How Much Does Unused Code Matter for Maintenance?

Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann
Institut für Informatik
Technische Universität München, Germany
Garching b. München, Germany
{*eders,junkerm,juergens,hauptmab*}*@in.tum.de*

Rudolf Vaas, Karl-Heinz Prommer
Munich Re
München, Germany
{*rvaas,hprommer*}*@munichre.com*

***Abstract*—Software systems contain unnecessary code. Its maintenance causes unnecessary costs. We present tool-support that employs dynamic analysis of deployed software to detect *unused* code as an approximation of *unnecessary* code, and static analysis to reveal its changes during maintenance. We present a case study on maintenance of unused code in an industrial software system over the course of two years. It quantifies the amount of code that is unused, the amount of maintenance activity that went into it and makes the potential benefit of tool support explicit, which informs maintainers that are about to modify unused code.**

***Keywords*-Software maintenance, dynamic analysis, unnecessary code, unused code**

I. Introduction

Many software systems contain unnecessary functionality. In [1], Johnson reports that 45% of the features in the analyzed systems were never used. Our own study on the usage of an industrial business information system [2] showed that 28% of its features were never used.

For consumer software, speculative or even unnecessary features might be justified to lure new customers into buying a product. For custom developed software, such as the business information systems developed and maintained at Munich Re, a re-insurance company in Germany, however, they provide no value at all.

For such systems, maintenance of unnecessary features is a waste of development effort. To avoid such waste, maintainers must know which code is still used and useful, and which is not. Unfortunately, such information is often not available to software maintainers. In our own study [2], expected and actual usage frequency deviated from 40% to 53% of the cases (depending on which stakeholder was involved in the evaluation). The picture was even clearer for entirely unused features: for over 70% of them, it surprised the stakeholders that they were not used at all.

Whether unnecessary code causes maintenance efforts depends on whether it actually needs to be adapted during maintenance. On the one hand, we expect *perfective* and *corrective* change requests [3] to mostly arise for features that are important to their users—otherwise they would not complain about bugs or demand changes. For unnecessary features, perfective and corrective maintenance effort can thus be expected to be low. On the other hand, however, *preventive* and *adaptive* maintenance [3] regularly affect code independent of the functionality it implements. Examples include the migration of a software system to a new programming language or platform, or the replacement of a component, such as the underlying database. Such changes also affect unused code. Both adaptive and preventive maintenance are regularly performed during software evolution. Extensive studies on maintenance effort reported that perfective and corrective maintenance constitute merely 48% [4], 64% [5], and 75% [6] of all change requests. The remaining changes comprise adaptive and preventive maintenance. We thus have to expect unnecessary code to be subject to maintenance, too. To perform maintenance tasks cost-effectively, maintainers must know which code is still necessary, and which is not.

Whether code is still necessary or not is determined by the function it fulfills for its users. The value of code is thus not an inherent property, but determined by its context. One way to approximate usefulness of code to its users is to monitor its usage in production, and compare it to its expected usage. If code is never used, and does not implement infrequently used functionality such as failure recovery, maintainers should investigate if the effort for its modification is justified.

However, the recording of usage information and consideration of unused code is not an integral part of software maintenance practice. Based on our experience and that of our industrial partners, we see two reasons for this. First, we have little empirical data on how much unused code exists in software and how strongly it affects maintenance. Second, we lack suitable tool support to capture usage data in production and present it in a way suitable to maintainers. As a consequence, it remains unclear how important unused code for cost-effective maintenance really is in practice. Given the amount of unused code in industrial software, we consider this precarious both for practice and for education.

Problem: Real-world software contains unnecessary code. Its maintenance is a waste of development resources. Unfortunately, we lack tool support to identify unnecessary code and empirical data on the magnitude of its impact on

maintenance effort. As a consequence, it is unclear how harmful unnecessary code is for software maintenance.

Contribution: In this paper, we present tool support to collect code-level usage information in a production environment, to approximate unnecessary code. We contribute a case study that analyzes the usage of an industrial business information system over the period of over 2 years. The study quantifies maintenance effort in unused code and shows the potential benefits of the tool support we propose.

II. Outline

The paper is structured as follows: In the next section, we define important terms. Afterwards, we introduce our tool support in detail, followed by a description of the case study. We then discuss our results and related work. We conclude with an overview of future work and a summary of our results.

III. Terms

Method: Units of functionality of a software system. Methods consist of a signature and a body.

Method genealogy: List of methods that represent the evolution of a single method over different versions of a software system. The list contains all versions of one method in chronological order.

Modified: A method genealogy is modified if not all its methods are equal with respect to their signatures and bodies. A method is modified if it is part of a genealogy which is modified.

