Feature Profiling for Evolving Systems

Elmar Juergens, Martin Feilkas,
Markus Herrmannsdoerfer, Florian Deissenboeck
Technische Universität München
Garching b. München, Germany

Rudolf Vaas, Karl-Heinz Prommer
Munich Re Group
München, Germany

***Abstract*—Traditionally, most work in program comprehension focuses on understanding the inner workings of software systems. However, for many software maintenance tasks, not only a sound understanding of a system's implementation but also comprehensive and accurate information about the way users actually use a system's features is of crucial importance. Such information *e. g.*, helps to determine the impact that a specific change has on the users of a system. In practice, however, this information is often not available. We propose an approach called *feature profiling* as a means to efficiently gather usage information to support maintenance tasks that affect the user interface of a software system. Furthermore, we present tool support for feature profiling and report on a case study in the insurance domain. In this study, we profiled the features of an application that is used by 150 users in 10 countries over a period of five months.**

***Keywords*-Usage Analysis, Software Maintenance**

I. Introduction

Software systems create value by effectively and efficiently supporting their users' tasks. As the tasks change over time, software systems must be continuously evolved to reflect these changes. Consequently, more than 50% of the maintenance activities carried out over the lifetime of a software system are of *perfective* nature, *i. e.*, they modify existing functionality or add new features [1], [2].

Traditionally, the program comprehension research community has mostly taken the role of a programmer that is faced with implementing a change request specified by requirement engineers. The programmer's tasks are to understand the change requests, locate the system's source code that needs to be modified or extended and implement the change requests. While these tasks are doubtlessly central, we argue that program comprehension could gain from taking a more holistic view that does not exclusively focus on the system itself but also takes its context into account. In particular, we believe that software maintenance would benefit from comprehensive and accurate information about the way users actually use a software system's features. Such information enables software engineers to make more informed decisions w.r.t. crucial questions like:

- What is the impact of proposed change requests for the system's users?
- Are there features that are not used at all and could potentially be removed to save maintenance efforts?
- What are the most important features to which test efforts should be dedicated [3]?

One might argue that requirements engineering should be responsible for supporting software maintenance with such data. Experience, however, shows that requirements engineering often focuses on the initial version of a system and does not consistently follow its evolution. Moreover, requirements engineering usually lacks the tools to obtain the required information in a reliable manner.

In this paper, we build on the techniques developed in program comprehension (and related) communities and propose *feature profiling*, a dynamic analysis technique, to obtain precise and complete usage data of a system's features. We present the feature profiling approach, the tools required to implement it as well as a comprehensive case study in the domain of business information systems that demonstrates how our approach can be applied in practice. Custom software in this domain offers unique opportunities for feature profiling, since its users are typically better known and accessible than users of off-the-shelf software or embedded systems. However, from our experience gained in several maintenance projects in the German insurance industry, the amount and accuracy of usage information available to software engineers is typically very limited.

Problem. Although many software maintenance tasks could benefit from comprehensive and accurate information about the usage context of a software system, it is often not available to software engineers.

Contribution. We propose an approach called feature profiling, based on dynamic analysis of deployed software, as a means to efficiently gather such comprehensive and accurate usage information for business information systems. We list the primary challenges of feature profiling and present solutions. Additionally, we introduce tool support for feature profiling and report on a large scale case study in the insurance domain. In this case study, we profiled an application that supports about 150 experts in 10 countries over a period of 5 month and found, among others, multiple features that are not used at all. Our approach protects the users' privacy by anonymizing the collected usage data.

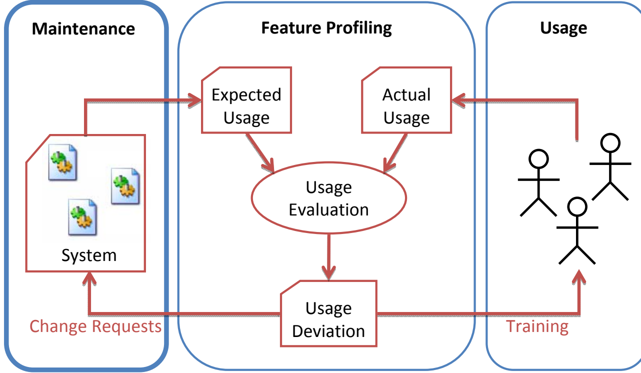

Figure 1. A control loop for feature profiling

II. Terms & Definitions

We define a *feature* according to Eisenbarth et al. [4] as a realized functional requirement that has observable behavior and can be triggered by a user. Features thus represent the functionality *offered* by a system. The functionality to print a document in a text processor, or to put an item into the shopping cart in a web shop, are examples of features. They can—but need not—match the requirements. Both system users and engineers know features. They can thus provide a foundation for communication between different stakeholders. *Feature groups* collect features of related functionality.

Features are implemented in the source code of a system. From a dynamic analysis perspective, source code can be regarded as a collection of methods[1]. One method can contribute to the implementation of several features. We call a method *characteristic* for a feature, if it is executed *always* when the feature is executed, but *never* when other features are executed. A single feature can have none, a single or multiple characteristic methods, depending on the system structure. A *feature beacon* is a characteristic method that has been selected manually from the set of all characteristic methods for the feature.

