Which Features Do My Users (Not) Use?

Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker
Technische Universität München, Germany

***Abstract*—Maintenance of unused features leads to unnecessary costs. Therefore, identifying unused features can help product owners to prioritize maintenance efforts. We present a tool that employs dynamic analyses and text mining techniques to identify use case documents describing unused features to approximate unnecessary features. We report on a preliminary study of an industrial business information system over the course of one year quantifying unused features and measuring the performance of the approach. It indicates the relevance of the problem and the capability of the presented approach to detect unused features.**

I. Introduction

Software systems contain unused features. Studies report that in practice 28% [1] to 45% [2] of a system's features are unused. These features lead to code areas that are not used in a productive environment. Maintaining unused code leads to unnecessary costs, since unused code is often unnecessary [3]. However, product owners directing maintenance efforts often do not know the actual usage of their system in its productive environment. Hence, from the perspective of a product owner, the question arises: "How can we identify the unused features to prevent unnecessary maintenance cost?".

To answer their question, we would like to collect usage data on feature level. Due to technical reasons, this is not well established in practice: code profilers record usage data on source code level, whereas profilers on feature level require instrumentation or annotation of source code that explicitly maps source code to features (see, e.g., [2]), which leads to additional maintenance efforts.

Problem: Usage data on code level reveals only which parts of the code, instead of features, are used or unused. As such, this information is helpful for developers who know the source code of the system and can decide whether to keep or remove source code based on this information. But this information does not help product owners, who are on project management or requirements level and, more important, who do not know the system's source code. This is just one reason for product owners being unable to relate unused code to features. Since product owners do not want to direct maintenance to unused features, they require usage information on the level of features, which is not given by profilers employed in industry. Hence, to help product owners planning maintenance efforts, we need to transfer runtime usage information to higher level usage data and bridge the gap between code level usage data and features.

Contribution: We propose an approach using text mining techniques to bridge the gap between code level usage information and features. In this approach, we approximate to features by analyzing use case documents (see, e.g., [4]). By extracting concepts from source code methods and matching the content of use case documents to these concepts, we can suggest those use case documents that describe most likely unused features.

We analyzed a real-world business information system from the reinsurance domain where features are documented in 46 use cases. We present a preliminary study, based on usage data collected for more than one business year, which lead to the discovery of two use case documents that describe unused features and two use case documents containing large parts that were not performed by the actual users of the system. The results show that the precision and recall of the approach are high, and therefore, indicate the validity of our approach.

II. Related Work

Feature Profiling: Besides [2], we are not aware of research work eliciting feature level usage information on the level of requirements. However, one approach to detect unused features is linking code to features manually. This is a tedious task, if even feasible in practice, since possibly thousands of locations in the source code have to be linked to requirements documents. Additionally, these links possibly become outdated in a changing system and this leads to even more efforts.

Software Usage Mining: Several approaches gather information about the usage of software: techniques for web usage mining [5] gained much attention by researchers, however, there are also techniques for other types of systems [2], [6], [7]. The difference to our work is that those approaches do not establish connections to requirements documents, such as use cases. Furthermore, these approaches often rely on a certain structure or instrumentation of programs, e.g. [7] is specific to software built on top of the Eclipse Framework. Our approach, in contrast, is generic as it inspects the source code text.

Feature Location: Feature Location refers to the task of identifying code areas that implement a certain feature. An extensive survey on feature location is given by Dit et al. [8]. Among the techniques proposed for this task are static and dynamic analyses [9] as well as text mining techniques [10], which is what we applied here. Text mining has been used for feature location, e.g., in [11] where features are located based on natural language documents. The main goal of feature location techniques is to answer a query about a feature by providing a list of matching source code areas. This differs from our goal, as we want to decide which features are not executed on the system.

Trace Link Recovery: Trace Link Recovery focuses on uncovering relations between different software-related artifacts. A taxonomy of trace link recovery techniques has been published by Cleland-Huang [12]. In terms of this taxonomy we use a technique based on term matching. Uncovering traces between requirements documents and source code to ease maintenance has been investigated by Charrada [13],

[14]. Lucia et al. [15] present an approach and a tool to automatically uncover trace links between a wider range of artifacts, among them use cases. Lormans [16] uses trace link recovery to identify requirements that have not been implemented. Compared to these works, which often use text mining techniques [17], we share the underlying techniques such as LSI, but establish relations between use cases and source code with the goal to find unexecuted features.

III. Approach

We explain our approach to identifying unused features expressed by use case documents, which is based on usage data analysis and Latent Semantic Indexing (LSI) for linking unused code to use case documents.

