and then in an automated way using our tool. At the end, we compared the results using two operational measures: precision and recall [14].

The rest of the paper is structured as follows: Section 1.2 presents the related work. In Section 1.3, we elaborate on definitions of feature in the literature. Then, by formalising the definition of a feature for web applications, we describe our approach for feature identification. Section 1.4 presents the case study and the results. In Section 1.5, we discuss threats to validity for this study. Finally, Section 1.6 concludes the work and presents future directions.

1.2 Related Work

The feature location field aims to locate features and their dependencies in the code of a software system. A recent systematic literature review on feature location [9] categorizes the existing techniques in four groups: static, dynamic, textual, and historical.

Static feature location techniques [15, 16, 17] use static analysis to locate features and their dependencies from the source code. The results present detailed information such as variable, class, method names and relations between them. The main advantage of these approaches is that they do not require executing the system in order to collect the information. However, they require to have an access to the source code. Moreover, static analysis generate a set of features dependent on the source code, so they involve a lot of noise (i.e. variable names that do not represent features).

Dynamic feature location techniques [18, 19, 20] use dynamic analysis to locate the features during runtime. As an input this technique requires a set of features, which has to be mapped to source code elements of the system (i.e. variables, methods, classes). As a result, a dependency graph among given features is generated. The main advantage of these techniques is that it shows the parts of the code called during the execution time. However, dynamic feature location techniques rely on the user predefined initial feature set, so they cannot generate a complete features set beforehand.

Textual feature location techniques [21, 22, 23, 24, 25] examine the textual parts of the code to locate features. As an input this technique requires to define a query with feature descriptions. Later, the method uses information retrieval and language processing techniques to check the variables, classes, method names, and comments to locate them. The main advantage of these techniques is that they map features to code. However, like dynamic feature location technique, it requires a predefined feature set with their descriptions.

Historical feature location techniques [26, 27] use information from software repositories to locate features. The idea is to query features from comments, and then associate them to the lines that were changed in respective code commit. The main advantage of these techniques is that they can map features to a very low granularity of the source code, that is to exact lines. However, as in dynamic and textual approaches, this technique cannot determine a complete features set in an automated manner. In the next section, we present our approach to address this issue.

1.3 A Method for Automated Feature Identification in Web Applications

There are a number of definitions in literature for what constitutes a feature. Below, we elaborate on some of these definitions and then present our formal definition for a *feature* for web applications.

1.3.1 Feature Definitions

The definition of 'a feature' varies widely depending on the area and purpose of the study. Classen et al. [28] made a detailed analysis on different definitions of a feature in the contexts of requirements engineering, software product lines and feature oriented software development. Below there are some of the definitions from this study:

- "A feature represents an aspect valuable to the customer" [29].
- "A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems" [30].
- "Features are distinctively identifiable functional abstractions that must be implemented, tested, delivered, and maintained" [31].
- "A logical unit of behaviour specified by a set of functional and non-functional requirements" [32].
- "A product characteristic from user or customer views, which essentially consists of a cohesive set of individual requirements" [33].
- "An elaboration or augmentation of an entity(s) that introduces a new service, capability or relationship" [34].
- "An increment in product functionality" [35].
- "A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option" [36].

Classen et al. [28] claimed that there is a need to have a definition, which covers all kinds of requirements, domain properties and specifications. Thus, they provided their own definition as: "A feature is a triplet, f = (R, W, S), where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification" [28].

However, most of the aforementioned definitions are either very generic or vague in order to be used for automatic feature identification, because they rely on subjective human evaluation (features may be interpreted differently by different people). For example, the aforementioned feature definition by Classen et al. [28] can identify different features for the same system depending on how the requirements specification document is written.

To the best of our knowledge, only one definition provided by Eisenbarth et al. [37] removes the subjective human element. It has been cited by most of the work done in feature location area. The authors define a feature as "a realized functional requirement of a system (i.e. is an observable unit of behaviour of a system triggered by the user)".

This definition makes it clear that 1) features are identified based on events triggered by users, and 2) they realise functional requirements. For example, a case where a user has to enter his email, password and press the login button in order to login is different from the case where system remembers his credentials and he just has to press the login button in order to login. In the first scenario, three features are identified, while in the second, only one even though the final state was the same (that is, the user logged into the system).