Unused: A method genealogy is *unused*, if none of its methods is executed in a productive environment. This is not necessarily useless or dead code, but code that was just not executed in a considered time frame.

Unnecessary: A method is *unnecessary*, if it is not needed to fulfill the system's intended purpose and could be removed. Domain and development knowledge is necessary to decide whether a method is unnecessary.

IV. Tool Support

The proposed tool support is divided into four steps: the collection of usage data, the analysis of the program structure, the combination of both in a data repository, and the generation of statistics that can be used by developers. The tool chain is illustrated in Figure 1.

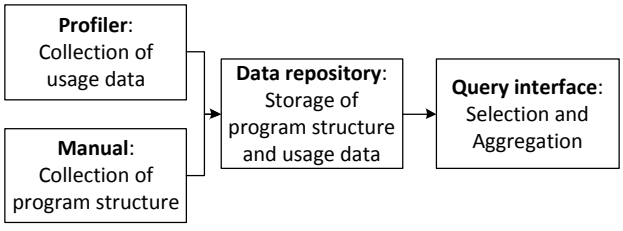

Figure 1. Schematic illustration of the proposed tool chain.

For collecting usage data, we use a profiler based on the .NET profiling API that logs method invocations.

For each program version that gets deployed, we collect assemblies (compilation units in .NET) manually to obtain the structure and functionality of the program. The reason why we work on the binary level and not directly on the source code level is that the system includes several components which are developed by different teams, have different release cycles and are only integrated in binary form. It is thus non-trivial to determine the complete source code for a program version that ran in the productive environment.

Usage data, as well as the software system's structure and functionality are stored in a central data repository. For calculating statistics, we provide a query interface. In the following sections, we explain the different steps in more detail.

A. Profiling Usage Data

When collecting usage data, we need to minimize the impact on the productive system while still providing enough accuracy to gain valuable information. Therefore, we use an ephemeral [7] profiler that records which methods were called within a certain time interval. The profiler does not record how often a method was called, just if it was called in a given time interval. For our study, we set the time interval to one day, to gain data that is accurate enough but produces very low performance impact. More information on the profiler can be found in [2].

Every method is instrumented with a profiling hook at (re-)start of the software system. This hook is removed after the first call of the method and therefore yields no performance impact on later invocations. This technique may miss methods that were inlined by the just-in-time compiler for performance issues, and, thus, we also instrument methods for the inlining event and count this event as an invocation. This is valid, because inlining is performed just in time by the virtual machine. The resulting data is written to a file at every shutdown of the system. Our approach is based on .NET, but not limited to it. It can also be applied to other environments with a virtual machine, such as Java.

B. Data Repository

To store usage data and the structure of every version of the examined software system, we use a database.

For every program version, the structure of the program is stored hierarchically. Program versions are decomposed into assemblies. Every assembly contains types (e.g., classes), which are themselves decomposed into methods. Types and assemblies only carry their names, whereas methods carry their signature and body. Figure 2 illustrates our data model.

After storing the program structure and usage data for all of the program versions, we map methods from one version to the next. Typically, one program version is succeeded by another version including bug fixes and change requests.

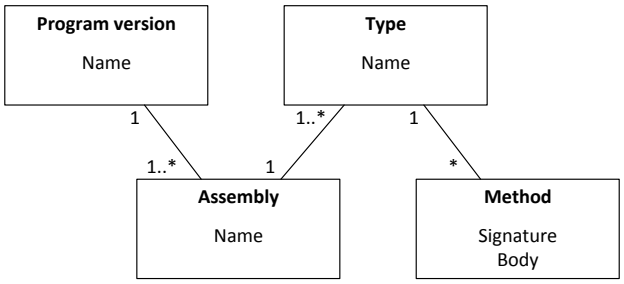

Figure 2. Data model of the program structure.

These changes are reflected in differences in method bodies and signatures, as well as in the structure of types and assemblies and their names. Because of these changes and in order to gain accurate usage data, methods have to be mapped from one program version to another. We perform this mapping by comparing the methods of succeeding program versions with respect to their signatures, bodies, enclosing types, and assemblies. We find the most accurate match for each method by first looking in the original type in the original assembly of the next program version. Types and assemblies are matched based on their names and contained methods. We then rate the similarity of methods, whereas the maximum similarity is given, if a method in the next program version is found in the same type and assembly with an exact match in the method name and parameters. The confidence in the similarity is hampered, if parameters or the enclosing type of a method have changed. The confidence is even lower, if only the parameter types of two methods are the same. We map methods to the most accurate match in the next version. This enables us to build lists of methods of different program versions that evolved from each other. We manage to map about 98% of all methods from one program version to the next. The list follows the ordering of the software system's versions. We call these lists *method genealogies*, as defined in Section III.