III. Feature Profiling

We propose to perform feature profiling in order to measure and evaluate the usage of a system's features. Figure 1 illustrates the general control loop for feature profiling. Feature profiling advocates to specify the expected usage of the features of an application (in a feature model) and to continuously check whether the actual usage conforms to these expectations. In case the usage evaluation uncovers significant deviations, there are two possibilities to react: on the one hand, the deviations may be caused by features that do not support the business processes (anymore) the way they should. A possible reaction is a modification of the features of the system. On the other hand, usage deviations may be caused by the a lack of user familiarity with certain features. Thus, training the users could be an appropriate reaction. In summary, feature profiling helps the developers to understand the usage of the features and to make informed decisions when maintaining the system based on their usage.

Our approach for feature profiling consists of four tasks. *1) Feature modeling:* The central artifact that is needed for feature profiling is a feature model. The feature model contains basic information about all the features that are implemented in a software system as well as their expected usage. *2) Feature location:* In order to measure the actual usage of a system in production, a mapping from the features and the code is needed. This mapping is specified by identifying feature beacon methods (*c.f.*, section II). *3) Execution monitoring:* Given a set of feature beacons, execution monitoring can be performed by installing a method profiler in the production environment to monitor the invocations of these methods. Due to the definition of the feature beacons, these measurements reveal the number of feature invocations. *4) Usage evaluation:* Based on the feature invocations, we can determine the deviations between actual and expected usage of the features. The following sections explain feature profiling in detail.

A. Feature Modeling

We use a feature model to capture the functionality of a system. Features are modeled as a tree, using feature groups as inner nodes. Each feature comprises a unique name and a textual description. A feature model for the software RSSOwl[2] is shown in Figure 2. Use cases and user manuals are a good starting point for identifying features and structuring the feature model. From our experience, the organization of the user interface can guide feature model construction. This has the advantage that the stakeholders are already familiar with this structure and the used terms.

We specify the expected usage for every feature in terms of a time interval. This interval represents the amount of time after which it would be astonishing if the feature would never have been executed by any user. As business information systems are usually used during workdays, weekends and public holidays are not taken into account. Some features might be intended to only be used very infrequently—*e. g.*, balance sheet generation is probably only executed once in a year. Other features are only intended to be used under special circumstances that only appear sporadically. For them, "never" is a suitable expectation. Fail-over features are an example for such functionality, as they are never executed if no failures occur.

B. Feature Location

The identification of feature beacons is a lightweight way of performing feature location. To a certain degree, develop-

[1] We do not differentiate between *methods* and *functions*. For simplicity, we only use the term *method*.

[2] For nondisclosure reasons, we use a feature model of the open source application RSSOwl instead of the application of Munich Re that we analyze in the case study in Section V.

ers typically have an estimation about which methods might be beacons for individual features. However, we need a structured technique to identify and evaluate feature beacon methods. We propose the following steps:

1. Feature tracing: Execute every feature in the feature model and record the set of methods that are executed. Repeat the execution several times with different arguments for each feature. As a result we gain a set of methods which have been invoked during the different feature executions.

2. Identification of characteristic methods using feature inference: The set of characteristic methods for a feature can be determined by what we call *feature inference*. Feature inference is an algorithm for selecting the methods that have been invoked in all executions of a certain feature and that do not appear in any execution trace of any other feature in the feature model.

3. Feature beacon selection: Select one appropriate feature beacon for each feature from the set of its characteristic methods. The developers should be involved in choosing the feature beacons. The choice is influenced by the identifiers of the characteristic methods. Methods with feature-specific identifiers should be preferred over general identifiers. Furthermore, the location of the methods within the structural architecture of the system also influences the choice. From our experience, in GUI-based applications, event handlers for buttons or menus as well as feature-specific services in a business logic layer are a good choice.

The result of feature location are feature beacons. Depending on the architecture of the system, there may be features without a single beacon candidate. Thus, these features cannot be monitored without modifying the code.

C. Execution Monitoring

Having feature beacons for the features of the application, the usage of the features can be measured by monitoring the invocations of the beacon methods. In general, this task can be achieved by using instrumentation or profiling techniques. However, as execution monitoring is applied in a production environment, it is a crucial requirement that the monitoring technique applied does not have a negative impact on the system's performance.

The information about the invocations of the methods gets written to a file or database as the output of the execution monitoring. By examining if this data contains invocations of the beacon methods, the actual usage of the features in the system is determined. This usage data is collected in regular time intervals and made accessible to compare it to the expected usage.

D. Usage Evaluation

In order to reason about the usage of the system, feature profiling puts the feature model and the collected usage data together. The measured actual usage can now be compared with the expected usage to identify deviations. Furthermore, changes of the usage behavior over time can be analyzed in order to identify changes in the business processes. The analysis results can be employed in different ways.

First, features that are not used at all might be removed from the implementation of the software system, thereby reducing code size and thus maintenance effort (if they are not fail-over features or similar). This lowers the overall lifecycle costs of the system.

Second, the results can be used to adapt the features implemented in a software system to increase its customer value. The usage data makes the actual usage of the system more transparent to the developers and enables more substantiated discussions with the stakeholders about the direction of the future evolution of the system.

Third, the actual usage can be used to prioritize change requests. Developers can use the usage data as an indicator for the importance of change requests that target certain features and prioritize their tasks accordingly. Thus, important changes are introduced earlier into the system.

Fourth, the usage data can be used as an operational profile (*c.f.*, [5]) for reliability engineering [6]. Knowing that features are used very frequently indicates that bugs in the corresponding code regions have serious impact on reliability. Thus, also the planning of testing and quality assurance activities should be guided by the data about the actual usage of the system.

