Is Static Analysis Able to Identify Unnecessary Source Code?

ROMAN HAAS, CQSE GmbH, Germany
RAINER NIEDERMAYR, University of Stuttgart, CQSE GmbH, Germany
TOBIAS ROEHM, CQSE GmbH, Germany
SVEN APEL, Saarland University, Germany

Grown software systems often contain code that is not necessary anymore. Such unnecessary code wastes resources during development and maintenance, for example, when preparing code for migration or certification. Running a profiler may reveal code that is not used in production, but it is often time-consuming to obtain representative data in this way.

We investigate to what extent a static analysis approach, which is based on code stability and code centrality, is able to identify unnecessary code and whether its recommendations are relevant in practice. To study the feasibility and usefulness of our approach, we conducted a study involving 14 open-source and closed-source software systems. As there is no perfect oracle for unnecessary code, we compared recommendations for unnecessary code with historical cleanups, runtime usage data, and feedback from 25 developers of five software projects. Our study shows that recommendations generated from stability and centrality information point to unnecessary code that cannot be identified by dead code detectors. Developers confirmed that 34% of recommendations were indeed unnecessary and deleted 20% of the recommendations shortly after our interviews. Overall, our results suggest that static analysis can provide quick feedback on unnecessary code and is useful in practice.

CCS Concepts: • **Software and its engineering** → **Maintaining software**; **Software evolution**; *Software maintenance tools*; • **General and reference** → *Metrics*; • **Social and professional topics** → *Software maintenance*; • **Information systems** → Recommender systems.

Additional Key Words and Phrases: unnecessary code, code stability, code centrality

ACM Reference Format:
Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2019. Is Static Analysis Able to Identify Unnecessary Source Code?. *ACM Trans. Softw. Eng. Methodol.* 1, 1, Article 1 (January 2019), 24 pages. https://doi.org/10.1145/3368267

1 INTRODUCTION

Unnecessary code is code in which no stakeholder has an interest. It is almost a rule that unnecessary code appears over time, no matter whether a traditional or agile development approach is used [2, 20, 25]. Unnecessary code is caused by:

(1) reimplementations for which the initial implementation is still available
(2) changes in stakeholders' interests leading to feature implementations that are no longer used by any user

As an example, Eder et al. found in a study on industrial business applications that about one quarter of the implemented features was not used by any user within two years [7].

Unnecessary code wastes resources in daily software development activities. Getting to know the code base and undertaking development tasks may be easier if the code base is smaller because

Authors' addresses: Roman Haas, CQSE GmbH, Munich, Germany; Rainer Niedermayr, University of Stuttgart, CQSE GmbH, Stuttgart, Germany; Tobias Roehm, CQSE GmbH, Munich, Germany; Sven Apel, Saarland University, Saarbrücken, Germany.

developers have a better overview. During compilation and test, unnecessary code wastes computing resources which slows down feedback from continuous integration pipelines to developers. As far as security is concerned, unnecessary code may increase the attack surface of the software. Additionally, at least for mobile applications, it is important to keep binary size as small as possible. Moreover, from a management perspective, maintenance efforts should not be invested into unnecessary code.

Unnecessary code becomes particularly cost-intensive if the whole code base needs to be migrated or has to undergo a certification. Certifications (e.g., security or code quality certification) can cause high costs if, to obtain the certificate, large parts of the software need to be cleaned up, first. An example for a costly migration scenario comes from our experience where a team migrated their software to a new database. They needed to manually analyze all SQL statements using a generic column selection to ensure that the database accesses work correctly and performant. While there are semi-automatic checks for this, manual reviews are still required because the checks are not working sufficiently reliable enough. Similar high efforts are to be expected for other cross-cutting, technical changes like the substitution of a library.

A common way to uncover unnecessary code is to *profile* the program execution for a certain time, that is, record which code is executed in production [7, 17]. Using such a *dynamic analysis* approach, code that is not executed can be identified and, as it was not executed during runtime, it might not be necessary. In general, profilers can have a perceptible influence on the performance of the profiled software system, but as only coarse-grained execution data is needed, this might be negligible. However, depending on the domain and extent of the software, such an analysis might require a long recording time span as even core features may be used only rarely (for example, annual financial statement functionality in business applications or inventory features in a logistics application). When the question of unnecessary code arises in practice (for example, when a migration is planned), it is typically no option to wait some more years until meaningful execution data is recorded. Hence, cheaper, complementary solutions that approximate unnecessary code without an expensive and lengthy dynamic analysis would be helpful.

In this work, we investigate an approach to identify unnecessary code *statically* based on the hypothesis that the *most stable* and, at the same time, *least central* code in the dependency structure of the software is likely to be unnecessary. For this purpose, we implemented an analysis that uses stability and centrality measures to recommend files as unnecessary code. These recommendations are meant to be starting points for practitioners to remove unnecessary code from their code base. Static analysis has less data at its disposal; in particular, runtime information is not available. So, we expect that the advantage of rapid feedback of static analysis comes with costs regarding the precision of the recommendations. Still, the key question is to what extent a static analysis approach helps developers to identify unnecessary code and whether they adopt recommendations by removing unnecessary code from the code base.

Identification of unnecessary code is a difficult problem. So, to validate whether recommendations of our static analysis approach represent unnecessary code, we employ three different oracles. (1) We compare code recommended as unnecessary with code that has been removed in historical *cleanup* commits (for 10 systems). (2) We check whether code recommended as unnecessary was not used in production environments (for 3 systems for which we have representative runtime *usage data*). (3) 25 developers reviewed recommendations of unnecessary code in a series of *interviews* (for 5 systems).

Our evaluation shows that deleted and therefore unnecessary code was, on average, more stable and less central in the dependency structure of the subject system. In addition, our recommendations refer to unused code in 64%–100% of all cases. The developer interviews revealed that 34% of recommendations pointed to unnecessary code. 29% of this code was still reachable, and only

32% have been spotted by a dead code detector. Developers found the recommendations useful and deleted 20% of recommended code from their code base. So, while being not perfect—as to be expected—a static analysis approach can provide quick feedback on potentially unnecessary code and is useful in practice.

This work makes the following contributions:

- A static analysis approach to identify potentially unnecessary code based on code stability and code centrality measured at the file level
- An empirical study on 14 open-source and closed-source projects using cleanups, runtime usage data, and developer interviews as oracle of unnecessary code, investigating to what extent static analysis is able to identify unnecessary code.

All data and more background on this work can be found at our supplementary Web site: https://figshare.com/s/8c3f63d2c620d1aae83a.

2 TERMINOLOGY

The aim of this work is to introduce and evaluate an approach for identifying unnecessary code using static code analysis. In what follows, we define the terms "unnecessary code", "unused code", and "dead code", and distinguish them from one another.

Used and Unused Code. Used code is executed at runtime within a certain time span in a production environment. In contrast, unused code is not being executed at runtime within that time span.

Dead Code. Dead code is code that is not reachable in the control flow graph from any entry vertex from the application code. Therefore, it cannot be executed during runtime, and hence, dead code is always unused code.

Unnecessary Code. Unnecessary code can be deleted from the code base because no stakeholder of the software project has an interest in it.

Unnecessary code is not the same as dead code, because unnecessary code can still be reachable in the control flow. Still, dead code can be unnecessary, but does not necessarily need to be. Examples could be implementations of new features that are not yet integrated into the software system and are therefore not reachable, yet. Unused code is also not necessarily unnecessary code. For instance, error handling, recovery, or migration code is considered as useful even if it is not (regularly) executed. The same applies for test code, which is not executed in a deployed production environment.

3 RELATED WORK

To the best of our knowledge there has been no prior work on identifying unnecessary source code using static analysis. There are three related research areas, though: prediction and detection of dead code, code debloating, and (dynamic) usage analysis.

3.1 Dead Code Detection or Prediction

Many developer tools provide the functionality to detect dead code. Unfortunately, this feature is referred to as unused or unnecessary code detection. Examples include the *unused code detection* in IntelliJ IDEA [10] and the feature to *remove unused resources* in Android Studio [34]. These tools are working on a different level of granularity as they are identifying unused variables or (private) methods. The Eclipse plugin *Unnecessary Code Detector* [13] is also able to detect unused classes, interfaces, and enums. We will use UCDetector as a baseline for the evaluation of our approach.