Maintenance between two versions is detected by comparing two consecutive methods in the same genealogy. There are three possible actions, a developer could have performed: Add, remove, or change methods.

All of the three actions can be detected in genealogies. If a method was added during the program evolution, its genealogy does not reach to the first program version. If a method was removed, its genealogy does not reach to the last program version and if a method was changed, the body or signature changes in the method genealogy.

Figure 3 depicts the most important kinds of method genealogies. The first genealogy is used twice, but never modified. The second genealogy is never used, but modified twice. The third genealogy is used twice and modified once.

Figure 4 illustrates the sets of method genealogies and their relationships. The method genealogies, that are interesting for our analysis, are *unused* and *modified*. These methods then can be split up into two sets again: Methods that were modified necessarily and unnecessarily.

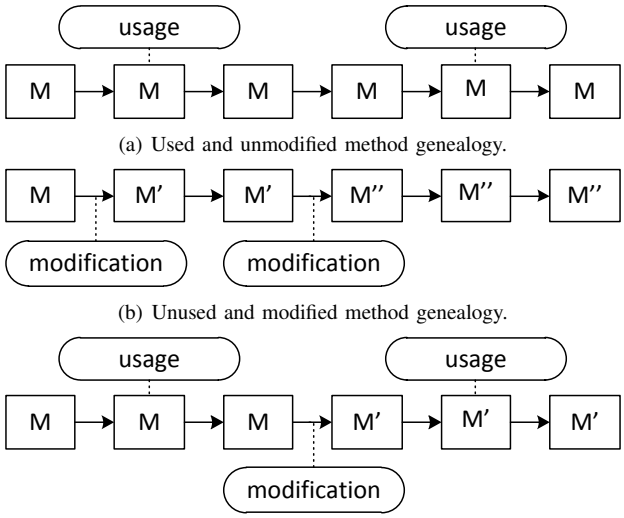

Figure 3. Different types of method genealogies.

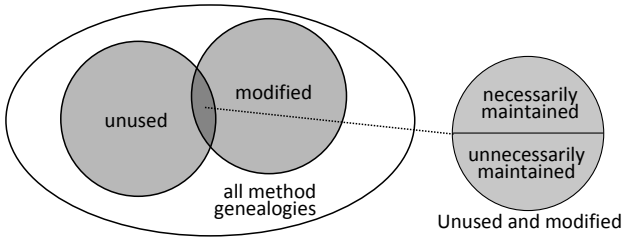

Figure 4. Analyzed sets of method genealogies and their composition.

Having the time interval a program version was productive, we can reconstruct the possible time interval in which a method was changed: Between the start of its own program version and the beginning of the next. We used this information in order to retrieve the source code that was affected by a modification and to discuss it with the developers. If a method was modified, each of its maintenance actions is recorded as single event.

C. Query Interface for Developers

To support developers in maintenance, we provide a query interface for the analysis results. This interface allows generating statistics about the percentage of unused method genealogies, the number of maintenance actions that have been performed on unused method genealogies, and the development of both over time. Thus, we use the query interface for obtaining the relevant number for the case study.

Furthermore, developers can search for methods they are maintaining and retrieve the usage frequency of these

methods. With the help of this information, developers can direct their maintenance effort.

V. Case Study

In this section, we explain the case study we conducted to quantify the impact of unused code on maintenance.

A. Research Questions

We formulate our research objective using the Goal-Question-Metric approach from [8]. The research objective is defined using the goal definition template as proposed in [9]:

> We analyze *usage and maintenance of a large industrial software system* for the purpose of *exploring the role of unused code* with respect to its *effect on maintenance* from the viewpoint of *maintenance engineers and developers* in the context of *industrially hosted business information systems*.

We infer the following research questions:

RQ1: How much code is unused in industrial systems? This question targets the existence and extent of unused code in industrial systems. If there is no unused code, our study would be irrelevant.

RQ2: How much maintenance is done in unused code? Having identified unused code, we answer the question about the existence and extent of maintenance effort that is spent on unused code.

RQ3: How much maintenance in unused code is unnecessary?
This research questions targets the existence and extent of maintenance that gets spent on unnecessary code. This question determines the severity of the problems caused by maintenance actions in unused code and the potential of savings of maintenance effort.

RQ4: Do maintainers perceive knowledge of unused code useful for maintenance tasks?
This question determines the usefulness of the proposed analysis. It is especially interesting whether the analysis helps developers to direct their maintenance effort.

B. Study Object and Subjects

We evaluated the research questions with respect to a business information system being in production 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 custom supporting software systems. The analyzed business information system implements damage prediction functionality and supports about 150 expert users in over 10 countries. An overview is shown in Table I.

Table I
Study object.