IV. Tool Support

We have implemented tool support for each of the four phases explained in Section III. In the following, we outline the tooling for each phase.

A. Feature Modeling

Feature modeling is supported by the *feature model editor*, as depicted in Figure 2. It is implemented based on a metamodel using the Eclipse Modeling Framework (EMF) [7] and provides multiple views to edit a feature model.

The tree view on the left shows the structure of a feature model. Leafs represent features; inner nodes represent feature groups. The tree view provides standard functionality to modify the feature model structure: creation, deletion, drag & drop, and copy & paste of model elements.

The properties view at the top right corner allows the user to modify the properties of a model element. The user can set the name and description for each model element as well as the expected usage for each feature (*c.f.*, Section III-A).

The problems view at the bottom right corner lists the violations of the validation rules built into the feature model editor. An example of a validation rule is that the name of a feature must be unique within a feature group. The validation rules produce problems with different severities indicated by different icons. The problems are also shown in the tree and properties view as icons decorating the model elements and properties where the problems exist.

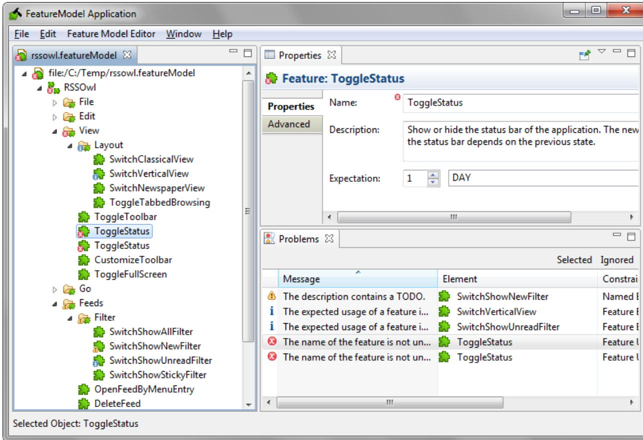

Figure 2. Feature model editor

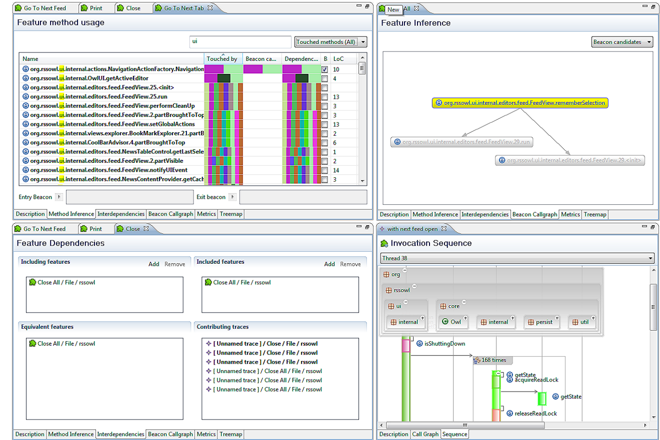

Figure 3. Feature beacon selection

B. Feature Location

Feature location requires tools to record method traces for feature executions and to select the beacons for features.

To record method traces for feature executions, we have implemented a *tracer* for the .NET virtual machine using its profiling API[3]. This way, the code of the traced software system does not need to be modified. Similar techniques can be applied for other virtual machines, like *e. g.*, the Java or ABAP virtual machine. The tracer records entry and exit events of each invoked method by using its fully qualified name. As a consequence, the tracer has a high impact on the performance of the system during tracing. However, this kind of tracing is only required to locate the features and not to monitor the execution of the system in production. Moreover, the tracer allows to exclude methods from recording. This way, methods from 3rd party code can be excluded to reduce the memory footprint.

The tracer is integrated seamlessly into the feature model editor. The user can start the tracer for a certain feature, and can then execute the feature in the traced software system. The method trace recorded during the feature execution is linked directly to the feature in the feature model. After all feature executions have been recorded, the editor can automatically infer the characteristic methods for each feature. From them, the user selects a beacon for each feature.

Figure 3 shows different views of the feature model editor that support the user in selecting feature beacons. The view at the top left corner shows the methods contributing to a feature and indicates whether they are characteristic for the feature or to which other features they contribute. If there are multiple characteristic methods, the view at the top right corner can display the call dependencies between them to support the user in choosing a method that has been called early during the execution of the feature. If there are no characteristic methods, the view at the bottom left corner can determine other features with which the feature can be merged to obtain characteristic methods. Finally, the view at the bottom right corner allows the user to inspect the method traces as sequence diagrams.

[3]http://msdn.microsoft.com/en-us/magazine/cc301725.aspx

C. Execution Monitoring

On the one hand, the impact on performance needs to be as low as possible, when monitoring the execution of the software system in production. On the other hand, we have to ensure that we do not miss a method invocation, since usage data could otherwise be incomplete.

We implemented a very efficient method profiler that employs *ephemeral* [8] profiling. The method profiler determines which methods have been executed. When the software system is started, each method is instrumented with a profiling hook that registers the invocation of the method. When the method is invoked for the first time, the hook is automatically removed from the method. As a consequence, the hook has no impact on the later invocations of the method, thereby significantly reducing the overall profiling impact. In addition, due to its implementation as an in-process component, the profiler does not require expensive context switches. While we implemented the profiler based on the .NET profiling API, the approach is not .NET specific. Its principle is also applicable to other platforms that use virtual machines, such as Java.