Streitel et al. [33] detect dead code on class level using dependency and runtime information. While our approach also relies on dependency data, we avoid the need of representative runtime information because it is often not available from production environments. Instead, we use a

heuristic based on implementation history and dependency information to identify unnecessary code.

Eichberg et al. [8] present a static approach to detect infeasible paths in code, aiming at the revelation of, for instance, unnecessary code or bugs. They use abstract interpretation to identify program execution paths that are not reachable. Our work focuses on unnecessary code and is not limited to dead code. Our approach is more coarse-grained in that we take only whole files and directories into consideration (see Section 4).

An approach related to ours is described by Scanniello [29]. He suggests a set of object-oriented and code-size metrics to predict dead code, where LOC, (weighted) methods per class, and the number of methods are the best predictors for dead code. We identify unnecessary code with the help of code metrics. However, our selection criteria are stability and centrality (with respect to the dependency graph of the software system).

Fard and Mesbah [9] implemented JSNose, a tool for detecting JavaScript code smells. Beside a large set of smells that are specific to JavaScript, they aim at detecting some generic smells, too. Unused or dead code are among these. Technically, Fard and Mesbah detect potentially unreachable code using a static analysis on the abstract syntax tree (AST). To detect unused code, they rely on runtime data, that is, they perform a dynamic analysis. In contrast, our approach identifies unnecessary code based on a static analysis without any runtime information.

3.2 Code Debloating

Jiang et al. [15] present *JRed*, a static analysis tool that trims Java applications, as well as the JRE, on the basis of class and method reachability. They aim for attack surface and application size reduction. Their focus lies on environments that have a specific purpose where feature-blown library functionality is not needed in its entirety. *JRed* constructs a call graph for the application and its used libraries, performs a conservative reachability analysis and finally trims unreachable application and library code. Our approach identifies unnecessary application code, without taking external library code into consideration, and is able to identify unnecessary code that is still reachable.

Sharif et al. [30] developed *TRIMMER*, which debloats applications that are compiled to LLVM bitcode modules. They rely on a user-defined configuration specification to perform three transformations: input specialization, loop unrolling, and constant propagation. This makes it possible to prune functionality that is not being used in this specific configuration. That is, *TRIMMER* is able to prune code that may still be reachable. However, fine-grained configuration information is required to be able to identify pruning opportunities, which is not necessarily available. Redini et al. [26] present *BinTrimmer*, which also aims for debloating of LLVM bitcode modules, but does not need any configuration (in contrast to [30]). In this work, we present a more coarse-grained approach that provides recommendations on unnecessary code for further manual investigation and, potentially, deletion.

3.3 Usage Analysis

In this section, we exemplarily show how runtime usage data are collected and used in the literature to detect unused and unnecessary code.

Juergens et al. [17] suggest feature profiling to monitor actual system usage at the level of application features. In a study on an industrial business information system, they found that 28% of the features were not used (over a timespan of 5 months). That is, over a quarter of the implemented features are candidates for code removal, that is, in this system, there is a lot of potentially unnecessary code. Hence, our assumption that there is a considerable amount of

unnecessary code is plausible. An important design decision is that our approach is static, such that it does not require to invest the time needed for a dynamic analysis to produce representative data.

Eder et al. [6] conducted a study where they instrumented an industrial business information system. They recorded usage data and analyzed the data with text mining techniques to identify use case documents that describe features that are actually unused. They found that, at least, 2 of 46 use cases do not occur in practice. Eder et al. even expect a higher proportion of unnecessary code, because on their system, experts had already cleaned the project before their study. Their work motivates our research and has similar aims. Still, we avoid the effort of a dynamic analysis and use static information to obtain much faster feedback on the question of which parts of the system may be unnecessary.

In another study, Eder et al. [7] collected usage data over two years for the same business information system, this time at the method level. They report that 25% of all methods have been never used over two years. They also found that less maintenance effort was put into unused code. Nevertheless, 48% of the modifications on unused code were unnecessary. That is, maintenance costs arose for unused code and could have been spent better on used code, although the overall proportion of unnecessary maintenance effort was relatively low (3.6%). Eder et al. pointed out that this fraction is most likely higher for other systems where not all developers are experts and have deep knowledge about their product.

4 A STATIC ANALYSIS APPROACH

We aim at identifying code that became unnecessary over time because reimplementations happened or stakeholder interests have changed. This may well lead to code that is not changed anymore, that is, *stable* code. While unnecessary code may well be stable, not all stable code is likely to be unnecessary: core concepts and features are also not modified frequently but are highly relevant.

Therefore, it is not sufficient to consider only code changes to identify unnecessary code. This is why we take also *centrality* of code in the dependency structure of the system into account: central features and concepts can be identified statically [3, 32] and should not be classified as unnecessary code. However, less central code that has not been changed for a while, might be unnecessary. So, our hypothesis is that stable and decentral code is *likely* unnecessary.

In the following, we describe a static analysis approach that implements this hypothesis. Although it is possible to apply our approach at the method level, we will operate at the file level to obtain suggestions for code that can be deleted as easy as possible. That is, our recommendations consist of (sets of) whole files. We derive recommendations for unnecessary code from static analysis data, that is, the set of most stable and least central files. Once identified, our approach groups the files in chunks and recommends the biggest chunks as most valuable candidates for removal to developers. The following subsections describe these two steps in more detail.

4.1 Stability and Centrality

To identify the most stable and decentral files, we need to define corresponding stability and centrality measures.

4.1.1 Stability. Software systems are inherently changing and, for studying the changes and its consequences, many different metrics are used. As our approach aims for the opposite of change, that is, stability, we tried to reuse one of the metrics from the literature [18, 19, 22–24]. Unfortunately, none of them fitted our needs because they do not appropriately consider the change history of a file, especially the number of changes and the point in time of each change. Hence, we propose a novel stability metric and explain it step by step in what follows.

We start with the calculation of a change score for each file which lies in $[0, 1]$ (0: no changes, 1: changed in all commits). We obtain the stability value by subtracting this score from 1, as stability is the opposite of change.

Software systems are typically not growing linearly in time but linearly in the number of commits [1]. That is, for example, there are weekends or holidays where much less changes are applied than in project phases with high development pressure. This is why our metric is commit-based.

We assume a linear software system history. That is, there is a total order over all commits $c \in CS$ of the commit set CS with sequential IDs, $1, \ldots, n$, that describe the evolution of the system's state:

$$\emptyset \rightarrow c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_n \tag{1}$$

Such a linearization of the commit history is always possible, even for branch-based development settings [21]. For example, to calculate stability for parallel branches, it suffices to interleave commits by their commit date to obtain a linearized commit history.

Most recent changes are least likely to be unnecessary and therefore, they have the highest impact on the change score. That is, there is a weight for each commit that depends on its position p in the sequence of in total n commits:

$$\text{weight}_c = \begin{cases} (1 - \text{weight}_{\text{min}}) \cdot \frac{|CS_{rec}| - (n-p)}{|CS_{rec}|} + \text{weight}_{\text{min}} & c \text{ is recent} \\ 0 & c \text{ is not recent} \end{cases} \tag{2}$$

We consider the set $CS_{rec} \subset CS$ of recent commits. That is, we sort all commits by their age and take only a fraction f_{rec} of all commits into consideration, such that $|CS_{rec}| = f_{rec} \times |CS|$, $f_{rec} \in [0, 1]$. Recent commits will have a minimal weight denoted as $\text{weight}_{\text{min}}$.

For each file f, we identify all commits that modified the file, denoted as the set CS_f. Note that we do not take modifications into account where only simple semantic-preserving refactorings (e.g., renamings, code moves) were applied, using the approach presented by Dreier [5]. We sum up the weights of the commits in which f was changed and normalize the result by the sum of all weights of the commits. Finally, we obtain the stability values as follows:

$$\text{stability}_{\text{f}} = 1 - \text{changeScore}_{\text{f}} = 1 - \frac{\sum_{c \in CS_{rec}} \text{weight}_c}{\sum_{c \in CS} \text{weight}_c} \tag{3}$$

4.1.2 Centrality. Besides stability, we consider independence from the rest of the system as an indicator for unnecessary code. Intuitively, central files on which many other artifacts of the system depend are highly relevant and cannot be deleted or changed easily. Files that have no or only a few files that depend on them are much easier to delete and probably less important for the system and, as a consequence, more likely to be unnecessary.