These two scenarios might be interpreted differently by different people depending on the abstraction level how they perceive what a feature is. In this study, we focus on the lowest granularity level features in order to be able to identify them automatically. Then, our approach allows decision makers to group similar features to represent them at higher granularity levels (see [38] for details).

Moreover, according to the definition non-functional requirements (such as performance requirements) are not features, but they might affect how features are implemented. In addition, some of them might evolve into functional requirements (such as usability requirements) during implementation, but then these would be identified based on the events triggered by users.

In this study, we extended the definition to include the features that are triggered not only by users, but by other systems as well (i.e. web services), since in some cases they can also be considered as users. We used this version of the definition when conducting our case study to evaluate the performance of the tool in automatically identifying features against manual identification.

1.3.2 A Formal Definition for a Feature in Web Applications

In the context of web applications, features that relate to functional requirements, are visible for system users through a web browser. This means that to identify features, it is not necessary to analyse a complex server side implementation of a system, which can be developed using different programming languages (i.e. Java, PHP, C). Therefore, in this study we focus only on the client side analysis. Currently, vast majority of the client side web based applications are designed using HTML5, JavaScript, and CSS technologies.

According to our feature definition, there are limited possibilities how a feature could be implemented using the aforementioned technologies. Therefore, a feature in a web application is: 1) a specific HTML5 element, which changes system's state that can be observed when event is triggered; 2) any HTML5 element, which has attached event that changes system's state in observable way.

In HMTL5 there are only three elements that can be considered features without having any event attached to them [39]. The elements are called ancor, input, and textarea. In addition, input element should not be of a type "hidden", because in such a way it used as a variable and is not a feature. The remaining elements become features if one of the four event types is attached to them: a mouse event (see Table 1.1), a keyboard event (see Table 1.2), a frame and object event (see Table 1.3), or a form event (see Table 1.4).

Table 1.1. Mouse Events

Event	**Description**
onclick	Event occurs when the user clicks on an element
ondblclick	Event occurs when the user double-clicks on an element
onmousedown	Event occurs when a user presses a mouse button over an element
onmousemove	Event occurs when the pointer is moving while it is over an element
onmouseover	Event occurs when the pointer is moved onto an element
onmouseout	Event occurs when a user moves the mouse pointer out of an element
onmouseup	Event occurs when a user releases a mouse button over an element

Table 1.2. Keyboard Events

Event	**Description**
onkeydown	Event occurs when the user is pressing a key
onkeypress	Event occurs when the user presses a key
onkeyup	Event occurs when the user releases a key

1.3.3 An Algorithm for Automated Feature Identification

We designed an algorithm to identify features in web applications. The main idea of the algorithm is to parse the Document Object Model (DOM) and to select all HTML5 elements that correspond to features, or all HTML5 elements that have one of the four aforementioned events assigned to them. The pseudo code of the algorithm is given below (see Algorithm 1). The

Table 1.3. Frame and Object Events

Event	Description
onabort	Event occurs when an image is stopped from loading before completely loaded (for object)
onerror	Event occurs when an image does not load properly
onload	Event occurs when a document, frameset, or object has been loaded
onresize	Event occurs when a document view is resized
onscroll	Event occurs when a document view is scrolled
onunload	Event occurs once a page has unloaded (for body and frameset)

Table 1.4. Form Events

Event	Description
onblur	Event occurs when a form element loses focus
onchange	Event occurs when the content of a form element, the selection, or the checked state have changed (for input, select, and textarea)
onfocus	Event occurs when an element gets focus (for label, input, select, textarea, and button)
onreset	Event occurs when a form is reset
onselect	Event occurs when a user selects some text (for input and textarea)
onsubmit	Event occurs when a form is submitted

algorithm identifies features on a low level of abstraction, which later can be manually increased by grouping features using Feature Usage Diagram [38].