Using this technique, we can measure whether a feature is used within a certain time interval, but not how often it is used. However, this information is enough for usage evaluation, as long as the time interval is short enough. We typically set it to a day. Therefore, the software system needs to be restarted each day to ensure that its methods are instrumented again. When the software system is stopped, the profiler stores the usage data in a file.

The profiler might miss invocations of methods that are inlined due to optimizations performed by the just-in-time (JIT) compiler. To alleviate this issue, the hook instruments the method also for the inlining event. We are safe to interpret the inlining of a method as an invocation, since the

inlining is performed just-in-time by the virtual machine, and we are only interested in the first invocation anyway. In addition, feature location also traces method inlining. Thus, feature beacon inference takes into account during which feature executions which methods are inlined.

To further reduce the performance impact, we could limit instrumentation to the methods that are feature beacons. However, we instrument all methods due to the following reasons. First, we are more robust w.r.t. changes of feature beacons due to software evolution. Second, we can already start monitoring, even if we do not have beacons for all features yet. Third, we do not miss feature executions, in case we made an error when inferring a feature beacon. Similar to the tracer, we can again exclude methods that we do not want to monitor.

D. Usage Evaluation

The results of the usage evaluation can be shown in the feature model editor. It can import method profiler data and determine feature executions using the feature beacons. As shown in Figure 4, the actual usage can be represented in a table where the rows represent the features and the columns represent the days. The feature model editor also compares the actual with the expected usage. The comparison is implemented as a validation rule, and deviations are indicated by warnings. Thereby, the user can easily spot deviations for further analysis. Another validation rule highlights the features that are not used at all. The aggregated actual usage data can be exported to an Excel file for further analysis outside the feature model editor.

V. Case Study

This section presents the industrial case study we performed to evaluate feature profiling.

A. Research Questions

We investigated 4 research questions to better understand the benefits and limitations of feature profiling:

RQ1: How well does beacon inference work in practice? Feature beacons are required to mine feature information from method profiling data. Whether feature beacons exist, and how difficult they are to discover, depends on the structure of the analyzed system. The practical suitability of beacon inference thus needs to be validated empirically.

RQ2: Do different software engineers have a consistent expectation of feature usage? Feature profiling compares expected against actual usage. This assumes that a common understanding of the expected usage exists among software engineers. However, different roles could cause usage expectations to vary. We need to understand usage expectation consistency to determine how to produce models of expected usage for feature profiling.

RQ3: Do expected and actual usage differ? If stakeholders already have an accurate understanding of the usage of a system, we can simply ask them and do not need to invest effort into usage measurement as we propose it. The difference between expected and actual usage is thus an indicator for the importance of feature profiling as opposed to simply interviewing stakeholders. Furthermore, it determines the potential usefulness of feature profiling to help align requirements and features.

RQ4: How does the profiler impact the analyzed system? Feature profiling needs to be performed in production to produce meaningful data. This is only possible, if the feature profiler has no noticeable impact on the usage of the application, both in terms of correctness and performance. This research question thus determines the feasibility of applying the feature profiler in a production context.

Table I
Study object

Language	Age (years)	Size (kLOC)	Engineers (max)
C#	8	360	9 (16)

B. Study Object

We evaluated research questions RQ1, RQ2 and RQ3 on a business information system at Munich Re Group. Munich Re Group is one of the largest reinsurance companies in the world and employs more than 47,000 people in over 50 locations. For their insurance business, they develop a variety of individual supporting software systems.

The analyzed business information system implements damage prediction functionality and supports about 150 expert users in over 10 countries spread over Europe, North and South America, China, Australia and Africa. An overview is shown in Table I.

We chose this system as study object for several reasons. First, the system has been in successful use for 8 years and is still actively used and maintained. Improvements in feature alignment are thus likely to improve usage value or decrease maintenance costs. Second, the development and usage context is typical for Munich Re Group. Its users are distributed across different countries. The software engineers are from different companies (some are employed by Munich Re, some by software suppliers) and work in different buildings. This distribution of users and engineers complicates communication inside and across the stakeholder groups and could thus allow a lack of alignment to remain unnoticed. Third, it is a web application. Its server offers a single point for usage data collection[4].

C. Suitability of Feature Inference (RQ1)

We analyze how well feature beacon inference can establish a link between features and source code in practice.

[4]Feature profiling is not limited to server-based applications, however. We are currently working on a solution that collects and aggregates usage data from all clients to create complete profiles and preserve user privacy.

Study Design. We modeled a subset of the features of the application. The subset comprises the three feature groups that were accessible on our machine. It covers the majority of the application functionality. For each feature, we produced traces. Afterwards, we inferred feature-characteristic methods to determine beacon candidates. From them, the beacon was manually selected for each feature. To determine practical suitability, we computed the ratio of features for which beacons could be determined and recorded the problems encountered during beacon inference.

Execution. We modeled all 111 features in the three feature groups. We manually executed each feature and used the tracer to record the executed methods. If a feature could be executed in different fashions (*e. g.*, via a menu or via a button), we produced several traces, one for each.

Results. We encountered several challenges during beacon inference. *Implementation overlap:* some features are implemented in the same set of methods. Which feature is executed depends on the values of the method parameters. No beacons can be determined for such features. *Feature interaction:* some features call other features. The called features thus do not have any characteristic methods that can be used for beacons. *Trace volume:* full method tracing produces large amounts of data. For features that involve heavy computation, tracing was infeasible, since the application timed out.