To identify the least central (i.e., "decentral") files, we construct the dependency graph $G = (V, E)$ of the software system, where code files f form the node set V and edges $e_{i,j} = \{f_i, f_j\}$ represent dependencies such as method calls or inheritance relationships between files f_i and files f_j. For object-oriented languages, we use classes instead of files as nodes in the dependency graph. To extract dependency information, we use an approach similar to Deissenboeck at al. [4].

As centrality measure, we use standard techniques from network science to rank nodes by their centrality (for details, see Section 5.5).

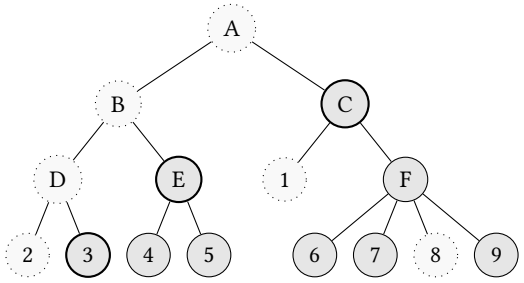


Fig. 1. Chunking. Files 1–9, directories A–F, $U = \{3, 4, 5, 6, 7, 9\}$, $f_{uc} = 0.75$. Necessary elements are dotted, unnecessary elements are solid. Recommendation: $\{C, E, 3\}$

4.1.3 Potentially Unnecessary Files. The stability and decentrality measures provide us two rankings over the source code files, that is, we can identify the set S of most stable files and the set D of most decentral files. Both sets can be determined by sorting the list of files, once by stability and once by decentrality, and cutting it off after a user-defined percentile p_s or p_d, respectively. As the distribution of stability and centrality values highly depends on the system, we use percentiles and not absolute thresholds. We determine a set U of potentially unnecessary files as $U = S \cap D$.

4.2 Chunking

In practice, the set of potentially unnecessary files may easily get very large. That is, it may be difficult to obtain an overview of files and their relations to other files in this set. As we aim for a recommender system that provides helpful suggestions, we need to reduce the number of recommendations to a manageable size.

To this end, we build chunks C_i of unnecessary code files, each containing multiple potentially unnecessary files and even whole directories. Recommendations of directories (e.g., packages or modules) are easier to understand and manage than presenting all their contained files. We consider directories to be potentially unnecessary if, at least, a fixed fraction of children $f_{uc} \in [0, 1]$ is marked as potentially unnecessary. Children are weighted by their size in LOC. If $f_{uc} < 1$, directories are classified as unnecessary even if not all children are marked as unnecessary. This balances precision and usability, as our approach is meant to provide recommendations that need to be processed by humans.

The general idea of chunking is sketched in Figure 1, where files 1–9 are equally large in terms of LOC, A–F are internal nodes of the tree (e.g., directories), $U = \{3, 4, 5, 6, 7, 9\}$, and $f_{uc} = 0.75$.

Our aim is to provide as beneficial recommendations for unnecessary code as possible. As deleting more code has in general a higher effect on maintainability, we suggest to recommend the largest chunks of unnecessary code, first. That is, we sort the chunks C_i by their size $|C_i|$ (counted in LOC, as suggested by Scaniello in [29]). Finally, and in accordance with Robillard et al. [27], we limit the recommendation size to, at most, 10 chunks, that is, we recommend the set $R = \{C_1, \ldots, C_{10}\}$.

5 STUDY DESIGN

We implemented our approach as a recommender system [14, 27] to evaluate our work on 14 open-source and closed-source software systems. To validate the recommendations, we employed three oracles to decide whether recommended code is actually unnecessary: cleanups, runtime usage data from deployed versions of a software system, and feedback from developers (see Section 5.2). The overarching question of our study is whether our approach is able to make *practically relevant*

recommendations for unnecessary code. For the study design and reporting, we follow the general guidelines for case study research by Runeson et al. [28].

5.1 Research Questions

RQ 1: Has deleted code been stable and decentral? Our hypothesis is that unnecessary code is likely stable and decentral. In a first step, we validate if this is true by mining code repositories: we determine whether code that was actually deleted by developers was stable and decentral.

RQ 2: Do code stability and code decentrality identify unnecessary code? Our static analysis approach identifies unnecessary code based on code stability and code decentrality. We investigate the precision of recommendations of our approach by comparing them to historical cleanups, usage data, and feedback from a series of developer interviews. We implemented our approach as a recommender system. We limit the recommendation size (see Section 4.2) such that developers get manageable input on unnecessary code. Limiting the recommendation size implies also to lower recall values, though. As we focus our approach on usability, we concentrate precision as performance metric for this evaluation (but we report recall values on the supplementary website, for completeness).

RQ 3: What fraction of unnecessary code is dead code and can be identified by a dead code detector? Previous work mainly focused on dead code detection or code debloating (see also Section 3), i.e. the detection of unreachable code. Our approach aims at recommendations of unnecessary code, which can–but does not need to be–dead code (see also Section 2). This research question investigates how much of the code recommended as unnecessary is actually dead to see to what extent our approach provides additional information beyond dead code detection. Thereby, we compare the precision of our approach with existing work and use it as baseline for the interpretation of our evaluation results. We use different dead code detection mechanisms and compare strengths and weaknesses of the different approaches regarding unnecessary code detection.

RQ 4: Do developers delete code recommended as unnecessary? We aim at supporting developers in identifying and removing unnecessary code so that they can focus resources on relevant parts of their system. We assume that developers are very cautious when deleting code, because the deletion of still needed code likely has negative consequences (e.g., complaints from users, higher management attention, more work to restore deleted code). As a consequence, even if developers consider a given piece of code as unnecessary, in doubt, they might not delete it to avoid negative effects. So, if developers follow our suggestion and delete recommended code, this underlines the usefulness of our approach.

RQ 5: What are characteristics of false positives? As we are applying a static approach that does not consider any runtime information, we expect to have false positives. We investigate incorrectly classified chunks to understand the limitations of our approach (beyond being a static approach) and to identify possible improvements.

5.2 Three Evaluation Oracles

Typically, there is no ground truth for which code is unnecessary. This is the reason why we use three different oracles to validate recommendations:

- Historical cleanups
- Runtime usage data
- Developer interviews

From cleanups, we can learn how stable and central that code was that has been deleted in the past. Runtime usage data provide insights on which code is actually used in production and

therefore necessary. Developers who are familiar with their code base are able to use their expert knowledge to identify unnecessary code to a certain extent.

Cleanups. We identified cleanup commits in the commit history of nine of our study subjects (see Section 5.3). A commit was considered as cleanup if (1) whole files or even packages were deleted and (2) if its commit message clearly stated that unnecessary code was deleted. File movements were not considered as cleanups. We identified file movements using the clone detection algorithm of Juergens et al. [16]. Their clone detector is scalable, incremental, and language independent, which makes it possible to consider large code bases and histories in our study.

Technically, we used the following case-insensitive pattern to identify cleanups by their commit message:

```
.*(remove(d)?|delete(d)?|unnecessary|unused|not\sus(e|ing)|obsolete).*,
```

but we excluded matches of the following pattern:

```
.*remove[ds]?\s(clone|finding|todo).*,
```

to remove commits from our cleanup list where `TODO`-comments were resolved or findings from code analysis tools were addressed (i.e., where no whole unnecessary files were deleted).

For each commit directly preceding a cleanup commit, we generated recommendations for unnecessary code and extracted data (stability and decentrality scores) of deleted and recommended files for further analysis.

Usage Data. For three of our study subjects (see Section 5.3), we recorded usage data using a profiler to determine which methods were executed, at least, once. To obtain representative data, we recorded usage over a period of 6–16 months, covering also critical time spans such as year closing or inventory time. We also generated recommendations $R = \{C_1, \ldots, C_{10}\}$ for unnecessary code using our implementation. We considered each file f_i in a recommendation $C_j = \{f_1, \ldots, f_n\}$ as true positive when no method declared in f_i was executed at all. In contrast, a false positive, is a file where, at least, one of its methods was executed.

One of the study subjects was highly configurable, that is, available features depended on the software configuration. We explicitly asked the developers to decide for each recommendation whether it was not used due to that fact.