Algorithm 1 A Complete Feature Set Identification

1: $result_set \leftarrow \emptyset$
2: $event_set \leftarrow \{all\ HTML5\ events\}$
3: $feature_set \leftarrow \{all\ HTML5\ features\}$
4: $DOM \leftarrow \{DOM\ of\ interest\}$
5: $element \leftarrow DOM.first$
6: **while** $element \neq \emptyset$ **do**
7:   **if** $element\ IN\ feature_set$ **then**
8:     $result_set \leftarrow element$
9:   **else if** $element.events\ IN\ event_set$ **then**
10:     $result_set \leftarrow element$
11:   **end if**
12:   $element \leftarrow DOM.next$
13: **end while**
14: **return** $result_set$

Algorithm 1 requires three input parameters: HTML5 event set of interest ($event_set$), HTML5 element set that represent features ($feature_set$), and the DOM of a website of interest (DOM).

The algorithm iterates through all elements of DOM and checks whether a selected element ($element$) is one of the elements from $feature_set$, or it has one of the events from the $event_set$ attached. If it is the case the $element$ is added to a result set ($result_set$), which is returned after loop iterations are finished.

The runtime complexity of the algorithm is $O(n(1+m)) = O(n^2)$. In takes $O(n)$ times to iterate over n elements in DOM. Assuming that $feature_set$ and $event_set$ are implemented using hash table, then to check if $feature_set$ contains $element$ takes $O(1)$ time. Therefore, to iterate over all $element.events$ and to check if they are in $event_set$ it takes $O(m)$ time.

We implemented this algorithm using the JavaScript programming language. The code is available as the external JavaScript library on the following website: `http://www.featurereduction.org`. Initially, it has a $feature_set$ predefined, which contains ancor, input, and textarea elements. Since the vast majority of the websites are built using just few events (i.e. onclick, onkeystroke), the tool allows user to select custom $event_set$ from a complete list of HTML5 events: a mouse event (see Table 1.1), a keyboard event (see Table 1.2), a frame and object event (see Table 1.3), or a form event (see Table 1.4). Then, a custom JavaScript library is generated, which contains the selected event set.

There are two ways to use this tool when identifying the set of features for a web application of interest: 1) put the aforementioned library directly on a website where it is hosted; 2) download the website and include the library in the downloaded version of it. Obviously, the latter approach is able to identify features only in the downloaded pages. It would not work on the pages that are dynamically generated or cannot be downloaded. Nevertheless, in a normal use case scenario, companies have a full access to their websites and can apply the first method.

Finally, the following information is presented as the result:

1. the full path to an element,
2. the title attribute of the element,
3. the name attribute of the element,
4. the id attribute of the element,
5. the value attribute of the element,
6. the text field of the element.

The full path to the element helps to find it in the DOM. We assumed that the title, name, id, value attributes and text field, if present, usually contain human understandable information, which contains a description of the feature. We test this assumption in our case study, which we present in the next section.

1.4 Case Study

We conducted a multiple-case study according to [40] to evaluate whether our tool performs well in identifying the features in web applications with respect to manual identification.

We selected three web applications for the case study: Google, BBC, and Youtube. The selected websites cover a wide range of daily used applications having search, news and video streaming features. One major reason why we chose these websites was that they are used in different contexts, and thereby, we could test our tool to identify variety of features developed for different contexts. Another reason was that most readers would be familiar with these applications as they are widely used (according to the Alexa traffic rating [41] they are among top 10 most popular websites in Great Britain). Finally, as these websites are not customised based on user demographics, the set of features would be the same for all participants of the case study and hence the results could be comparable.

In addition to the first author of this paper, 9 subjects, who are frequent users of the case applications, participated in this study. This is a convenience sample, where the participants have varying backgrounds (i.e. medicine, design, computer science) and sufficient knowledge and experience in using web applications.

1.4.1 Case Study Conduct

Before the case study, we downloaded the main web pages of the case websites to make sure that all participants have the same version of the website. After that, we distributed the downloaded pages as executable systems to the participants, introduced the participants our formal feature definition and then asked them to manually identify the features. The participants were given as much time as they need to complete the task.

As our purpose in this case study was to identify manually all the features of the selected applications in order to compare them to the results of our automated tool, we needed to have a correct and compete set of features that were identified manually. Therefore, after the set of features identified by the participants converged to a one final set, we considered this as the complete set and we stopped asking more people to participate in the case study. In parallel, the first author of this paper also identified the set of features in the same way as the participants to cross-check the features identified by the participants.