A. Background: Latent Semantic Indexing

LSI [18] measures the similarity between documents contained in a document corpus. Similarity is expressed by a value between -1 and 1, where a higher value means the compared documents are more similar[1]. LSI identifies words belonging to a common concept (e.g., 'car' and 'automobile'), enabling it to deal with synonyms.

LSI starts with creating a term document matrix (*terms* × *documents*) with entries for each word in each document. The entries are calculated by a global and local weighting function for each word. On this matrix, given the number of desired dimensions (which can be interpreted as the number of concepts), singular value decomposition is performed, resulting in a smaller matrix where words are replaced by their concepts. This matrix represents every document as a vector in the space of concepts. The similarity of documents is calculated by comparing their vectors using cosine similarity.

B. Finding Unused Features

The approach is divided into several steps as illustrated in Figure 1 and uses the variables listed in Table I. It assumes the source code using the same concepts in the same language as the use case documents.

TABLE I. Approach: Variable names and meanings.

Variable	Description
M	Set of all method names
U_i	The ith use case document in all use case documents
M_{used}, M_{unused}	The set of used/unused methods
C_{used}, C_{unused}	The set of concepts in method names in M_{used} / M_{unused}
$s_{U_i,used}$, $s_{U_i,unused}$	The similarity of U_i to C_{used} / C_{used}
Σ_{U_i}	The score of U_i

Input:

- *Use case documents* written in natural language describing the flow of steps users perform.
- *Usage data* collected by an ephemeral profiler[2] [19]. The resulting log files list the executed methods[3].
- *Source code* of the software system[4].

① Extract and Filter Methods:
Input: Source code
Procedure: Filter out generated code and test code[5] and extract method names M from the remaining methods' signatures.
Output: M.

② Partition Methods:
Input: M, usage data log files
Procedure: Partition method names by usage (used or unused).
Output: M_{used}, M_{unused}.

③ Extract Concepts for Method Sets:
Input: M_{used}, M_{unused}
Procedure: Extract concepts[6] from method names. We rely on the coding convention that method names are written in camelCase or single words are split by underscores. For example, the method `getRatingAgency()` is transferred into the list of concepts `[get, rating, agency]`.
Output: C_{used}, C_{unused}.

④ Compare (LSI):
Input: Corpus of use case documents C_{used} and C_{unused}.
Procedure: Extract cleaned text[7] from use case documents. Compute the similarities of every use case document U_i to the word sets C_{used}, C_{unused} using LSI[8].
Output: For each use case document U_i: $s_{U_i,used}$, $s_{U_i,unused}$.

⑤ Calculate Scores:
Input: For every use case document U_i: $s_{U_i,used}$ and $s_{U_i,unused}$
Procedure: Calculate the score Σ_{U_i} for every use case document U_i as: $\Sigma_{U_i} = s_{U_i,unused} - s_{U_i,used}$. In this step, we want to score a use case document higher, the more similar it is to the concepts of unused methods, and lower, the more similar it is to the concepts of used methods, since there might be use case documents that match well to both sets of methods.
Output: For every use case document U_i: Σ_{U_i}.

⑥ Sort Use Case Documents:
Input: Σ_{U_i} for every use case document U_i
Procedure: Sort the use case documents by their score in descending order.
Output: Ranking of use case documents. Higher ranked use cases are more likely to describe unused features.

Parameter Estimation: The parameters used for configuring LSI highly impact the results. Therefore, we suggest to estimate parameters based on a sample use case document that contains unused features and is identified manually by a system expert. We iterate through possible parameters and choose the parameters with which the proposed approach ranks this use case document highest. The rationale behind this approach is that identifying just one use case document that expresses an unused feature can be done with less effort than identifying all – which then is facilitated by our approach.

[1] The range of some other implementations of LSI is between 0 and 1.

[2] This profiling technique imposes less performance impact than classical profiling techniques. This improves the applicability in productive systems.

[3] But no information about how often a method was called.

[4] The approach also works on Java-Bytecode and IL Code for C#, since we rely only on method signatures that are extractable from these artifacts.

[5] Test methods are never executed in the productive environment. Generated code does not contain words relevant for our approach and has not been manually maintained.

[6] Single words contained in method names.

[7] We omit information like authors or version history. Additionally, we remove stop words and stem the text.

[8] To compare word sets to documents, we generate one document for each concept set by writing the contained words to a text document.