Developer Interviews. 25 developers from 5 software projects (see Section 5.3) validated our recommendations for unnecessary code in their code base. Specifically, we presented them the 10 highest-ranked recommendations for unnecessary code for the most recent revision of their main development branch. We asked them to classify recommendations by answering the question *"Do you consider the suggested file(s) or package(s) as unnecessary?"* using predefined response options such as "Yes, all suggested files or packages are not needed anymore" or "Yes, some files or packages are not needed anymore, but others are currently needed". Furthermore, we asked the developers whether they would actually delete the recommended code from their code base.

5.3 Study Subjects

We evaluated our approach on a number of open-source and closed-source software systems (see Table 1). We selected the most popular open-source projects on GitHub written in Java or C#, which were still actively maintained and provided the possibility to contact developers, for example, on a mailing list or a support forum. We contacted developers from 20 open-source projects and received answers from 3 of them (see Table 1). The closed-source products are developed by professional development teams. Our evaluation partners (two different and independent companies) asked us to anonymize their data.

Overall, the subject projects are from various domains, including software development, databases, search engines, game engines, and business information systems. All projects have comparably

Table 1. Overview of open-source study subjects (top), closed-source study subjects (end), and oracles: cleanups (C), usage data (U), dev. interviews (D)

Project	Oracle	Lang.	LOC	Commits	Cleanups
BAZEL	C	Java	323 K	9.1 K	3
COREFX	C	C#	2.94 M	18.4 K	2
ECLIPSE-RECOMMENDERS	C	Java	84 K	3.8 K	3
ELASTICSEARCH	D, C	Java	909 K	26.4 K	1
JENKINS	D	Java	31 K	24.6 K	0
MOCKITO	D, C	Java	66 K	4.0 K	1
MONODEVELOP	C	C#	1.07 M	48.0 K	1
NETTY	C	Java	346 K	7.9 K	3
OPENRA	C	C#	130 K	22.8 K	3
REALM	C	Java	74 K	6.4 K	2
BIS 1	D	Java	324 K	7.4 K	nA
BIS 2	U	ABAP	4.84 M	nA	nA
BIS 3	U	ABAP	2.26 M	nA	nA
TEAMSCALE	D, U	Java	775 K	71.5 K	nA

Table 2. Oracles used to answer research questions

	Cleanups	Usage Data	Dev. Interviews
RQ 1	✓		
RQ 2	✓	✓	✓
RQ 3			✓
RQ 4			✓
RQ 5			✓

long histories (at least, 1,000 commits and many even several ten thousands of commits). They are of medium to large size (31 KLOC to 4.8 MLOC) and are written in different programming languages (ABAP, C#, or Java). As already explained, we use cleanups (C), usage data (U), and developer interviews (D) as oracles for unnecessary code. Table 1 indicates which oracles were available for which projects.

5.4 Operationalization of Research Questions

Next, we describe how we answer our research questions using the three oracles—see also Table 2.

Research Question 1. We analyze deleted files in cleanups right before their deletion. To be able to compare stability and decentrality of such files over different systems, we report normalized stability and decentrality ranks. We compare these distributions with the distributions for non-deleted files in our study subjects at the same time and test for a significant difference using a Mann–Whitney U test.

Research Question 2. We suppose that only a small fraction of unnecessary code is deleted in cleanups, that is, cleanups provide incomplete information about unnecessary code. So, we expect low performance values as it is unlikely that the biggest chunks being recommended match the

typically small proportion of deleted code. To further investigate this issue, we analyse how many deleted files were identified as potentially unnecessary (but were not recommended because they were not in the largest chunks).

From the usage data, we know whether recommended files were executed in the execution history available for us. We generated recommendations for unnecessary code for the most recent revision with available usage data. We report the proportion of recommended files that were not executed and are therefore likely to be unnecessary from usage perspective. This measure is influenced by the execution ratio e of the investigated study subject which expresses how many files were executed at least once. To be able to interpret the results of this analysis, we need meaningful reference values. Therefore, we calculate the expected recommendation performance of a random recommendation system, which is $1 - e$, as we consider *not*-executed files as unnecessary code in this context. For example, for *BIS 2*, 42% of files were executed. So, this oracle will assess 58% of files as unnecessary which is also the expected performance of a random recommendation system. To asses the recommendation performance of our analysis implementation, we first investigate whether our analysis implementation is significantly better than a random recommendation system. A χ^2 test will be used to estimate whether the precisions of a random recommendation system and our analysis implementation are independent. As the χ^2 test does not provide insights into the degree of independence, we also analyse the effect size expressed with the odd's ratio.

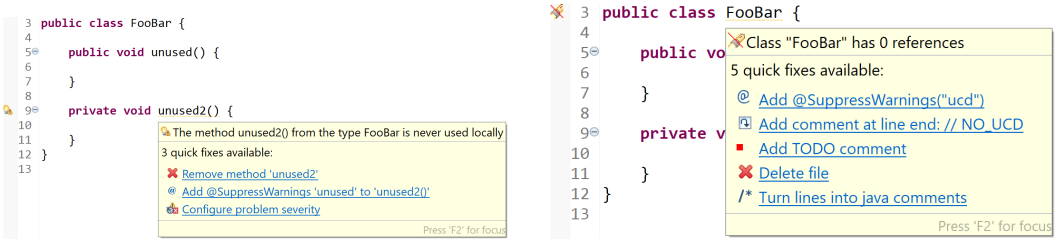
Finally, we analyze data from the developer interviews to learn how many of the recommendations referred to code considered unnecessary by developers (C_{ud}). We use the answers from developers to calculate the precision of the recommendations as $\frac{|C_{ud}|}{|R|}$.

Research Question 3. In this research question, we compare dead code detectors with our approach. For this purpose, code classes are classified as necessary/unnecessary (by our approach) and reachable/dead code (by a dead code detector) and both sets are compared. We aim for a comparison with state-of-the-art dead code detectors that can be used with our study objects. More specifically, the tools need to be able to analyze our study object's code bases, and their licenses should not restrict to non-profit usage because some of the study subjects are commercial applications.

We could not use all our study subjects and oracles for this comparison. For cleanups, we miss information about necessary code as we assume that deleted code is unnecessary but miss information about undeleted code. For usage data, the majority of code bases is written in ABAP and, to our knowledge, there are is no dead code detector which goes beyond (private) variable level available for this language. So, we concentrate on the study subjects for which we interviewed developers. For these, we have recommendations about unnecessary code from our approach, information about the necessity of code from the interviews and information about reachability from setting up dead code detectors.

All of the study subjects from which we interviewed developers are written in Java. Eclipse, a well-known and popular IDE for Java, provides a dead code detection tool, which is enabled by default without the need for special knowledge or configuration. The Eclipse plugin Unnecessary Code Detector (UCDetector) [13] promises more sophisticated analysis for dead code, especially unreferenced class implementations. We use both, Eclipse (version 2019-03) and UCDetector (version 2.0.0) to study how much developers can expect from the Eclipse standard implementation and whether it is worth to install UCDetector plugin. Figure 2 shows example warnings of dead code (please note that Eclipse only detects the private method as unused in Figure 2a). As we found that these tools have limitations, we also analyzed manually whether recommended code classes are actually dead code by checking incoming code dependencies.

To answer RQ 3, we analyze what fraction of files recommended as unnecessary and classified as unnecessary by developers are detected as dead code. We report the analysis results using



(a) Unused private method detected by Eclipse (b) Unused class detected by UCDetector

Fig. 2. Examples for warnings against dead code

contingency tables, which makes it easy to compare and interpret the results. First, we report the performance of the dead code detection tool that is integrated into the Eclipse IDE which detects only unused private fields and methods. We considered a class as dead code if Eclipse detected all methods and fields of it as dead. Second, we use UCDetector to find dead code in our recommendations. Classes for which UCDetector raises a warning, because there are no incoming references or only references from test code, are considered dead code. Finally, we report how many files are actually dead code. To obtain this number, we investigated the source code manually, taking into account incoming references and project-specific information about dynamic class loading mechanisms like dependency injection (which we obtained from our interviews).

Research Question 4. We asked the developers whether they would actually delete the recommended files, which would underline their certainty that the files are indeed unnecessary and show whether they expect a benefit from the deletion.

Research Question 5. We inspected the code of all, according to developers, incorrectly classified code chunks and interviewed the developers to learn why our approach was unable to recognize that the chunks are still relevant. We deduce common characteristics of these chunks and discuss which factors should be considered in an improved version of our approach to reduce the number of false positives.