Later, we asked the participants to write down the information about features in a text file. Finally, we used our tool to identify the features from the downloaded web pages. The results from the tool were printed out to the console of a web browser.

To evaluate the performance of our tool we used operational measures: *precision* and *recall*. *Precision* indicates whether the tool collects elements

that are not features and *recall* indicates whether the tool determines the complete features set.

As a pre-analysis, we verified with all the participants each feature, which they identified and recorded. We excluded the data provided by two of the participants as they provided too generalized outputs (i.e. "menu widget features"). They mentioned that it was too much time consuming task. Therefore, they could not provide a complete feature set.

To compute the precision and recall, we compared features identified by the participants and the tool to the complete feature set. In addition, we compared the data collected by the tool with the feature description provided by the participants.

Then, we made informal interviews with the participants to receive feedback on the features, which were identified by them, but not identified by the tool, and vice versa.

1.4.2 Results and Analysis

The results showed that there were both *visible features* and *hidden features* in the applications of the case study. Visible features are the ones that participants were expected to detect manually through a web browser without using other tools or looking at the code of a given website. Google had 33, Youtube had 80 and BBC had 96 visible features in total.

The analysis of the results showed that the precision was 100% in both manual and automatic identification of visible features. This means that both the tool and the participants could identify the elements that correspond to features. Figure 1.1 presents the results of the recall measure.

The recall measure to automatically identify visible features by the tool was 100% for the Google website, 91% for the Youtube website, and 100% for the BBC website. When we investigated why our tool could not identify all the features in the Youtube website, we saw that this is due to Flash technologies used in this website. This showed that our tool has some technological limitations when identifying features.

When we compared the performance of our tool to manual identification of the features, we saw that our tool overperforms the participants in most of the cases. We checked with the subjects why they failed to identify all the visible features. Here are the reasons we found after our interviews:

- *Missed.* Some of the features were simply missed by the participants due to carelessness in manual identification. It was understood that the great majority of such features were not visible instantly in the main page as they were placed in drop down, or pull down lists.
- *Redundant.* The participants reported that there were some features that had the same functionality as the ones, which they had already identified. Therefore, they didn't add these features in the list. For example, the link

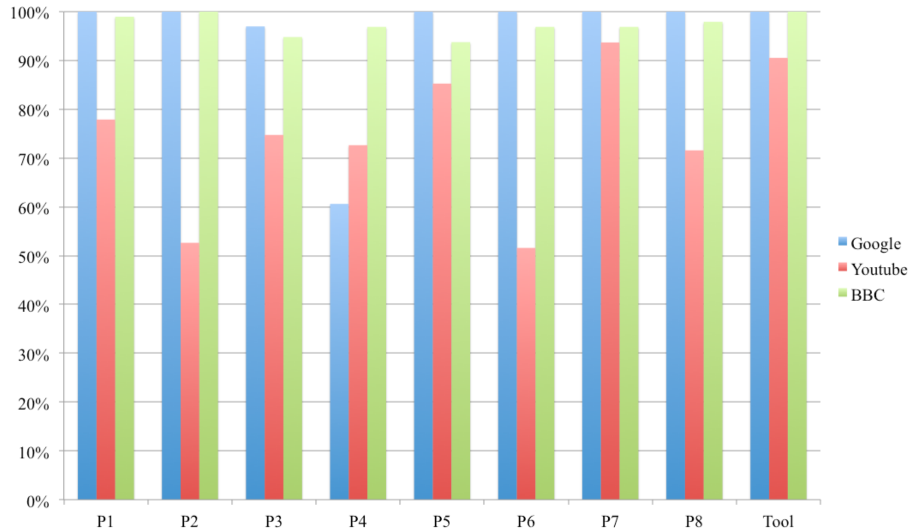

Fig. 1.1. Case Study Results – Recall

on the icon of the commentator and the link on the name of the commentator in Youtube leads to the same page. Therefore, some participants identified only one feature instead of two in this case.