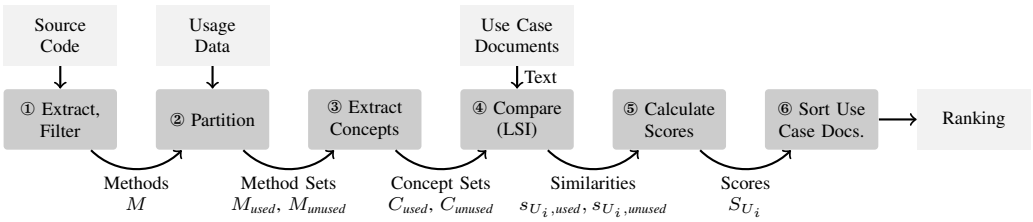

Fig. 1. Schematic illustration of the approach. Light boxes are input or output artifacts, while dark boxes are steps in the approach. Arrows indicate data flow.

IV. Preliminary Study

The goal of this preliminary study is to show the relevance of the research problem and to validate the applicability of the approach in a real world case study. Consequently, we formulate the following two research questions:

RQ1: How many use case documents that express unused features are in the system? This question targets the existence and number of unused features to show that the problem of unused features exists in practice.

RQ2: Is the approach capable of detecting use case documents describing unused features? This question focuses on the validity and performance of our approach in terms of precision and recall, and the applicability in real world examples.

A. Study Object and Subject

We perform the study on a business information system at Munich Re, which is one of the world's leading reinsurance companies with more than 47,000 employees in reinsurance and primary insurance worldwide. For their insurance business, they develop a variety of custom software systems. The business information system analyzed in this study implements damage prediction functionality and supports ca. 150 expert users in over ten countries. Table II illustrates the study object's main characteristics.

TABLE II. Study object: Characteristics.

Language	C#
Age (years)	10
Size (kLOC)	313
Size (#methods)	31,991
Use case documents	46

The system's usage was monitored for 414 days in its productive environment. We filtered out methods and types that only have a testing purpose and did not take external libraries into account. This leads to 11,034 unused methods and 20,957 used methods.

For validating our results, we interviewed the leading architect of the system (*system expert* in the remainder of the paper). Since he has been developing the system for 10 years, he has good knowledge about the system itself but also about the domain. Therefore, we argue that he is capable of estimating whether a feature contained in an use case document is unused based on method usage data.

B. Study Execution

First, we collect usage data, the software system's source code, and its use case documents. From the use case documents, we automatically extract the text.

Second, we present unused methods to the system expert and identify one use case expressing an unused feature manually for parameter estimation[9] (see Section III).

Third, we generate the ranking following our approach as described in Section III.

Fourth, we present the generated ranking to the system expert. He classifies the use case documents as *used* (features contained in the use case document are used completely or only a small part is not used), *partly unused* (large portions of the contained features are not used), and *unused* (the use case was never performed) according to usage data and his knowledge about the system.

C. Results

Our approach produced the ranking shown in Table III. This ranking was produced by using use case document UC2 for the calibration of LSI and we use it for answering our research questions.

TABLE III. Ranking, similarity score, and expert classification of use case documents.

Rank	Name	Score	Expert Classification	
1.	UC1	0.18	unused	
2.	UC2*	0.18	unused	*Used for calibration
3.	UC3	0.16	used	
4.	UC4	0.16	partly unused	
5.	UC5	0.12	partly unused	
6.-46.	...	$\leqslant 0.10$	used	

RQ1: We found two use case documents expressing unused features which were ranked highest by our approach (UC1 and UC2), and two use case documents, which expressed partly unused features (UC4 and UC5). However, UC3 was used according to the system expert. The use case documents ranked below the fifth rank sometimes contained small unused portions, but therein, the features contained in use case documents are described in a too coarse grained fashion (at the level of use case steps or whole flows) to make statements about usage.

RQ2: The top two use case documents in our ranking express unused features. Especially, UC1 was ranked higher than UC2, which was used for calibration and known to contain only unused features. Hence, we found one use case document expressing a completely unused feature. Furthermore, with detecting UC4 and UC5 and ranking them also high, the approach is capable of detecting use case documents expressing unused features. However, we ranked UC3, which describes used features, high, leading to a drop in precision.

[9]For LSI, we used 17 factors for singular value decomposition. The local weighting functions returns 1 if a term occurs in the document and 0 else. The global weighting function is inverse term document frequency.

Since our approach produces a ranking of use case documents rather than classifying use case documents by their feature usage, we calculate precision and recall depending on the number of use case documents (or the number of top ranks) we consider. Precision and recall dependent on the ranks are shown in Table IV. For our calculations of precision and recall, we classify use case documents describing features with large unused parts also as relevant (*true*), because they contain large regions that never have been performed and should be considered by the product owner when planning maintenance. Considering use case documents expressing partly unused features as relevant, our approach achieves an average precision (*AP*)[10] of 0.89, and taking only use case documents into account describing completely unused features, *AP* is 1, since our approach ranks both relevant use case documents highest.