Language	C#
Age (years)	8
Size at beginning (kLOC)	360
Engineers (max)	9 (16)
Min. # Methods (size at beginning)	13908
Max. # Methods (size at end)	21664
# Versions	19

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. Understanding the impact of unnecessary code on maintenance is thus likely to decrease maintenance costs. Second, the development and usage context is typical for the 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 at different sites. This distribution of users and engineers complicates communication inside and across the stakeholder groups and could thus lead to a lack of usage information. Third, it is a web application. Its server offers a single point for usage data collection. Our study subjects are two maintainers of the system. Both have been working in the system for 8 years actively. Thus, they have deep knowledge about the system.

C. Study Design

We conduct our study in two major steps. First, we collect program and usage data. Second, we analyze the data in four steps oriented at our research questions.

RQ1: Amount of unused code: We answer RQ1 using the profiling data. We calculate the fraction of the number of unused method genealogies off the overall number of method genealogies for all individual program versions as well as in total.

RQ2: Maintenance in unused code: For RQ2, we need to identify modifications in method genealogies. Modifications in a genealogy occur between two program versions. Therefore, we compare successive methods and determine if they differ. This way, we find all modifications for a genealogy. If a modification took place in an unused genealogy, we conclude that the maintenance effort for this modification was spent on unused code.

RQ3: Amount of unnecessary maintenance and RQ4: How does the analysis help the developers In order to answer RQ3 and RQ4, we discuss our findings with the developers of the system. There are typically large parts of a system's code that are systematically unused in production such as unit tests or code related to batch jobs that are not executed in the productive environment. In order to get meaningful results, it is important to exclude such code from the analysis. Therefore, in a first round, we select unused, but

maintained methods in a way that every part of the system that exhibited unused code is represented in the sample. Additionally, we select methods that seem to be noticeable (e.g., methods with a large change in size or methods whose names suggest unit tests). This results in a set of 24 methods. We present this sample to a developer and use the results to improve the filters in our analysis. These filters are also applied for the measurements to answer RQ1 and RQ2. In a second round, we take a random sample of cases, which we discuss in detail with a different developer of the system in order to elicit the reasons why the code was not used and to quantify the fraction of unnecessary maintenance. We are able to discuss 27 cases with this developer. Furthermore, we investigate how the developer would have acted with knowledge about the unused code and discuss if a tooling as proposed would be helpful for supporting maintenance tasks.

D. Execution

During the analysis period of two years, we gathered usage data of 19 different program versions. Figure 5 shows the distribution of the program versions over time. The uncovered time intervals exist due to missing data that was lost because of technical errors.

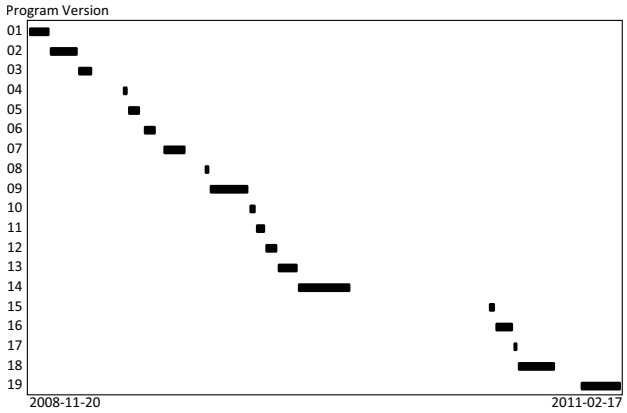

Figure 5. All program versions with their time span of deployment.

In the following, we discuss the concrete procedure for answering our research questions.

RQ1: The investigation of RQ1 requires some prerequisites. The program structure, consisting of assemblies, types, and methods, is extracted from the binary versions of the software system by using the .NET reflection technique. We then store the whole structure of the program in a relational database to perform the method mapping. We map methods based on several heuristics that consider the signature and location of a method, the name of its containing type and assembly.

This results in a multistage mapping procedure to find methods in the succeeding program version. At first, the algorithm maps assemblies and types in the next program version based on their names and contents. Based on the found relations between assemblies and types, methods are mapped. Methods are preferably matched if they have the same location and the same signature. If no completely matching method is found at the exact location, methods are compared based on the method's signature (name, return type and argument types). This way, we can match methods with arguments added and removed or with a changed name. If no match was found at this stage, the whole software system is searched for the method based on the same heuristics as before. This way, we find methods that were moved.

If iterated over all pairs of consecutive program versions, this procedure leads to method genealogies that reach at most from the first program version to the last.

RQ2: Maintenance actions can be derived by comparing method signatures (name, return type, and argument types) and bodies of consecutive methods. For method bodies, we check intermediate language code of methods for equality. If signatures or bodies differ within one genealogy, we count this as a modification.