Finally, we compared the representation of the results provided by our tool and the results provided by the participants. We noticed that most of the participants used the name of a feature from a website to describe its functionality. After analyzing the tool results, we found out that this information was stored in text, value, or title attributes. The rest of the attributes (id attribute and name attribute) were not useful for this task. To describe the location of a feature participants indicated the main container where a selected feature belongs to. On the other hand, the full path of the feature provides even more detailed information as the one provided by the participants.

One major result of the case study was that our tool could identify a high number of *hidden features*, which could not be detected by the participants manually. In the Google website, our tool identified 15, in the Youtube website 8, and in the BBC website 126 hidden features.

After analyzing the source code of such features, we found out that most of these were related to personal user preferences, or the device that is used to open the website. For example, if a user opens a website using a mobile phone or tablet, then the feature set is adapted accordingly. In addition, there were a few features, which did not seem to add any value (such as 3 hidden textarea elements in the main google website that are not necessary). Those features might be important for the developers of the website for some reason, or might be just obsolete features, which hang in the code without any specific

purpose. This shows that out tool has the potential to indicate different kind of features, which could be removed after developers assess them.

1.5 Threats to Validity

We discuss the validity threats of this study according to the categorization suggested by Runeson and Host for case studies [42]: 1) Construct validity, 2) Internal validity, 3) External validity, and 4) Reliability.

Construct Validity.

Construct validity refers to what extent the operational measures represent what is investigated according to the research questions. One validity threat could have been related to our definition for feature for web applications as well as how it is interpreted. Therefore, we first built our definition upon one of the widely cited definition in the feature location research area [37]. We also kept the granularity level of a feature at the lowest level so that it would be possible to group related features if one needs to represent them at higher levels.

Furthermore, we formalised our definition in order to be able to implement an algorithm to automate feature identification. This in turn also helped in avoiding subjective misinterpretation of the definition by the participants when they were identifying the features manually. One of the operational measures used in this study was precision. As the precision values for both manual and automatic identification of visible features were 100%, we also conclude that all elements identified as features with the tool comply with our definition.

Another possible validity threat could have been due to making a mistake in identifying a complete feature set as the second operational measure; recall, was calculated based on this figure. To mitigate this threat, first, one of the authors of this study manually identified the features. Then, we observed whether the manually identified features converge to the one final set as we receive input from more participants. If participants identified new features, which were not present in the initial complete feature set, then we extended it with those features. We stopped involving more subjects in a manual feature identification when we saw that adding more participants was not adding any new information. Therefore, we believe that this set should reflect the complete set.

Internal Validity.

Internal validity concerns the causality relation between the treatment and the outcome, and whether the results do follow from the data. In this study, we evaluated the performance of our method and tool in comparison to manual

identification of features. One validity threat could have been if the participants did not have enough knowledge about selected websites to perform the task well. To mitigate this task we chose the participants, who were existing users of the selected websites. Furthermore, we purposefully chose the case applications that are frequently used. In this way, we believe that the subjects had sufficient knowledge about the features of the websites. In fact, all participants showed similar performance for all three cases.

External Validity.

External validity refers to what extent it is possible to generalize the findings to different or similar contexts. We designed the case study as a multiple-case study where we used three different web applications. Therefore, we could evaluate the method and the tool developed in this study for a variety of features coming from different web applications. However, one threat, which we could not totally avoid, might have occurred due the technology used when developing the case applications. We observed that our tool has some technological limitations in identifying the features. There is a possibility that the results could have been affected if other case applications, which use very different technologies, had been selected. However, we believe that our results can be generalizable to some extent, as we replicated the study using three different web applications developed for different contexts using different technologies. Still, there is a need to test the method and the tool for very different web applications. Furthermore, we could not evaluate our approach for the cases where decision makers prefer to group features and represent them at higher granularity levels. Our results are also not generalizable for other systems, which are not web applications (i.e. desktop applications). The goal and the scope of this study was to design an approach to automatically identify features in web applications. Therefore, there is a need to define what a feature constitutes for other application types. Then, algorithms and/or rules could be created to identify a complete feature set.

Reliability.

Reliability reflects to what extent the data and the analysis depend on the specific researchers. We used two objective operational measures in this study: precision and recall. Therefore, we do not see any validity threat in interpreting them. However, one validity threat could have been the interpretation of the feature set descriptions provided by the participants. To mitigate this treat, we verified with each participant what we understood from their inputs.