TABLE IV. RESULTS: PRECISION AND RECALL OF THE APPROACH FOR GIVEN RANKS (USE CASES THAT EXPRESS PARTLY UNUSED FEATURES CONSIDERED RELEVANT).

Ranks	Precision	Recall
1	1.00	0.25
2	1.00	0.50
3	0.67	0.50
4	0.75	0.75
5	0.80	1.00
6	0.67	1.00

D. Discussion

RQ1: Unused features exist. Since these features may lead to unnecessary maintenance, we consider the problem addressed by our approach as relevant. When presenting the results to the system expert, he was not aware of the existence of unused feature, which was proven by usage data. Therefore, the reasons for the identified features being unused are unclear and demand further investigation.

However, we gained anecdotal insights into the reasons for the existence of unused features in other systems (reported by the system expert):

- Requirements were demanded and formulated, but never used, because workarounds that were easier to use than the actual system also fulfilled the task.
- Features were not completely implemented at the time of the inspection and were therefore not possible to perform.

RQ2: With good precision considering the highest use case documents in the produced ranking, and also good recall, the presented approach helps finding unused features. The system expert found the information helpful, since based on this he can direct maintenance actions aligned more along the users' needs.

General: With the presented approach, we narrowed down the search space for use case documents expressing unused features from 46 (all) use case documents to 6 use case documents. According to the system expert, the remaining use case documents contained only features that *had* to be used by the actual users to do their daily business. Therefore, our approach reduces the effort that has to be spent to find unused features by giving hints to product owners.

The study object already was cleaned from known unused code and use case documents containing unused features in a refactoring phase before we monitored its usage for the study. Hence, our results only point out to use cases that have been overseen by experts in this previous phase. Thus, we expect more unused features in other systems that were not cleaned.

E. Threats to validity

The results presented might be flawed due to technical errors in the usage data collection. Due to the nature of our profiler that records method calls by registering to the just in time compile event, but also to the inline event of methods, it might record methods as used that were not used[11]. This could produce less unused methods, which leads to less unused features. However, this profiler was used for several years in the environment under consideration (see [3], [21], [22]) without significant or visible errors. Therefore, we are confident that the errors introduced are small, if existent.

In this study, we applied the usage data of one year. Even though different time spans might produce other results, we considered this appropriate. This assumption was confirmed by the system expert.

Furthermore, we might not have collected all relevant use case documents for the system, since these were scattered through the company's storage system. This would lead to possibly more unused features than we presented. However, this leads only to an underestimation of unused features. Additionally, we might also not have found all use case documents describing unused features, since the system expert might not be aware of all unused features. We did not find any methods that belonged to a feature that was not used except for the features expressed by the use case documents we identified.

As we conducted the preliminary study only on one system, generalizability of our results is threatened. Especially, choosing different parameters for the tracing algorithm might be necessary for replicating the study on other systems. To mitigate this threat, we presented our approach to parameter estimation and also estimated parameters using UC1. The configuration resulting from using UC1 rather than UC2 was the same as using UC2, and therefore we consider this threat as minor.

V. FUTURE WORK

Based on the insights we gained in our preliminary study, we are motivated to take further steps in the area of the presented research work. We divide our future work in short term tasks, next steps, and long term items, the future research questions.

[10] We calculate the average precision *AP* according to [20] by

$$AP = \frac{1}{R}\sum_{i=i}^{n} r_i \left(\frac{\sum_{j=1}^{i} r_j}{i}\right) = \frac{1}{R}\sum_{i=1}^{n} \left(P(i)\cdot r_i\right), \text{ where}$$

$$r_i = \begin{cases} 1 & \text{if document at rank } i \text{ is relevant} \\ 0 & \text{else} \end{cases}, \text{ and}$$

$P(i)$ is the precision at rank i, R is the total number of relevant documents, and $n = 46$, since we considered 46 use case documents. *AP* measures how many irrelevant documents are amongst the relevant documents in a ranking.

[11] Such as inlined methods that were not executed.

Next steps: The validation we conducted in this study indicates the abilities of our approach. However, many questions regarding the correctness and limitations are still open. In future work, we plan to perform broader evaluations focusing on the applicability and benefit of our approach.

So far, our approach is uncovering links between source code and use case documents. However, other software engineering artifacts contain valuable information too and can help supporting development and maintenance of software. In future work, we plan to extend our approach to include design artifacts such as user stories or business rules and process artifacts such as change requests or bug tracking issues.