5.5 Implementation and Analysis Configuration

In Section 4, we presented our approach to identify unnecessary code. The implementation of our approach is based on the software quality-analysis suite Teamscale [12], which provides multi-language support. We implemented facilities for the calculation of stability, centrality, chunks and the recommender system for unnecessary code.

To calculate centrality, we need to find a network algorithm that calculates centrality of nodes in a graph. The same approach was used before by other researchers to find the most important classes of a software system [3, 32, 36]. Their results show that PageRank and HITS without priors are able to express centrality of classes in software systems, which is why we rely in our implementation on one of those two centrality measures.

To find parameter settings that are working well in practice, we selected two software systems that we are familiar with and from which we know that unnecessary code has already been deleted in the history: JabRef, an open-source bibliography reference management tool, and Teamscale, a closed-source software quality analysis suite. In the history of JabRef, we identified 14 cleanups via their commit messages, and for Teamscale, we took 2 larger cleanups into consideration. We used these cleanup data to generate recommendations with various configurations of our approach and identified a parameter set that performed best in recommending files that are being

Table 3. Parameter settings of static analysis approach

Metric	**Parameter** (cf. Sec. 4)	**Value**
Stability	$\text{weight}_{\text{min}}$	0.1
	f_{rec}	0.67
Decentrality	Centrality measure	HITS w/o priors
	Dependency type	import-dependencies
Potentially Unnecessary Files	p_s	33
	p_d	10
Chunking	f_{uc}	0.8
	Max. recommended chunks	10

deleted. Table 3 shows the parameter values for configuring our analysis implementation, which we used throughout the evaluation. We did not use cleanup data from JabRef or Teamscale to answer our research questions because we used these data sources already to configure our analysis implementation. Note, however, that we recorded usage data and received developer feedback from Teamscale, which is why it still appears in the list of our study subjects (Table 1). Details on the selection procedure is available elsewhere [11] and the corresponding data can be found at the supplementary Web site.

6 STUDY RESULTS

In this section, we report the results for each research question. Table 4 summarizes the precision of recommendations for all study subjects and oracles.

6.1 Stability and Decentrality of Deleted Code (RQ 1)

Figure 3 shows box plots of the distribution of normalized stability and decentrality ranks (in $[0, 1]$) of deleted files from all study subjects right before their deletion, compared with the corresponding distribution for non-deleted files. 355 out of 418 investigated deleted files have the highest possible stability rank of 1 because they have the highest possible stability value, that is, they were not changed in the recent past. That is why the corresponding box plot consists of actually only one vertical line. In general, 66% of files are not changed in the recent past, therefore the corresponding median is also close to zero. The box plot indicates that most of the deleted files are very stable, while 63 outliers are comparably unstable. A Mann–Whitney U test confirms that there is a significant difference between deleted files and non-deleted files ($p < 0.05$).

> Summary. *Deleted and non-deleted files belong to different populations with respect to stability and centrality.*

6.2 Identification of Unnecessary Code (RQ 2)

For simplicity, we separate the results for the three oracles.

Results with cleanups as oracle. Table 4 provides average precision values for the recommendations we generated for the software revisions before cleanups. In total, our approach has a rather low average precision of 2.7% for identifying code that is to be deleted.

Table 5 lists more details on the cleanups and how many deleted files were marked as potentially unnecessary, or were even recommended as part of the largest chunks. Due to chunking and

Table 4. Recommendation precision (if applicable)

Project	Cleanups	Usage Data	Dev. Interviews
BAZEL	1.9%		
COREFX	2.7%		
ECLIPSE-RECOMMENDERS	4.6%		
ELASTICSEARCH	1.0%		30%
JENKINS	0%		0%
MOCKITO	6.7%		60%
MONODEVELOP	2.0%		
NETTY	4.5%		
OPENRA	1.0%		
REALM	20.1%		
BIS 1			50%
BIS 2		100%	
BIS 3		100%	
TEAMSCALE		63.6%	30%

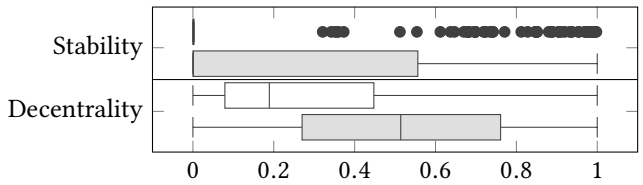


Fig. 3. Distribution of normalized stability and decentrality ranks for deleted files (clear) and non-deleted files (shaded) for all project with cleanup data

the recommendation size of 10 chunks, not all as potentially unnecessary identified files are recommended. In total, we investigate 418 files that were deleted in cleanups. 30.9% of these files were classified as potentially unnecessary by our analysis implementation, which is the recall of our approach for the cleanup oracle before limiting the recommendation size for usability reasons. Taking the recommendation size limitation into consideration, we obtain a recall of 5.5%.

Results with usage data as oracle. Next, we compare recommendations with runtime usage data obtained from three of our study subjects. Table 6 presents the execution proportion of files and how many files were recommended as unnecessary code. The table contains also the number of recommended and non-executed files (which are, from usage perspective, indeed unnecessary). For the first two study subjects, none of the many recommended files were executed, which is a perfect result. The recommendations for the study subject TEAMSCALE contained 12 files (36%) deemed unnecessary but that were executed and are therefore very likely necessary. As the recommendations for *BIS 2* and *BIS 3* covered much more files, and all of them were not used, the average precision is $\frac{1{,}039}{1{,}051} \approx 99\%$.

A χ^2 test on our approach and a random recommendation system with an expected hit rate of $1 - e$ shows that our approach significantly outperforms a random selecting system ($p < 0.001$). The odds ratio for these two subjects is 0.001, respectively 0.002, which implies a very large effect size. For TEAMSCALE, our approach cannot outperform such a random selection ($p > 0.05$).

Table 5. RQ 2: Evaluation of recommendations for potentially unnecessary files using cleanup oracle

Project	Total Deleted Files	Deleted and Potentially Unnecessary	Deleted and Recommended
BAZELBUILD	4	0	0
COREFX	113	33	9
ECLIPSE-RECOMMENDERS	60	8	6
ELASTICSEARCH	37	37	2
MOCKITO	15	1	1
MONODEVELOP	84	20	2
NETTY	58	12	3
OPENRA	45	18	0
REALM	2	0	0
Total	418 (100%)	129 (30.9%)	23 (5.5%)

Table 6. RQ 2: Evaluation of recommendations for potentially unnecessary files using usage data oracle

Project	Execution Rate (e)	Non-executed Files	Non-executed Recommended Files
BIS 2	42%	734	734 (100%)
BIS 3	46%	284	284 (100%)
TEAMSCALE	40%	33	21 (64%)
Total		1,051	1,039 (99%)

Results with developer interviews as oracle. The feedback from developers on recommendations for potentially unnecessary code in their code base was overall positive and is summarized in Figure 4. In total, 50 recommendations (i.e., chunks of potentially unnecessary code) were evaluated by developers. 17 of the recommended code chunks contained classes that were considered as unnecessary by the respective developers. That is, 34% of our recommendations pointed to unnecessary code.

> SUMMARY. *The average precision of recommendations varied between oracles (cleanups: 3%, usage data: 99%, developer feedback: 34%). All oracles indicate that unnecessary code can be identified using code stability and code centrality.*

6.3 Relationship Between Dead Code and Unnecessary Code (RQ 3)

Table 7 shows the three contingency tables displaying the number of classes that are reachable (✷) and dead (†) according to the dead code detection mechanism, as well es necessary (Nec.) and unnecessary (Unnec.).

Eclipse (Table 7a) reported for no recommended class unused private methods so that no class was identified as dead code. So, all 62 unnecessary classes were not detected as dead code by Eclipse. UCDetector (Table 7b) reported in total 76 dead classes, of which 24 were classified as unnecessary by developers, while 52 were *actually necessary*. So, only 32% of the dead code warnings by UCDetector referred to unnecessary code. The other 68% of the dead code warnings refer to code that was classified as necessary by developers.