Beacon inference is complicated by the semi-decidability of whether a method is characteristic for a feature. Methods may appear characteristic for a given set of traces, but turn out not to be, as more traces become available. Such methods were excluded from the results through manual inspection by the developers.

Due to these problems, we only succeeded to infer beacons for 76 out of 111 features. For the remaining ones, the system would need to be modified. To avoid inaccuracies due to incomplete traces, it needs to be complemented by manual inspections of the discovered beacons. The applicability of feature beacon inference in practice is thus limited by the structure of the system and the available effort.

D. Consistency of Usage Expectation (RQ2)

We determine how consistent the usage expectations of different stakeholders are for the modeled features.

Study Design. We asked project participants for their usage expectation for each modeled feature. Each participant was interviewed independently. We chose participants from different roles to capture different engineering perspectives. Actual usage information produced by feature profiling was not made available to them to not bias their expectations. On the results, we computed Cohen's Kappa—a measure for inter-rater agreement—to capture expectation consistency.

Execution. Three project participants took part in the experiment: a product manager, a Munich Re-internal software maintainer and an external software maintainer. All participants had been working on the project for over one year. Each participant recorded his usage expectation, *c. f.*, Section III-A for each of the 76 features. Answers were chosen from a drop-down list from these values: "1 day", "2 days", "3 days", "4 days", "5 days", "2 weeks", "3 weeks", "1 month", "2 months", "3 months", "6 months", "9 months", "1 year", "2 years". The participants only took workdays into account.

Results. Only for 8 of the 76 features did the three participants have identical usage expectations. Computation of Cohen's Kappa yielded 0.21, indicating weak agreement[5].

This measurement treats slight and significant deviations uniformly. If, *e. g.*, two participants select "2 days", and one selects "3 days", we do not treat this differently than if the third participant had rated "2 years". However, intuitively, the agreement in the former case is substantially stronger than in the latter. To reflect this, we aggregated the expectations into the ranges "$\leq$ one week", "between one week and one month", "between one month and half a year" and "over half a year". For these ranges, Cohen's Kappa yields 0.59. While better, this still indicates a certain amount of disagreement between the stakeholders.

In summary, while there is a certain amount of agreement between different stakeholders, their usage expectancy is not consistent. We thus cannot expect the expectancy of a single engineer to reflect the expectancy of other stakeholders well.

E. Actual versus Expected Usage (RQ3)

We investigate how well the usage expectations of the different stakeholders match the *actual* usage of the system.

Study Design. We performed ephemeral method profiling on the production server. This way, the methods executed by *all* users of the system are recorded for the study period. However, we did not collect any user-specific information (both for privacy and performance reasons). To determine feature usage, we mapped the feature beacons against the called methods to determine which features were executed on which day. Finally, we compared the actual usage against the usage expectation of each of the three participants.

Execution. We installed the profiler on the server that hosts the analyzed system. The server recycled its application pools every night. This triggered the profiler to write out its report and reset its state. Feature profiling was performed over a period of five months. After one week, we analyzed the reports and validated their plausibility with the system stakeholders to ensure result validity.

Analysis of the method execution reports had to cope with feature beacon evolution. During the analysis period, three consecutive versions of the software were in production.

[5]It also indicates that the agreement is—statistically significantly—higher than random assignments. See [9] for explanation of the Kappa measure.

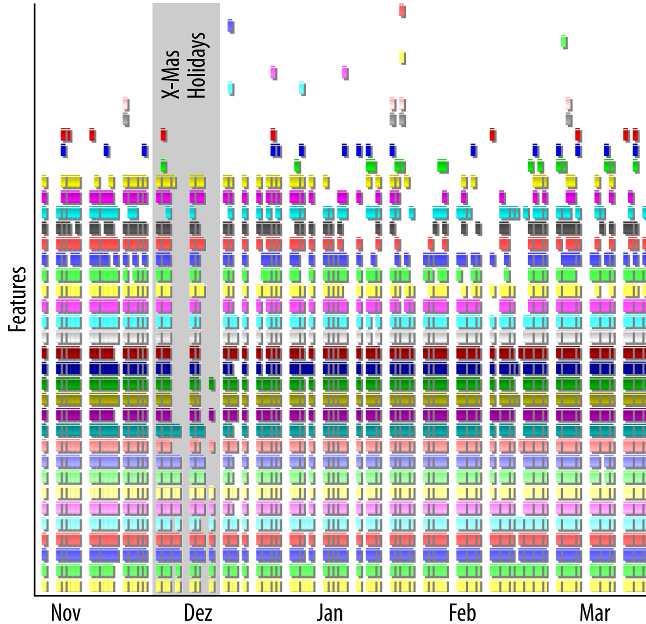

Figure 4. Actual feature usage

Feature beacon inference was performed on the last version. However, during software evolution, methods—including feature beacons—change. Some of the identified beacons did not exist in all three software versions. Of the 76 feature beacons, 53 (70%) existed in all three software versions. We limited the analysis of feature usage to these.

Comparison of actual and expected usage was performed as follows. For each feature, we computed the longest period (in workdays) in which it had not been used. If this period was more than twice, or less than half, of the usage expectation of the feature, we considered it a deviation. We computed the deviations between the features and the usage expectations of each of the three study participants. In addition, we determined how many features were not used at all during the study period.