Being aware of unused feature implementation can help to direct maintenance tasks efficiently. In future work, we plan to enrich our methods to create a closer feedback loop to uncover unused features early and thereby reduce maintenance costs effectively from the beginning of the maintenance phase.

Future research questions: One question arising from our study is why there are features which have been specified once but are actually never used in the implemented software systems. This information can be valuable feedback for software engineering research to correct requirements elicitation methods and techniques to specify, design and implement just those functionalities which are actually needed.

Furthermore, once unused implementations have been detected it is still unclear how to proceed: the spectrum of options reaches from simply removing unused functionality up to adapting user training so users can optimize their work by using abandoned features. We need structured approaches to cope with unused functionality in software systems.

VI. Summary and Conclusions

Unused features often are unnecessary. Therefore, maintenance of unused features possibly leads to unnecessary costs. Unfortunately, information about the usage of a software system is collected by profilers on code level. This leads to the problem that product owners, that do not know the source code of their systems, are not able to establish a mapping between usage information and features. Therefore, we provide an approach to the question "How can we identify the unused features to prevent this unnecessary maintenance cost?".

We proposed an approach that bridges the gap between unused code and features using LSI for linking methods to use case documents as an approximation to features and to calculate a ranking that sorts use case documents by their likeliness of containing features that are not used by actual users of a software system. In a preliminary study we showed that our approach yields promising results: with the ranking produced by our approach, we found two use case documents containing completely unused features and two use case documents describing features with large unused parts. Furthermore, we presented results that indicated our approach performs well in terms of precision and recall.

Acknowledgments

The authors thank Christoph Frenzel and Veronika Bauer for their helpful comments on this work, and Karl-Heinz Prommer and Rudolf Vaas for their efforts interpreting the data.

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "EvoCon, 01IS12034A", and "Software Campus project ANSE, 01IS12057". The authors assume responsibility for the content.

References

[1] J. Johnson, "Roi, it's your job," Keynote at XP '02.

[2] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. Prommer, "Feature Profiling for Evolving Systems," in *ICPC*, 2011.

[3] S. Eder, M. Junker, E. Jurgens, B. Hauptmann, R. Vaas, and K. Prommer, "How Much Does Unused Code Matter for Maintenance?" in *ICSE*, 2012.

[4] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Addison-Wesley Reading, 1999.

[5] J. Srivastava, R. Cooley, M. Deshpande, and P.-N. Tan, "Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data," *ACM SIGKDD Explor. Newsl.*, vol. 1, no. 2, 2000.

[6] M. El-Ramly and E. Stroulia, "Mining Software Usage Data," in *MSR*, 2004.

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

[8] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, 2013.

[9] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *Empir. Softw. Eng.*, vol. 18, no. 2, 2013.

[10] N. Alhindawi, N. Dragan, M. Collard, and J. Maletic, "Improving Feature Location by Enhancing Source Code with Stereotypes," in *ICSM*, 2013.

[11] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "SNIAFL: Towards a Static Noninteractive Approach to Feature Location," *ACM Trans. Softw. Eng. Methodol.*, vol. 15, no. 2, 2006.

[12] J. Cleland-Huang and J. Guo, "Towards More Intelligent Trace Retrieval Algorithms," in *RAISE*, 2014.

[13] E. Ben Charrada, A. Koziolek, and M. Glinz, "Identifying Outdated Requirements Based on Source Code Changes," in *RE*, 2012.

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

[15] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering Traceability Links in Software Artifact Management Systems Using Information Retrieval Methods," *ACM Trans. Softw. Eng. Methodol.*, vol. 16, no. 4, 2007.

[16] M. Lormans and A. van Deursen, "Reconstructing Requirements Coverage Views from Design and Test Using Traceability Recovery via LSI," in *TEFSE*, 2005.

[17] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the equivalence of information retrieval methods for automated traceability link recovery," in *ICPC*, 2010.

[18] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by Latent Semantic Analysis," *J. Am. Soc. Inf. Sci. Technol.*, vol. 41, no. 6, 1990.

[19] O. Traub, S. Schechter, and M. D. Smith, "Ephemeral Instrumentation for Lightweight Program Profiling," School of engineering and Applied Sciences, Harvard University, Tech. Rep., 2000.

[20] A. Turpin and F. Scholer, "User performance versus precision measures for simple search tasks," in *SIGIR*, 2006.

[21] S. Eder, B. Hauptmann, M. Junker, R. Vaas, and K.-H. Prommer, "Selecting Manual Regression Test Cases Automatically Using Trace Link Recovery and Change Coverage," in *AST*, 2014.

[22] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer, "Did We Test Our Changes? Assessing Alignment between Tests and Development in Practice," in *AST*, 2013.