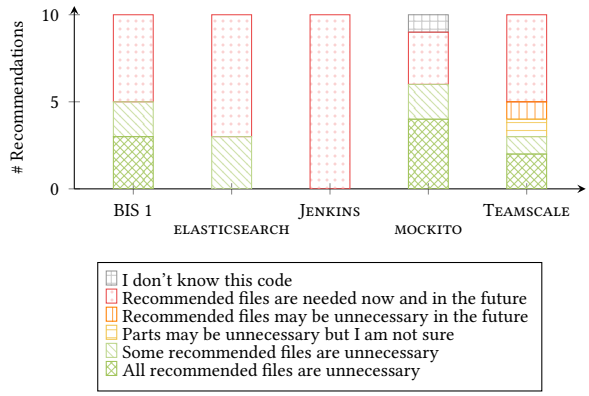


Fig. 4. Developer feedback on recommendations

Table 7. Contingency Tables for Dead Code Detectors and Unnecessary Code

(a) Eclipse

	✸	†	∑
Nec.	179	0	179
Unnec.	62	0	62
∑	241	0	241

(b) UCDetector

	✸	†	∑
Nec.	127	52	179
Unnec.	38	24	62
∑	165	76	241

(c) Manual investigation

	✸	†	∑
Nec.	179	0	179
Unnec.	18	44	62
∑	197	44	241

Our manual analysis has confirmed that, in total, 44 classes considered unnecessary were not reachable during runtime. 18 classes that were reachable were also considered unnecessary by developers. That is, 29% of our correct recommendations were not dead code.

> SUMMARY. *29% of the files recommended as unnecessary by our approach were still reachable and therefore could not by detected by a dead code detector. Hence, our approach adds value beyond dead code detection. Furthermore, dead code detection tools have a low precision as only 32% of files deemed unreachable were indeed unnecessary.*

6.4 Deletions of Unnecessary Code by Developers (RQ 4)

To learn whether developers are certain about their categorization of recommendations into necessary and unnecessary code, and whether they benefit from our recommendations, we analyze the findings from our developer interviews. Each developer team was presented 10 recommendations for unnecessary code and were asked whether they would delete the recommended code. We present our findings for each subject system separately.

BIS 1. Nine developers discussed our recommendations. For two recommendations, they stated that they will delete the corresponding code. They discussed another recommendation related to migration code, and some team members wanted to delete the code, as the corresponding migrations are already finished, while others wanted to keep the code to avoid any risk. One recommendation was related to very old code that no one was familiar with enough to be sure that the code is not needed anymore.

ELASTICSEARCH. Two developers of the ELASTICSEARCH team took part in the interview. They stated that three recommendations that covered some unnecessary code referred to a forked copy of another software project that is currently used by ELASTICSEARCH, but the need for it is being removed slowly. The goal is to remove the fork from the code base.

JENKINS. Two JENKINS developers provided feedback on our recommendations but agreed with none. They pointed out that extensibility of their product is a major design goal as there are thousands of plug-ins available. Due to external and unknown dependencies, it is not easy to decide whether code is unnecessary, even for core developers. Therefore, they are not going to remove any code that our analysis implementation recommended.

MOCKITO. One of the core developers of MOCKITO took part in our interview. He acted upon three recommendations and deleted the corresponding code after the interview. In addition, for the recommendation that he states to be partially correct, he will delete the unnecessary code in the future. Moreover, he wants to delete the code of another recommendation after discussing it with the team.

TEAMSCALE. Eleven development team members participated in a face-to-face evaluation session. One recommendation was related to old code that is actually not working anymore and, hence, the code was immediately removed during the evaluation session. Three other recommendations were deleted shortly after the session.

> SUMMARY. *Developers considered recommendations indeed as unnecessary code and deleted 20% of them shortly after our interview. So, developers benefit from the recommendations and actually delete unnecessary code.*

6.5 Characteristics of False Positives (RQ 5)

The evaluation of the developer feedback revealed that the following characteristics were major causes for an incorrect classification of chunks as unnecessary (i.e., false positives):

- use of dependency injection frameworks, which link specific implementations at runtime and thereby introduce dynamic dependencies; dynamic dependencies cannot be detected by static analysis, which means that injected files appear decentral to static analysis, whereas they are central;
- use of reflection for invocations (e.g., to create instances or invoke methods);
- data transfer classes that are serialized and mainly used outside of the (analyzed) Java code (e.g., in a JS UI);
- interfaces at the system border (e.g., extension points for plugins or provided services), with code accessing these interfaces residing outside the repository;
- interfaces and abstract classes that are rarely changed and only have dependencies within the class hierarchy;
- incomplete feature implementations that are not yet connected to the code base and that were committed in a single commit, and hence, exhibit a low code centrality and a relatively high stability.

Most of these patterns have in common that not all dependencies can be retrieved statically and are therefore missing in the dependency graph that is used to compute the centrality measure. Consequently, the computation of unnecessary files uses an underrated centrality value in such cases, as to be expected.

> SUMMARY. *Most incorrect classifications are caused by missing dynamic or external dependency information.*

7 DISCUSSION

In this section, we discuss the results for each research question as well as more general best practices on how unnecessary code can be handled in practice. We conclude this section with threats to validity.

7.1 Research Questions

Research Question 1. We found that files that were deleted in cleanups tend to be stable and decentral—which supports the key hypothesis of our approach. The reason for why so many deleted files have the highest possible stability rank is that many of them were only changed at the beginning of the project's history, that is, there were no recent changes on them.

Research Question 2. Comparing our recommendations with cleanups, our analysis did not recommend most of the deleted files as unnecessary shortly before the cleanup (resulting in low precision of 2.7%). In many cleanups only a handful of files was deleted (see also Table 5). Since our approach is configured to recommend 10 chunks of files, for usability, there were typically more recommended than deleted files. There is clearly a trade-off between understandability and the precision of recommendations. Still, our approach identifies 31% of the deleted files as potentially unnecessary, which shows that even simple approaches (like ours) are able to identify unnecessary code.

The results of the evaluation on usage data were perfect for two study subjects: no executed files were recommended as unnecessary code. Nonetheless, the recommendations for Teamscale were less precise. A reason for that might be that Teamscale's code base is well-maintained and regularly cleaned up, according to the developers. This is also why the number of recommendations was smaller: only much smaller chunks were identified by the analysis. As already discussed, dynamic dependencies also resulted in incorrect classifications. Although unused code does not necessarily imply unnecessary code, it is remarkable that our approach recommended so many unused files.

Four of five developer teams were able to identify unnecessary code using our tool, and three of them decided shortly after our interviews to actually delete it. This way, at least 20% of our recommendations were deleted from the code base. Given that the approach works only on statically available information (which has the advantage of immediate feedback), we consider this a good cost-benefit ratio.

The purpose of our tool is to provide developers starting points when they need to identify unnecessary code. Participants of our survey confirmed this use case and stated that our analysis implementation was very helpful in identifying unnecessary code. In some cases, they did not even know that the recommended code existed in their code base. False positives were usually identified quickly as such, so the effort spend on false positives was negligible. Overall, the developers rated the precision well enough and considered it worth the effort to investigate the recommendations.

The oracles show mixed results, which is in their very nature. Cleanups take only unnecessary code into consideration that was actually removed by developers. For large code bases, we expect that only a small fraction of unnecessary code will be deleted during the life cycle of the software. Usage data is valuable for identification of unnecessary code and our evaluation. Unfortunately, it is hard to obtain meaningful data (which also motivates our static approach). The developers of BIS 2 and BIS 3 (we used only usage data from these systems in our evaluation) are responsible for very large code bases that are evolving comparably quickly. They were not confident enough to decide whether given code is unnecessary, which confirms previous findings [7]. Nevertheless, in our case, the developers participating in our interviews were nearly always confident about their categorization, especially when they actually deleted unnecessary code.

In most cases, all oracles indicate that stable and decentral code is a good candidate for unnecessary code. More importantly, developers were able to identify unnecessary code using our approach that they actually removed from their code base.

Research Question 3.

Eclipse detects only unused private methods and none of the classes that we analysed had such unused code. This is why no class was reported as dead code. From our point of view, the tooling of Eclipse is useful to identify some unnecessary methods; for the identification of unnecessary classes or even packages, however, the tooling of the IDE is insufficient.