1.6 Conclusion and Future Work

In this paper, we presented an approach and an associated tool for automating features identification in web applications. The results of the case study

showed that our approach has a significant potential to identify features in web applications.

Furthermore, we found out that our tool is also able to identify hidden features in addition to visible ones, which could not be identified manually by users. Some of these features appeared to be with no significant value, that is they might be candidates to be removed from the system. Therefore, we see this method as a first step towards automatic feature reduction process.

Moreover, the results of the case study showed that manual feature identification is prone to human mistakes even for websites with a relatively small number of features. Therefore, we conclude that development of such a tool provides more benefits when used to identify features in big systems with complex structures and high number of features.

As a future work, we plan to investigate ways to capture features implemented by other technologies such as Flash. We will analyze if similar approach can be applied in desktop applications. Furthermore, we will explore the ways how a complete feature set can be visualized. We plan to investigate how the relations between features can be detected automatically.

The automatic feature identification is the first step towards automatic feature usage monitoring. We plan to explore how such information can be used to improve product. For example, it would be interesting to understand if relocation of features can increase the usage and bring more value for the company.

Acknowledgment

The authors would like to thank the participants of the case study who provided their precious time and effort.

References

1. B. Elliott, Anything is possible: Managing feature creep in an innovation rich environment, In Engineering Management Conference, pp. 304-307, IEEE, 2007.
2. F. D. Davis and V. Venkatesh, Toward preprototype user acceptance testing of new information systems: implications for software project management, IEEE Transactions on Engineering Management 51(1), 2004.
3. C. Ebert and R. Dumke, Software Measurement, Springer, 2007.
4. R. T. Rust, D. V. Thompson, and R. W. Hamilton, Defeating feature fatigue, In Harvard Business Review, 84 (2), pp. 98-107, 2006.
5. G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications, In Proceedings of the FSE/SDP workshop on Future of software engineering research, 2010.
6. C. Ebert, P. Abrahamsson, and N. Oza, Lean Software Development, In IEEE Software, pp. 22-25, Oct. 2012.

7. E. Ries, The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses, Journal of Product Innovation Management, 2011.
8. S. Marciuska, C. Gencel, and P. Abrahamsson: Exploring How Feature Usage Relates to Customer Perceived Value: A Case Study in a Startup Company. In: 4th International Conference on Software Business. pp. 166177, 2013.
9. B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, Feature Location in Source Code: A Taxonomy and Survey, In Journal of Software Maintenance and Evolution: Research and Practice, 2011.
10. R. Atterer, M. Wnuk, and A. Schmidt, Knowing the user's every move: user activity tracking for website usability evaluation and implicit interaction, In Proceedings of the International Conference on World WideWeb, pp. 203, 2006.
11. Microsoft Spy++, url: http://msdn.microsoft.com/en-us/library/aa264396(v=vs.60).aspx, Last visited on the 31st of March 2013.
12. OpenSpan Desktop Analytics, url: http://www.openspan.com/products/desktop_analytics, Last visited on the 31st of March 2013.
13. Google Analytics, url: http://www.google.com/analytics, Last visited on the 31st of March 2013.
14. C. D. Manning, P. Raghavan, and H. Schutze: Introduction to information retrieval. Cambridge: Cambridge University Press, 2008.
15. K. Chen and V. Rajlich, Case Study of Feature Location Using Dependence Graph, In Proceedings of 8th IEEE International Workshop on Program Comprehension, pp. 241-249, 2000.
16. M. P. Robillard and G. C. Murphy, Concern Graphs: Finding and describing concerns using structural program dependencies, In Proceedings of International conference on software engineering, pp. 406-416, 2002.
17. M. Trifu, Using Dataflow Information for Concern Identification in Object-Oriented Software Systems, In Proceedings of European Conference on Software Maintenance and Reengineering, pp. 193-202, 2008.
18. A. D. Eisenberg and De K. Volder, Dynamic Feature Traces: Finding Features in Unfamiliar Code, In Proceedings of 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, pp. 337-346, 2005.
19. J. Bohnet, S. Voigt, and J. Dollner, Locating and Understanding Features of Complex Software Systems by Synchronizing Time, Collaboration and Code-Focused Views on Execution Traces, In Proceedings of 16th IEEE International Conference on Program Comprehension, pp. 268-271, 2008.
20. D. Edwards, N. Wilde, S. Simmons, and E. Golden, Instrumenting Time-Sensitive Software for Feature Location, In Proceedings of International Conference on Program Comprehension, pp. 130-137, 2009.
21. M. Petrenko, V. Rajlich, and R. Vanciu, Partial Domain Comprehension in Software Evolution and Maintenance, In International Conference on Program Comprehension, 2008.
22. A. Marcus, A. Sergeyev, V. Rajlich, and J. Maletic, An Information Retrieval Approach to Concept Location in Source Code, In Proceedings of 11th IEEE Working Conference on Reverse Engineering, pp. 214-223, 2004.
23. S. Grant, J. R. Cordy, and D. B. Skillicorn, Automated Concept Location Using Independent Component Analysis, In Proceedings of 15th Working Conference on Reverse Engineering, pp. 138-142, 2008.