Results. Figure 4 depicts the actual usage of the features during the study period. Each feature is depicted as a colored horizontal bar. For each day, the feature bar fragment is only visible, if the feature was executed on that day. Features are ordered by execution frequency: seldom used features at the top, often used ones at the bottom. Weekends and Christmas exhibit low feature usage. Of the 53 profiled features, only 38 were used during the five months of study. The remaining 15 features (28%) were not used at all.

The actual usage deviated from the usage expectation of the product manager for 43% of the features. For the internal and external developer, it deviated in 40% and 55%, respectively. In the majority of the deviations, the system was used *less* than expected. The Munich Re-internal stakeholders had a more accurate usage expectation (deviations of 43% and 40%) than the external developer (55%). However, deviations for all three of them are substantial. We thus answer RQ3 positively: for the analyzed system, actual usage does deviate from expected usage.

For the 15 unused features, usage expectations are depicted in Table II. For 5 features, it was unexpected for all three engineers that they were not used. For 6 further features, it was unexpected for at least one engineer. Only for 4 features did the engineers agree that the expected usage interval is larger than the study period.

F. Feature Profiling Impact (RQ4)

Since the profiler performs load-time instrumentation of the code, it can, in principle, skew execution results. Moreover, since it monitors jitting and inlining operations, it can slow down execution speed. In this study, we analyzed impact on both correctness and performance.

Study Design. To evaluate its impact on correctness, we executed a unit test suite with and without the profiler and checked whether it influenced the results. For performance impact evaluation, we executed a .NET performance benchmark with and without the profiler and compared execution times. Third, we installed the profiler on the machine that was used for testing. It was thus part of both the system and load tests done during maintenance.

Execution. We used the unit test suite of the Mono C# compiler[6], which is itself written in C#, to evaluate correctness impact. We chose it for its size ($>$ 3500 unit tests) and its coverage of all .NET language constructs. It exercises a large part of the profiling API and we thus assume that it has a high probability to uncover errors in the profiler. For performance evaluation, we used benchmarks from the Zorn CLI[7] Benchmark suite[8]. *Ahc* is a memory-intensive compiler benchmark, *Lcs* a compute-intensive compression benchmark. Each exists in three versions that operate on increasingly large data (and thus have increasing runtimes). For each, we took the best result from 10 runs to reduce influence of background tasks.

Results. All unit tests produced the same results with and without the profiler. We thus did not detect any impact of the profiler on correctness. The results of the performance benchmark are depicted in Table III. The impact of the profiler on performance is measurable, but very small. Importantly, it does not grow with bigger input. This confirms the effectiveness of ephemeral profiling.

Finally, during system testing of the study object that is performed during every maintenance iteration, the test engineers did not notice any impact of the profiler on the test results. In summary, the profiler does not impact

[6]http://www.mono-project.com/

[7]CLI stands for "Common Language Infrastructure" and refers to the .NET virtual machine and its platform technology.

[8]http://research.microsoft.com/en-us/um/people/zorn/benchmarks/default.htm

Table II
USAGE EXPECTATIONS OF UNUSED FEATURES

#Feat.	Product Manager	Internal Dev.	External Dev.
5	$\leq$ 3 months	$\leq$ 3 months	$\leq$ 3 months
2	$\geq$ 6 months	$\leq$ 3 months	$\leq$ 3 months
4	$\geq$ 6 months	$\leq$ 3 months	$\geq$ 6 months
4	$\geq$ 6 months	$\geq$ 6 months	$\geq$ 6 months

Table III
PERFORMANCE BENCHMARK RESULTS

	ahc1	ahc2	ahc3	lcs1	lcs2	lcs3
No profiler	0.7s	3.3s	7.5s	2.1s	8.3s	18.6s
With profiler	0.8s	3.4s	7.5s	2.4s	8.6s	18.8s
Δ	0.1s	0.1s	0.0s	0.3s	0.3s	0.2s

correctness, and its performance impact is small enough for use in production, even in computation and memory intensive applications.

G. Threats to Validity

The biggest threat to the generalizability of our study is its limitation to a single system. While both the technical aspects of the system and the development context are representative of the systems at Munich Re, the transferability of, *e. g.*, the usage deviation results to other systems inside and outside Munich Re is unknown. Future work is required to foster our empirical understanding of deviations of actual and expected usage. However, the study demonstrates the practical applicability of our approach for feature profiling and the usefulness of its results.

Our approach to compare actual and expected usage uses thresholds. More specifically, we consider expected usage to deviate from actual usage, if the former is less than *half* or *twice* as much as the latter. We chose these thresholds, since they capture our intuitive understanding of deviation. Such thresholds are necessarily somewhat arbitrary—a different choice would result in different deviation values. It is thus hard to interpret, *e. g.*, the difference between 40% and 43% deviation. However, we are convinced that the metric is a good indicator for the magnitude of the usage deviation, since it coincided with the intuitive reaction of the stakeholders to the actual usage results.

Some of the modeled features could not be profiled, either because beacon inference failed or because the beacons were not stable across the analyzed versions. However, since these features were distributed equally across the modeled feature groups, the remaining features are still representative for the system. The missing features, thus, do not invalidate the usage deviation results.