RQ3: To obtain information about how much code is unnecessary, we present a random sample of method genealogies from the unused, but modified, method genealogies to maintainers. With the help of the maintainers, we split our sample into two groups: A set of maintained and unused, but necessary, method genealogies and a set of maintained, but unnecessary, methods.

RQ4: The interviews we conducted to answer RQ3 also provided information for RQ4. In the interviews, we asked the developers, how useful and interesting the provided information was. Furthermore, having identified the unnecessarily maintained method genealogies, we quantify the accuracy of our analysis by comparing the set of unused, but maintained, method genealogies with the set of unnecessarily maintained method genealogies.

Technical details: We conducted the study using a machine with two 2.4 GHz processor cores and dedicated 4 GB of RAM. We were using a relational database for storing the program structure and usage data. The complete evaluation toolkit is written in Java. Inserting the program structure and usage data of all 19 program versions into the database took about 7 hours. Mapping methods and generating the results took about 5 minutes.

Table II
DISTRIBUTION OF MODIFIED AND UNMODIFIED METHOD GENEALOGIES, DEPENDING ON USAGE.

	Used	Unused	Total
Unmodified	53.3%	22.9%	76.2%
Modified	21.7%	2.1%	23.8%
Total	75%	25%	100%

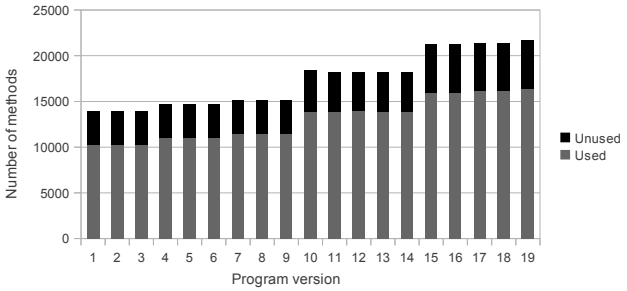

Figure 6. Number of used and unused methods per program version.

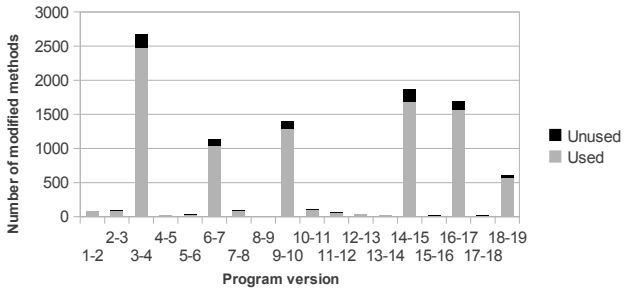

Figure 7. Number of methods that were modified from one program version to the next. The grey bar shows the part of the methods that are used; the black bar shows the part of methods that are unused.

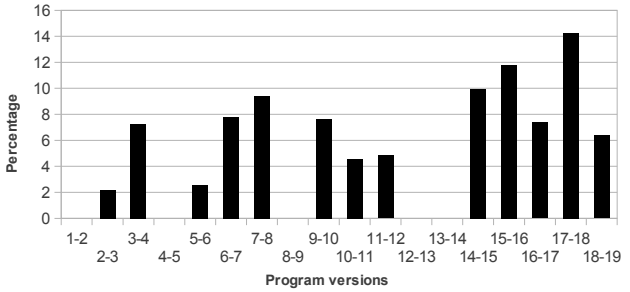

Figure 8. The fraction of the modified and unused methods of all modified methods.

E. Results

In the following, we present the results of our study by providing numbers for our research questions. The numbers are based on the method genealogies we obtained after filtering. That means unit tests and code related to batch jobs that are not executed in production are not included.

In the system, we identified 25,390 method genealogies. Of these, 6,028 were modified with a total of 9,987 individual modifications. This means that considerable maintenance effort took place during the analysis period.

RQ1: Amount of unused code: Table II shows the distribution of used and unused method genealogies. We found that 25% of all method genealogies were never used during the complete period. The fraction of unused methods is roughly stable across program versions, as illustrated by Figure 6.

RQ2: Maintenance in unused code: We first compared the degree of maintenance (i.e. percentage of maintained genealogies) between used and unused method genealogies. We found that 40.7% of the used method genealogies were maintained, but only 8.3% of the unused method genealogies. That means, unused methods were maintained less intensively than used methods. The unused genealogies account for 7.6% of the total number of modifications. Figure 7 shows how the modifications are distributed over the program versions in absolute numbers and Figure 8 shows the percentage of the number of unused methods off the number of modified methods for each program version.