24. E. Hill, L. Pollock, and K. V. Shanker, Automatically Capturing Source Code Context of NL-Queries for Software Maintenance and Reuse, In Proceedings of 31st IEEE/ACM International Conference on Software Engineering, 2009.
25. D. Poshyvanyk and A. Marcus, Combining formal concept analysis with information retrieval for concept location in source code, In Program Comprehension, pp. 37-48, 2007.
26. A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail, CVSSearch: searching through source code using CVS comments, In Proceedings of IEEE International Conference on Software Maintenance, pp. 364-373, 2001.
27. S. Ratanotayanon, H. J. Choi, and S. E. Sim, Using Transitive changesets to Support Feature Location, In Proceedings of 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 341-344, 2010.
28. A. Classen, P. Heymans, and P. Y. Schobbens, What's in a feature: A requirements engineering perspective, Fundamental Approaches to Software Engineering, pp. 16-30, 2008.
29. M. Riebisch, Towards a more precise definition of feature models, Modelling Variability for Object-Oriented Product Lines, pp. 64-76, 2003.
30. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, Feature-oriented domain analysis (FODA) feasibility study, Carnegie-Mellon University, Pittsburgh, Software Engineering Institute, 1990.
31. K. C. Kang, Feature-oriented development of applications for a domain, In Software Reuse, 1998. Proceedings, pp. 354-355, IEEE, 1998.
32. J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, Addison-Wesley Professional, 2000.
33. K. Chen, W. Zhang, H. Zhao, and H. Mei, An approach to constructing feature models based on requirements clustering, In Requirements Engineering, 2005. Proceedings, pp. 31-40, IEEE, 2005.
34. D. Batory, Feature modularity for product-lines, Tutorial at: OOPSLA, 6, 2006.
35. D. Batory, D. Benavides, and A. Ruiz-Cortes, Automated analysis of feature models: challenges ahead, Communications of the ACM, 49(12), pp. 45-47, 2006.
36. S. Apel, C. Lengauer, D. Batory, B. Moller, and C. Kastner, An algebra for feature-oriented software development, Number MIP-0706. University of Passau, 2007.
37. T. Eisenbarth, R. Koschke, and D. Simon, Locating Features in Source Code, In IEEE Computer, 29(3):210, Mar. 2003.
38. S. Marciuska, C. Gencel, X. Wang, and P. Abrahamsson: Feature Usage Diagram for Feature Reduction. In: Agile Processes in Software Engineering and Extreme Programming. pp. 223-237, Springer Berlin Heidelberg, 2013.
39. HTML Reference, `http://www.w3schools.com/tags/default.asp`, Last visited on the 31st of March 2013.
40. R. K. Yin, Case study research: Design and methods, SAGE Publications, Incorporated, 2002.
41. Alexa traffic rating, `http://www.alexa.com/topsites/countries/GB`, Last visited on the 31st of March 2013.
42. P. Runeson and M. Host, Guidelines for conducting and reporting case study research in software engineering, In Empirical Software Engineering, pp. 131-164, 2009.