One limitation of the ephemeral profiler is that it cannot differentiate between a single or more executions of a method. It just does not differentiate between, *e. g.*, 1 and 100 executions of a feature during a single day. If this information is necessary, all invocations of feature beacons need to be monitored. The increased profiling impact could be compensated by only instrumenting beacon methods.

H. Discussion

The deviation between the expected and actual usage was substantial. During the five month study period, 40% of the features were used differently than expected by the best and 55% by the worst estimate. 28% of the features were not used at all. We consider these results to be valuable for project control. This was confirmed by the project members, who consider the feature profiling results as valuable information for feature alignment, planning and control.

The analysis of usage expectation consistency revealed that different stakeholders have different usage expectations. This leads to two conclusions. First, the feature model should reflect the expectation from different stakeholders. Its content thus needs to be created collaboratively. Second, explicit modeling of expected feature usage can serve as a catalyzer for communication between different stakeholders. Different usage expectations can result from different perceptions of the importance of a feature. A common understanding of feature usage and centrality, however, can be useful when making trade-off decisions during, *e. g.*, quality assurance effort allocation or time-to-market considerations.

Feature beacon inference is feasible, but failed to produce beacons for some features. The problems can, in principle be overcome by restructuring the analyzed application, *e. g.*, by splitting methods that implement multiple features. Alternatively, more invasive tracing and profiling that inspects parameter and variable values could help to identify further feature executions, albeit at the cost of higher performance impact. However, even with system restructuring and more invasive techniques, reverse engineering of feature beacons is tedious. We expect it to be simpler to determine feature beacons already during forward engineering. Their annotation, *e. g.*, through code annotations, could substantially reduce feature inference efforts. However, beacon inference is still a viable method to determine beacons in existing code or if the analyzed application cannot be modified.

The case study discovered a number of features that were not used at all. The reactions to them are twofold: the unused features with a usage expectation below the study period are candidates for removal, if discussions with stakeholders reveal that they are no longer needed. The remaining features need to be profiled further until a decision can be taken on how to treat them. If they are still unused once their usage expectation has been exceeded, they will also become candidates for removal.

VI. RELATED WORK

We are not aware of other approaches that employ feature-level usage information to support maintenance. However, our approach builds on existing work from several areas. We

relate it to usage mining, feature location, remote analysis of deployed software and program profiling below.

Usage mining analyzes interaction patterns in user behavior. Web usage mining [10]–[12] analyzes server logs to reveal web site usage. Data mining algorithms are employed to reveal frequent interaction patterns. It is employed to, *e. g.*, evaluate web site usability and support personalization. Recently, techniques from web usage mining have been transferred to analyze software systems. In [13], El-Ramly and Stroulia propose to instrument legacy software to infer user interaction patterns. In [14], Murphy, Kersten and Findlater monitor command invocations to analyze how developers employ features of the Eclipse IDE such as software refactorings. In contrast to our work, these approaches are exploratory and do not target maintenance.

Feature location aims to support maintenance by identifying regions of code that implement a feature. Both dynamic [15], [16] and static [17], [18] approaches and combinations thereof [4], [19] have been proposed. Those approaches provided valuable insights into the complex relationship between solution and application domain artifacts to us and provide the foundation for our feature location approach. However, they aim at reverse engineering of feature location knowledge. In contrast, our work makes constructive use of feature locations in order to profile feature executions.

Remote analysis of deployed software has recently been proposed by several researchers. Hilbert [20] proposes to employ agents to collect usage information in deployed software to support usability engineering. Orso et al. [21] investigate means to distribute monitoring tasks across users to reduce associated impact. Liblit et al. [22], [23] propose remote program sampling to isolate bugs. Elbaum and Diep [24] survey existing approaches to support testing by profiling deployed software. Haran et al. [25] present approaches to classify execution data gathered during remote program analysis in support of further analysis. These approaches were a valuable inspiration for our work and provide general indication for the feasibility of profiling deployed software. However, to the best of our knowledge, none of them produces information on the level of application features. They are hence not targeted at usage analysis and maintenance.

Program profiling [26] is an established practice in performance engineering to identify problematic code. Existing approaches can be categorized into exact and statistical profilers. While exact profilers yield precise results, their potentially devastating impact on performance inhibits their application on production machines. Statistic approaches sacrifice precision to reduce performance impact and thus can be applied to continuous profiling of deployed software [27]. Ephemeral profiling, as we employ it, combines exact results with minimal impact—thus combining the advantages of both approaches.

VII. Future Work

In the future, we plan to switch from the presented reverse engineering approach to a forward engineering approach and to use a different mechanism to mark feature beacons.

From Reverse to Forward Engineering. Up to now, we used feature profiling in a reverse engineering manner, *i. e.*, we built a feature model and determined the beacons for an existing system. This is suboptimal for two reasons. First, the feature model itself is a valuable artifact as it explicitly captures the system's features. It should thus be built and maintained in parallel to development, saving effort for its reverse-engineering. Second, the identification of feature beacons is tedious, because developers loose the knowledge about the location of features in the source code. To address these problems, we plan to make feature modeling and beacon identification a standard development activity.

Marking Feature Beacons. We found that some features cannot be monitored since they have no feature beacon (as some features are entered through the same method). Moreover, the study showed that evolution sometimes invalidates beacons, *e. g.*, because a method has been moved or renamed. To remedy this, we want to integrate feature beacons more tightly into the code, *e. g.*, by inserting special logging statements that increment the call counter for their feature. This places feature beacons inside methods and makes them more robust against changes.