UCDetector is a tool specialized on the identification of dead code like unused classes. Hence, it was no surprise that the Eclipse plugin performed much better in identifying unnecessary classes than the IDE itself: 24 of 76 dead classes were indeed classified as unnecessary by developers. Most of the 52 dead classes that were actually necessary were loaded dynamically during runtime, so the static reference detection of the plugin could not find these dependencies. This is similar to the limitations of our approach, which we discuss for RQ 5. Moreover, there were 38 classes classified as reachable but still unnecessary, 18 more than we identified in our manual investigation. The reason for this observation is that UCDetector did not detect all clusters of unnecessary code where there were no references but from within the cluster. This is why we report 38 reachable unnecessary classes identified by UCDetector and only 18 of these classes in our manual investigation.

In contrast to the evaluation results—where no dead classes were classified as necessary—we know from experience with customer code bases that dead code is not necessarily unnecessary. For example, new features may be implemented in the main development branch and be intentionally unreachable so that no one executes code that is still work in progress. That is, unnecessary code cannot be detected reliably by dead code detectors alone.

The results for RQ 3 also put the precision results of our approach (RQ 2) in perspective. We consider the problem of unnecessary code detection as even harder than dead code detection, because code may also be unnecessary even if it is still reachable. Developers agree with 36% of our recommendations, showing that our approach generates at least similarly good results as dead code detectors. But, in contrast to dead code detectors, our approach takes reachable code into consideration and, this way, provides additional and helpful information to maintainers.

Research Question 4. All developers (except from Jenkins) found, at least, one recommendation useful enough to eventually delete the recommended code. In two different projects, the developers considered it worth to discuss the recommendations within their development team. Our study showed that even the most experienced developers do not know the whole code base and are therefore unsure. In such cases, it is better to discuss with several team members whether the code is still relevant.

For platform projects or APIs, which aim at high extensibility, such as Jenkins, it is harder to decide which code is unnecessary—at least, when the visibility of classes is not limited. Therefore, additional factors would need to be considered to find unnecessary code: class visibilities (publicly visible code may have hidden external dependencies) or manifest exports could be taken into account and corresponding dependencies needed to be resolved. In such circumstances, we expect our approach to be even more precise in identifying unnecessary code for highly extensible software projects.

Research Question 5. Our results indicate that a major reason for false positive recommendations are unknown dependencies, which lead to an underrated centrality value. For example, dependency injection is becoming more and more popular [35]; however, it is difficult or even infeasible to resolve the arising dependencies using static code analysis—which was to be expected and is a

drawback that our approach has in common with state-of-the-art dead detectors like UCDetector. Therefore, to increase the precision of the recommendations, a possible future improvement would be to compute cross-language dependencies and to take non-code files into account, such as configuration files specifying dependencies. This highly system-dependent tailoring is feasible for practitioners that are experts of their software systems. For our evaluation, however, this would have required deep knowledge about the architecture of all our study subjects.

7.2 General Discussion

Lessons Learned. Unnecessary code is difficult to identify as the necessity of a piece of code cannot be deduced from the code alone (except the special cases discussed below). We assume that deleted code represents a lower bound for unnecessary code because developers were so confident to delete it but probably not all unnecessary code gets deleted. Furthermore, we assume that unused code represents an upper bound for unnecessary code because used code is necessary and not every unused piece of code is unnecessary.

Human judgment about the necessity of a piece of code is ambivalent: On the one hand, we consider it mandatory because static usage analysis might miss information and therefore produce false positives which should be cross-checked. Similarly, the results of dynamic usage analysis do not identify unnecessary code as unused does not always imply unnecessary. On the other hand, we consider it problematic because the opinion of developers might deviate from reality or developers lack knowledge about a piece of code and its necessity. Hence, we suggest to use a combination of human judgment and tool support when analyzing and handling unnecessary code, especially if developers are not sure about its necessity. Tool support could include profiling over a specific period of time to determine whether the code is still in use and reminders about potentially unnecessary code to ease management of many such code pieces.

There are different strategies to handle unnecessary code, depending on the developers confidence and the project context: deletion of the code from the code base, movement of the code to a special package (to signal that it is probably unnecessary), deactivation of the code (i.e., making code unexecutable without deleting it, e.g., by adding a (log) message and returning immediately at the entry point or performing no immediate action but ignoring the code in future migrations).

In several instances, we observed developers not deleting unnecessary code. While sometimes the reason was surely that they did not have enough knowledge to reliably judge whether the code was truly unnecessary, we hypothesize that sometimes the risk of code deletion was bigger than the benefit of it. That is, developers get little acknowledgment for deleting code but get into big trouble when deleted code is still necessary. Hence, we suggest to study benefits and risks of code deletion in more detail in future work.

Aggregation and filtering of unnecessary (or unused) code, for example, from file to package level, for presentation to developers is a difficult task because it has to consider conflicting goals: on the one hand, the number of recommendations presented to developers should be minimized to ease human analysis, implying aggregation to big chunks; on the other hand only unnecessary code pieces should be recommended to ease interpretability, implying no aggregation.

We see two types of static analysis regarding code necessity. The first type is *analytical* static necessity analysis which can reliably discriminate between necessary and unnecessary code. Examples are dead code analysis (a piece of code is not reachable and therefore unnecessary), reachability analysis for project files (a source file is not included in any project file and therefore its code is unreachable and unnecessary) and platform-incompatibility analysis (a piece of code does not support the execution platform and is therefore not executable and unnecessary, for example, when code does not support Unicode on a Unicode-based platform). The second type is *heuristic-based*

static necessity analysis which uses heuristics to discriminate between necessary and unnecessary code pieces. Our approach is an example as it uses stability and decentrality as heuristics.

We see the following implications for researchers, practitioners and tool vendors.

Implications for Researchers. Study which limitations of static usage analysis are systematic and which can be overcome by extending the approach, design and evaluate strategies for handling (potentially) unnecessary code, design and evaluate strategies for aggregating unnecessary (or unused) code and study the benefits and risks of code deletion.

Implications for Practitioners. Use analytical static necessity analysis whenever possible to exploit its fast analysis, to use the recommendations of our approach as starting points for further–human or dynamic–analysis of code necessity.

Implications for Tool Vendors. Implement analytical static necessity analysis, add tool support for handling code with unclear necessity, for example follow-up reminders, and implement a mechanism for fast (de-)activation of code pieces.

7.3 Threats to Validity

Construct Validity. A threat to construct validity of our evaluation and especially *RQ 2* is that there is no ground truth for unnecessary code. To mitigate this risk, we employ three oracles that indicate whether recommendations for unnecessary code are true: cleanups, usage data, and developer interviews. Cleanups spot unnecessary code that was actually deleted. Cleanups as oracle do not categorize non-deleted code into unnecessary and necessary, though. Usage data reveal code that was not executed, which we claim to be likely unnecessary, if usage was recorded over a sufficient time span in a representative context. Nevertheless, there may be good reasons for code not being executed but still being necessary (e.g., disaster recovery code if no disasters occurred in the recorded time). Developers can help to identify unnecessary code if they know their code base very well. Previous studies (e.g., by Juergens et al. [17]) have shown that developers' expectations do not always meet the de facto necessity of code. That is why we decided to rely in our evaluation not only on developer feedback but also on other oracles. Nevertheless, participants from several projects were confident that recommended code was unnecessary and therefore removed it from the code base. Using three oracles makes it possible to take different perspectives on unnecessary code, which strengthens the validation of our recommendations.

Internal Validity. To answer *RQ 3*, we considered classes as dead code if Eclipse detected all methods and fields in that class as unused code. So, if there were classes that were only partially recognized as unused code, we considered them as reachable. In general, this would mean that we put Eclipse's dead code detection in a poor light if many unused fields and methods were identified and only a few were missed. However, Eclipse reported no unused method or field in any investigated class. That is, we see no threat to validity because of our classification design. In *RQ 5*, the list of typical characteristics for false positive recommendations is not meant to be complete. The list only contains the most common characteristics mentioned by developers, and we got developer feedback for only some systems. Therefore, this list is primarily intended to give an indication for which further factors need to be taken into consideration to reduce the number of false positives. Further developer feedback and more systems would be necessary to allow generalizability. The main purpose of RQ 5 is identifying aspects of our approach that can be addressed to further improve the recommendation quality (see also Section 8).

External Validity. Generalizability is a common issue in software-engineering research because software systems vary in a lot of parameters [31]. In addition, the age of the system, the number of developers, their development experience, and the development process of the project might influence the emergence of unnecessary code. Our mitigation strategy of this threat was to select both open- and closed-source systems (which may apply different development processes) from

various domains with different sizes and written in different languages. As a result of the evaluation on this diverse set of software systems, we have seen that our approach works. This is why we are confident that our approach can also be applied on other software systems to identify unnecessary code.