RQ3: Amount of unnecessary maintenance: We reviewed the examples of unused maintenance with the developers. By inspecting the affected code and researching the reason why it is not used, we found that in 9 of 27 (33%) cases, the unused code was indeed unnecessary. In another 4 cases (15%) the code in question was no longer existent as it was either deleted or moved. That means that in nearly every second case unused methods were either unnecessary or potentially deleted from the system. The exact actions could not be retrieved from the versioning system.

RQ4: How does the analysis help the developers: In both discussion rounds, we encountered great interest in the analysis results, especially in the cases in which unused methods were maintained. Often, the developers were surprised that the respective method was not used. When investigating the reason why a particular method was not used, in some cases the developers discovered a bug (e.g., a business function was not correctly hooked into the UI). Regarding the tool support we got the feedback that information about unused methods would be helpful to support maintenance.

Experiences with the mapping of methods: In general, we found that our procedure to identify genealogies works well in the majority of cases. The fraction of methods that could be mapped from one program version to the next usually was around 98%. Only in two cases it was significantly lower (90% and 93%). As it is sensible to assume that a certain number of methods are deleted without replacement due to refactoring or changed requirements, the fraction of methods genealogies that are erroneously terminated should be very small. To support this claim we manually inspected 20 genealogies. As far as we could tell all of them were correct.

F. Interpretation

This section presents an interpretation of the results based on the research questions.

RQ1: Amount of unused code: In our case study, 25% of the implemented code has never been executed and, therefore, can be considered as unused. However, this

amount does not only consist of code of unused features, but also of error handling routines such as exception handlers.

RQ2: Maintenance in unused code: Bringing together the code usage with the locality of the changes, it is noticeable that most of the modifications have been done in code which has been executed. Only 7.6% of the system's changes affect methods that have never been executed. This means that most of the maintenance (92.4%) has been spend on actually used code. For this, we have the following explanation. Change requests primary address the key functionality of a system. Functionality, which is rarely or never executed, is less likely affected by change requests. Furthermore, it is more likely to find bugs in executed code as in code which is not executed at all. Additionally, developers are aware that some code is seldom or never executed and focus their maintenance effort on actually used code.

RQ3: Amount of unnecessary maintenance: Based on our measurements, 7.6% of all maintenance modifications have been performed in unused code regions. To decide whether modifications are unnecessary or not, we can combine the answers from the developer interviews. Therefore, 48% of all modifications performed in unused code can be considered unnecessary. The cleanup of all unused code would cause unreasonable effort. Thus, we suggest tool support to warn developers in case they are performing maintenance tasks in an unused code region. With this, the developer can decide individually if the planned maintenance modifications are necessary or not. During our interviews with the developers, we found that the actions taken to maintain unused and unnecessary code are very similar to the actions that are taken to maintain used code (from refactoring to more complex adaptions). This implies that the actual effort spent on unnecessary code is comparable to the aforementioned numbers.

RQ4: How does the analysis help the developers: Connecting the amount of maintenance in unused code (7.6%) with the ratio of unnecessary maintenance from the interviews (48%), 3.6% of the overall maintenance is needless. However, using usage data during maintenance, a developer can be sure in 92.4% that the performed changes are definitely necessary, whereas in the remaining 7.6% which are not used there is a 48% chance to avoid unnecessary changes. Furthermore, during the developer interviews, several bugs have been detected which are directly related with unused code.

G. Threats to Validity

In this section, we discuss the threats to validity in our study. We structure the threats by internal, external and construct validity.

Internal validity: We consider genealogies as used, if they were used at an arbitrary time during the examination period. If a method was modified after its last usage, we are still considering the method genealogy as used and modified. Therefore, we are missing method genealogies that are unnecessarily maintained after their last usage, because they are not used in the future. This results in an underapproximation of the amount of maintenance actions in unused code and implicates more conservative estimations, which we see, however, as a minor threat.

There are two missing time intervals in our study due to technical errors as shown in Figure 5. Because of that, we compared program versions that did not follow each other directly. However, we are able to map nearly as many methods between these program versions as between all other versions. Thus, we consider this to be a minor threat. The missing data also affects usage data. This results in method usages, which are not considered in our analysis. The discussion with the maintainers did not show any methods that were used in the maintainers' opinion. Thus, we consider this as a minor threat.

External validity: We examined only one system with our analysis. To gain more general numbers about how much unnecessary code is maintained during the software life cycle, the investigation of our research questions on more systems is needed. This allows for a generalization of our findings, whereby we currently plan similar studies on other systems.

Construct validity: Another threat to validity is that method mapping may produce false method genealogies. This effect can be divided into two classes: Methods are set into relation that should not be and methods are not set into relation that should be. This can cause under- and overestimation of the maintenance actions on unused code.