First Experiences. While we haven't had the opportunity to fully evaluate the forward engineering approach, we gained experience with in-code feature beacons. We found that they are significantly more robust against code changes and can be created, even retroactively, with relatively low effort—retrofitting the beacon logging statements took about one person week for 120 features in a medium-sized system.

VIII. Conclusion

For effective software maintenance, program comprehension must go beyond a system's inner workings. Understanding the source code and other artifacts is undoubtedly crucial, but is not enough to answer some questions arising during maintenance: How many users will a change to a certain feature affect? Which code can be removed because the features it implements are no longer required? Will a certain feature not be used until the end of the year, so that we can safely move testing effort to other areas for now? To provide better answers, we need to understand the usage context of a software system.

In this paper, we proposed feature profiling as a lightweight approach to continuously monitor system usage on the level of application features. It relies on dynamic analysis of software in production to achieve comprehensive and accurate results.

We performed feature profiling on an industrial business information system for a period of five months for *all* of

its over 100 users in 10 different countries. 28% of the monitored features were not used at all during that period, indicating potential for code removal. Furthermore, the study showed that different stakeholders had different expectations of system usage. Interestingly, actual usage deviated from all of them. However, since many maintenance decisions implicitly or explicitly involve usage context information, we consider this as precarious.

We are convinced that this holds for other systems as well, as distributed development and usage—as in the analyzed system—pose a real challenge for forming and maintaining a consistent understanding of its usage context. We are optimistic that feature profiling can serve as a catalyzer for substantiated discussions and help create a picture that is consistent both among stakeholders and with actual usage. Feature profiling can thus extend program comprehension to include usage context information.

Acknowledgments. We are grateful to Tobias Habermann for supporting feature inference and the implementation of the feature editor, and to Lars Heinemann for helpful comments.

References

[1] B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "Characteristics of application software maintenance," *Commun. ACM*, vol. 21, no. 6, pp. 466–471, 1978.

[2] J. T. Nosek and P. Palvia, "Software maintenance management: changes in the last decade," *Journal of Software Maintenance*, vol. 2, no. 3, pp. 157–174, 1990.

[3] L. Huang and B. Boehm, "How much software quality investment is enough: A value-based approach," *IEEE Softw.*, vol. 23, no. 5, pp. 88–95, 2006.

[4] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 210–224, 2003.

[5] J. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin, "The operational profile," in *Handbook of software reliability engineering*. McGraw-Hill, 1996, pp. 167–216.

[6] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Authorhouse, 2004.

[7] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Eclipse Modeling Framework 2.0*. Addison-Wesley, 2009.

[8] O. Traub, S. Schechter, and M. Smith, "Ephemeral instrumentation for lightweight program profiling," Department of Electrical Engineering and Computer Science, Harvard University, Cambridge, Massachusetts, Tech. Rep., 2000.

[9] A. J. Viera and J. M. Garrett, "Understanding interobserver agreement: The kappa statistic," *Family Medicine*, vol. 37, no. 5, pp. 360–363, 2005.

[10] R. Kosala and H. Blockeel, "Web mining research: a survey," *SIGKDD Explor. Newsl.*, vol. 2, no. 1, pp. 1–15, 2000.

[11] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web usage mining: discovery and applications of usage patterns from web data," *SIGKDD Explor. Newsl.*, vol. 1, no. 2, pp. 12–23, 2000.

[12] M. Spiliopoulou, "Web usage mining for web site evaluation," *Commun. ACM*, vol. 43, no. 8, pp. 127–134, 2000.

[13] M. El-Ramly and E. Stroulia, "Mining software usage data," in *MSR'04*. IEEE, 2004.

[14] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the Eclipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, 2006.

[15] N. Wilde and M. C. Scully, "Software reconnaissance: mapping program features to code," *Journal of Software Maintenance*, vol. 7, no. 1, pp. 49–62, 1995.

[16] M. Denker, J. Ressia, O. Greevy, and O. Nierstrasz, "Modeling features at runtime," in *MoDELS'10*. Springer, 2010.

[17] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *IWPC'00*. IEEE, 2000.

[18] D. Poshyvanyk, A. Marcus, V. Rajlich, Y.-G. Gueheneuc, and G. Antoniol, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *ICPC'06*. IEEE, 2006.

[19] T. Eisenbarth, R. Koschke, and D. Simon, "Aiding program comprehension by static and dynamic feature analysis," in *ICSM'01*. IEEE, 2001.

[20] D. M. Hilbert, "Large-scale collection of application usage data and user feedback to inform interactive software development," Ph.D. dissertation, 1999.

[21] A. Orso, D. Liang, M. J. Harrold, and R. Lipton, "Gamma system: continuous evolution of software after deployment," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 65–69, 2002.

[22] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Not.*, vol. 38, no. 5, pp. 141–154, 2003.

[23] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI'05*. ACM, 2005.

[24] S. Elbaum and M. Diep, "Profiling deployed software: Assessing strategies and testing opportunities," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 312–327, 2005.

[25] M. Haran, A. Karr, M. Last, A. Orso, A. A. Porter, A. Sanil, and S. Fouche, "Techniques for classifying executions of deployed software to support software engineering tasks," *IEEE Trans. Softw. Eng.*, vol. 33, no. 5, pp. 287–304, 2007.

[26] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *SIGPLAN'82*. ACM, 1982.

[27] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: where have all the cycles gone?" *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 357–390, 1997.