8 CONCLUSION AND FUTURE WORK

Unnecessary code wastes resources in many ways and can cause superfluous costs (e.g., when certifying or migrating code). Dynamic analysis can be used to identify unnecessary code, which often comes at the cost of recording representative usage data. In this work, we evaluated to what extent a simpler and cheaper static analysis approach is able to identify unnecessary code. The key hypothesis is that stable and decentral code is likely unnecessary.

In our evaluation, we used three oracles to investigate whether a static approach is actually able to identify unnecessary code: cleanups, usage data, and developer interviews indicate whether corresponding recommendations point indeed to unnecessary code. We used 14 open-source and closed-source projects from various domains, written in different languages, with a long development history. Our evaluation results show that unnecessary code, which has already been removed, is rather stable and decentral. 31% of the deleted files of the investigated cleanups were identified as potentially unnecessary. Compared to a random selection strategy, our tool was for two out of three study subjects significantly better in identifying non-executed code. Finally, our interviews with, in total, 25 developers show that 34% of the recommendations of our tool point to actually unnecessary code. 29% of unnecessary classes are reachable, so, they cannot be detected by dead code detectors. Moreover, the dead code detection plugin UCDetector identified only 32% of unnecessary code as dead. Overall, developers deleted 10 out of 50 discussed code fragments after the interview. This underlines the confidence of their statements and emphasizes the usefulness of our approach in practice.

In our study, we identified reasons for false positives of our approach, in particular, implicit dependency information that cannot be retrieved from the code, directly. In the future, we plan to overcome some of the limitations caused by missing information about dynamic dependencies. In general, other types of dependencies that cannot be recognized statically could be approximated. For example, one could implement a heuristic that checks if any artifact in the code base references a file name (or class identifier) and add a corresponding dependency to the dependency graph. This way, it would be easier to represent dependencies from non-code artifacts, code in other programming languages, or injections.

In this work, we focused on unnecessary code from a development and maintenance perspective. It would be interesting to see whether similar approaches help test developers to focus their test effort on relevant parts of the software system.

ACKNOWLEDGMENTS

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant “SOFIE, 01IS18012A”. Apel’s work has been supported by the German Research Foundation (AP 206/11-1). The responsibility for this article lies with the authors.

REFERENCES

[1] I. Ahmed, U. A. Mannan, R. Gopinath, and C. Jensen. 2015. An Empirical Study of Design Degradation: How Software Projects Get Worse over Time. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–10.

[2] G. Canforaand L. Cerulo and M. Cimitileand M. Di Penta. 2014. How changes affect software entropy: An empirical study. *Empirical Software Engineering* 19, 1 (2014), 1–38.

[3] I. Şora. 2015. A PageRank based recommender system for identifying key classes in software systems. In *Proceedings of the International Symposium on Applied Computational Intelligence and Informatics*. IEEE, 495–500.

[4] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. 2010. Flexible Architecture Conformance Assessment with ConQAT. In *Proceedings of the International Conference on Software Engineering*. ACM, 247–250.

[5] F. Dreier. 2015. Detection of Refactorings. Bachelor's thesis, Technical University of Munich. Retrieved October 18, 2019 from https://www.cqse.eu/publications/2015-detection-of-refactorings.pdf

[6] S. Eder, H. Femmer, B. Hauptmann, and M. Junker. 2014. Which Features Do My Users (Not) Use?. In *Proceedings of the International Conference on Software Maintenance and Evolution*. IEEE, 446–450.

[7] S. Eder, M. Junker, E. Juergens, B. Hauptmann, R. Vaas, and K. H. Prommer. 2012. How much does unused code matter for maintenance?. In *Proceedings of the International Conference on Software Engineering*. IEEE/ACM, 1102–1111.

[8] M. Eichberg, B. Hermann, M. Mezini, and L. Glanz. 2015. Hidden Truths in Dead Software Paths. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 474–484.

[9] A. M. Fard and A. Mesbah. 2013. JsNose: Detecting JavaScript Code Smells. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, 116–125.

[10] T. Gee. 2016. Unused Code Detection in IntelliJ IDEA 2016.3. Retrieved October 18, 2019 from https://www.youtube.com/watch?v=43-JEsM8QDQ

[11] R. Haas. 2017. Identification of Unnecessary Source Code. Master's thesis, Technical University of Munich.

[12] L. Heinemann, B. Hummel, and D. Steidl. 2014. Teamscale: Software Quality Control in Real-time. In *Proceedings of the International Conference on Software Engineering*. ACM, 592–595.

[13] Spieler J. 2019. UCDetector: Unnecessary Code Detector. Retrieved October 18, 2019 from http://www.ucdetector.org/

[14] D. Jannach (Ed.). 2011. *Recommender systems: An introduction*. Cambridge University Press.

[15] Y. Jiang, D. Wu, and P. Liu. 2016. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *Proceedings of the Annual Computer Software and Applications Conference*. IEEE, 12–21.

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. 2009. Do Code Clones Matter?. In *Proceedings of the International Conference on Software Engineering*. IEEE, 485–495.

[17] E. Juergens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, and K. H. Prommer. 2011. Feature Profiling for Evolving Systems. In *Proceedings of the International Conference on Program Comprehension*. IEEE, 171–180.

[18] J. Krinke. 2008. Is Cloned Code More Stable than Non-cloned Code?. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation*. IEEE, 57–66.

[19] J. Krinke. 2011. Is Cloned Code Older Than Non-cloned Code?. In *Proceedings of the International Workshop on Software Clones*. IEEE, 28–33.

[20] M. M. Lehman and L. A. Belady (Eds.). 1985. *Program evolution: Processes of software change*. Academic Press Professional.

[21] S. B. Maurer. 2014. Directed Acyclic Graphs. In *Handbook of graph theory*. CRC Press, 180–195.

[22] M. Mondal, C. K. Roy, Md. S. Rahman, R. K. Saha, J. Krinke, and Schneider K. A. 2012. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the Annual Symposium on Applied Computing*. ACM, 1227–1234.

[23] R. Moser, W. Pedrycz, and G. Succi. 2008. A Comparative Analysis of the Efficiency of Change Metrics and Static Code Attributes for Defect Prediction. In *Proceedings of the International Conference on Software Engineering*. ACM, 181–190.

[24] J. C. Munson and S. G. Elbaum. 1998. Code churn: A measure for estimating the impact of code change. In *Proceedings of the International Conference on Software Maintenance*. IEEE, 24–31.

[25] D. L. Parnas. 1994. Software aging. In *Proceedings of the International Conference on Software Engineering*. IEEE/ACM, 279–287.

[26] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel. 2019. BinTrimmer: Towards Static Binary Debloating Through Abstract Interpretation. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer, 482–501.

[27] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann (Eds.). 2014. *Recommendation Systems in Software Engineering*. Springer.

[28] P. Runeson and M. Höst. 2008. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2008), 131–164.

[29] G. Scanniello. 2014. An Investigation of Object-Oriented and Code-Size Metrics as Dead Code Predictors. In *Proceedings of the EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 392–397.
[30] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the International Conference on Automated Software Engineering*. ACM, 329–339.
[31] J. Siegmund, N. Siegmund, and S. Apel. 2015. Views on Internal and External Validity in Empirical Software Engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE, 9–19.
[32] D. Steidl, B. Hummel, and E. Juergens. 2012. Using Network Analysis for Recommendation of Central Software Classes. In *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 93–102.
[33] F. Streitel, D. Steidl, and E. Jürgens. 2014. Dead Code Detection on Class Level. *Softwaretechnik-Trends* 34, 2 (2014).
[34] Unknown. 2019. Reduce your app size. Retrieved October 18, 2019 from https://developer.android.com/topic/performance/reduce-apk-size#remove-unused
[35] H. Y. Yang, E. Tempero, and H. Melton. 2008. An Empirical Study into Use of Dependency Injection in Java. In *Proceedings of the Australian Conference on Software Engineering*. IEEE, 239–247.
[36] A. Zaidman and S. Demeyer. 2008. Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance and Evolution: Research and Practice* 20, 6 (2008), 387–417.