These imprecisions arise due to the lack of information about the exact history of methods, types, and namespaces and the resulting estimation of relationships of different methods. We minimize these effects by implementing a rather conservative search algorithm, which matches method based on several heuristics, considering all possible successors and matching methods with the highest probability. In addition, we manually analyze further random samples of method genealogies. Since we did not find any errors, this strengthens our confidence in a low overall amount of errors.

VI. Discussion

Our analyses, as well as the interviews with the maintainers show results that exceed our research questions. In this section, we discuss these results.

The analysis shows that 3.6% of the maintenance was unnecessary. We expect this low number to be caused by the structure of the development team. Most of the developers are maintaining and developing the system since the beginning and are experts for the domain, as well as for the system. In an environment where developers change more frequently, it is likely that there is less knowledge about the actual usage of the program and, thus, more maintenance in unnecessary code.

According to the developers, the main causes for maintenance of unused code are:

- Exception handling
- Interfaces that had to be implemented
- Code for future use
- Code for testing

We also detected code that was about to be removed or was removed shortly after our examination interval. This means that our analysis was able to identify unnecessary code the developers were aware of.

However, the maintaining developers found the information our analysis provided very useful. In 48% of our findings for maintenance of unused code, the developers did not know why the method was not used. Knowledge about the usage frequency would significantly have changed the behavior of the developers regarding maintenance. In the sample set of methods, the most interesting findings pointed to bugs. For example, we found a method that checked certain conditions that validated a data set. With this check not being performed, it was possible to insert inconsistent data into the system's database. According to the maintainers, the unnecessarily maintained methods are undergoing deeper investigation. These methods are either used in the future or subject to removal.

With our analysis, we narrowed the set of methods to look for unnecessary maintenance from 6369 (all unused method genealogies) to 529 methods (maintained method genealogies that were not used). This makes it a lot easier for maintainers to identify misdirected maintenance effort. This effect was also confirmed by the maintainers. These results and their interpretation as well as the feedback of the developers, points out that this analysis is useful for maintainers in practice.

VII. Related Work

To the best of our knowledge, other approaches that use usage information to find out unused code to support maintenance do not exist. However, our approach builds on existing work from several areas. We relate it to *remote analysis of deployed software*, *program profiling*, *code coverage testing*, *diff and semantic diff*, as well as *unnecessary code elimination*.

Remote analysis of deployed software: has been proposed by several researchers. Hilbert [10] proposes to employ agents to collect usage information in deployed software to support usability engineering. Orso et al. [11] investigate means to distribute monitoring tasks across users to reduce associated impact. Liblit et al. [12], [13] propose remote program sampling to isolate bugs. Elbaum and Diep [14] survey existing approaches to support testing by profiling deployed software. Haran et al. [15] 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 are targeted at usage analysis and maintenance.

Program profiling: [16] 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 [17]. Ephemeral profiling [7], as we employ it, combines exact results with minimal impact, thus, combines the advantages of both approaches.

Code coverage testing: In software testing, metrics such as instruction, branch, path, or condition coverage are used to measure the quality of test suites. These metrics describe to which degree a program has been tested based on its control flow graph. Many testing tools track the control flow during test execution by either injecting additional measurement code or using a system's debugging interface. Both affect the execution time and memory consumption in a negative way. Since our test object was under productive use, none of these techniques could be applied. We focused on tracking just method executions in a lightweight way, which affects the system's run time behavior in a minimal way. We injected measurement code, which is executed just once for every first execution of a method. Since the application has been restarted every day, method usage could be tracked on a day time precision.

Syntactic and semantic diff: Differences between programs can be determined on several levels. There are approaches comparing two versions of a program on the syntactic, as well as on the semantic level. The UNIX diff tool, for example, performs a lexical analysis of two text sources. Even little textual changes, which have no effect on the compilation, such as removing unnecessary line brakes or spaces, will be detected as changes. Some work has been done in finding differences based on abstract syntax trees (AST) of programs [18], [19], [20]. By comparing programs on their abstract structure, lexical changes, which do not affect the behavior, are ignored. Another approach is to compare programs on the semantic level. Semantic Diff [21], for example, creates local dependency graphs to compare their observable input-output behavior. LSdiff [22] uses logical structural deltas to detect and understand similarities in Java code. Since compiling source code already filters little changes of the source code, which does not change the system's semantics, we focused on comparing the binary representation of the system. However, during the compilation of .NET applications, some information, which is necessary to understand the intention of the developer, gets lost.

Unnecessary code elimination: Most compilers perform optimizations to remove unnecessary code. Code, which does not affect the applications result (dead code) or cannot be executed at all (unreachable code), is detected and not included in the compilation result [23], [24], [25]. In some development environments, this analysis is already performed during coding which helps the developer to remove this code. Since all this is performed before executing the system, the decision whether code is unnecessary is based on the static information available at compile time. In our approach, we perform a dynamic analysis using the actual usage data, which exists not until the execution of the system. To this end, we can detect code, which is technically reachable, but still never executed.

VIII. Future Work and Conclusion

In this section, we provide a conclusion, a short summary of our work, and an overview of the results of our case study. Afterwards, we outline our future research plans.

A. Conclusion

Real-world software systems typically contain unnecessary code. Maintenance of this code is unnecessary and produces unnecessary maintenance costs.

To understand the impact of unused code on maintenance, we monitored the usage and maintenance actions of an industrially hosted business information system for over two years. We quantified the amount of unused code and measured how often such code is maintained. Furthermore, we investigated to what extent maintenance tasks on unused code are unnecessary. We conducted our study by using the presented tool support.

From our analysis, we draw two main conclusions: A large portion of the code has not been used over the analysis period of two years (25% of all methods). However, a surprisingly low amount of maintenance (7.6% of all maintenance actions) is spent on this fraction of the software system. Therefore, unused code is not a severe problem in the maintenance of the examined system. But nearly 50% of the maintenance actions that were performed on the unused parts of the system affected methods that were unnecessary and caused unnecessary maintenance effort or even bugs. The information received during interviews with the maintainers of the examined software system indicates that our analysis is helpful for them.

We believe that our analysis would show a greater amount of unnecessary maintenance for projects with a different structure of the maintaining team. We are optimistic that this analysis helps directing maintenance efforts more effectively.

B. Future Work

Motivated by the results of our study, we plan a number of improvements and validations for the proposed analysis in the future.

Increase accuracy: At this stage, we are not able to map functionality of methods that is migrated into other methods. In order to gain more precise statistics, we plan to improve our mapping mechanisms. Moreover, we are working on more elaborated statistics and metrics that measure unnecessary maintenance effort.

Another possibility to increase the accuracy of our analysis is to employ procedures for filtering exception handlers and similar parts of the code from the set of unused and maintained methods. We are optimistic to raise the ratio of unnecessarily maintained code in our findings.

Test control: Our tooling can also be applied to test systems. Combined with knowledge about changes to an underlying system, we can point testers to methods that were changed, but not tested afterwards.

Representative study: In this study, we only observed one large software system. In the future, we will monitor more systems to gain more general results about the maintenance of unused code. Furthermore, the presented tool support only targets systems written in C#. As our approach is not limited to this programming language, we plan to adopt it also for other systems that work on virtual machines, for example, Java.

Acknowledgment

We are grateful to Markus Herrmannsdoerfer for his help with the execution of the study. We also thank Lars Heinemann, Markus Herrmannsdoerfer, and Daniel Méndez Fernández for their helpful comments.

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 '11*, 2011.

[3] IEEE, "IEEE standard glossary of software engineering terminology," Standard, 1990.

[4] D. Yeh and J.-H. Jeng, "An empirical study of the influence of departmentalization and organizational position on software maintenance," *J. Softw. Maint. Evol. Res. Pr.*, 2002.

[5] H. D. Rombach, B. T. Ulery, and J. D. Valett, "Toward full life cycle control: Adding maintenance measurement to the SEL," *J. Syst. Softw.*, 1992.

[6] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, and J. D. Valett, "Understanding and predicting the process of software maintenance release," in *ICSE '96*, 1996.

[7] 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.

[8] V. Basili, G. Caldiera, and H. Rombach, "The Goal Question Metric Approach," *Encyclopedia of Software Engineering*, vol. 1, 1994.

[9] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering: An introduction*. Kluwer Academic Publishers, 2000.

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

[11] 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, 2002.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Notices '03*, vol. 38, no. 5, 2003.

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

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

[15] M. Haran, A. Karr, M. Last, A. Orso, Alessandro d 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, 2007.

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

[17] 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, 1997.

[18] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *PLDI '90*, 1990.

[19] W. Yang, "Identifying syntactic differences between two programs," *Softw., Pract. Exper.*, vol. 21, no. 7, 1991.

[20] J. E. Grass, "Cdiff: A syntax directed differencer for C++ programs," in *UXENIX C++ '92*, 1992.

[21] D. Jackson and D. A. Ladd, "Semantic diff: A tool for summarizing the effects of modifications," in *ICSM '94*, 1994.

[22] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *ICSE '09*, 2009.

[23] R. U. J. D. Aho, Alfred V.; Sethi, *Compilers - Principles, Techniques and Tools*. Addison Wesley, 1986.

[24] A. Appel, *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.

[25] S. K. Debray, W. Evans, R. Muth, and B. De Sutter, "Compiler techniques for code compaction," *ACM Trans. Program. Lang. Syst.*, vol. 22, 2000.