Poster: Recommending Unnecessary Source Code Based on Static Analysis

Roman Haas
CQSE GmbH
Munich, Germany

Rainer Niedermayr
University of Stuttgart, CQSE GmbH
Stuttgart, Germany

Tobias Röhm
CQSE GmbH
Munich, Germany

Sven Apel
Universität Passau
Passau, Germany

***Abstract*—Grown software systems often contain code that is not necessary anymore. 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 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 static 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 of our approach with historical cleanup actions, runtime usage data, and feedback from 25 developers of 5 software projects. Our study shows that recommendations generated from stability and centrality information point to unnecessary code. Our results suggest that static analysis can provide quick feedback on unnecessary code that is useful in practice.**

I. 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 [1]–[3]. 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 [4].

Unnecessary code wastes resources. In particular, it becomes cost-intensive if the whole code base needs to be modified, for example, during a migration to a new technology. In daily development work, it wastes computing resources during compiling and testing it, which slows down feedback from continuous integration pipelines to developers. Moreover, from a management perspective, maintenance efforts should not be invested into unnecessary code.

We investigate a light-weight 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.

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. (2) We check whether code recommended as unnecessary was not used in production environments. (3) 25 developers reviewed recommendations of unnecessary code in a series of *interviews*.

II. 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. Of course, there are other cases of stable code, for example, core concepts and features. 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 are often necessary and can be identified statically [5], [6]. 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.

III. Study

We have implemented our approach as a recommender system that suggests 10 files or packages as unnecessary code and evaluated it on 14 open-source and closed-source software systems. The overarching question of our study is whether our approach is able to make practically relevant recommendations for unnecessary code.

A. Research Questions

RQ 1: 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 accuracy of recommendations of our approach by comparing them to historical cleanups, usage data, and feedback from a series of developer interviews.

RQ 2: Do developers delete code recommended as unnecessary? If developers follow our suggestion and delete recommended code, this underlines the usefulness of our approach.

TABLE I
RQ 1: EVALUATION OF RECOMMENDATIONS FOR POTENTIALLY UNNECESSARY FILES USING USAGE DATA ORACLE

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

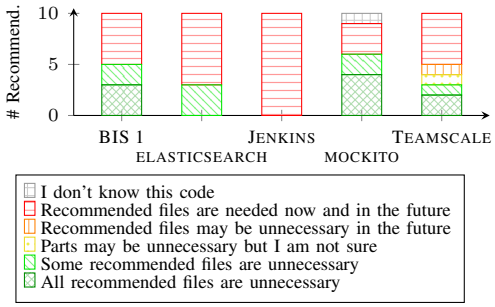

Fig. 1. Developer feedback on recommendations

B. Study Results

In this section, we report the results for the research questions.

RQ 1. Identification of Unnecessary Code: For simplicity, we separate the results for the three oracles.

Results with cleanups as oracle. 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, but only 5.5% of the deleted files were recommended as unnecessary.

Results with usage data as oracle. Next, we compare recommendations with runtime usage data obtained from three of our study subjects. Table I 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 recommended files were executed, which is a perfect result. The recommendations for the study subject TEAMSCALE contained 12 files (36%) that deemed unnecessary but were executed and are therefore very likely necessary.

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 ($p > 0.05$).

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 1. In total, 50 recommendations 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.

RQ 2. Deletions of Unnecessary Code by Developers: 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.

IV. CONCLUSION

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 paper, 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.

Our evaluation has shown that our recommendations refer to unused code in 64%–100% of all cases. The developer interviews revealed that 34% of recommendations pointed to unnecessary code that was even still reachable and therefore would not have been spotted by, for example, 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.

ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant "SOFIE, 01IS18012A". The responsibility for this article lies with the authors.

REFERENCES

[1] M. M. Lehman and L. A. Belady, Eds., *Program evolution: Processes of software change*. Academic Press Professional, 1985.
[2] D. L. Parnas, "Software aging," in *Proceedings of the International Conference on Software Engineering*. IEEE/ACM, 1994, pp. 279–287.
[3] G. Canfora, L. Cerulo, M. Cimitile, and M. Di Penta, "How changes affect software entropy: An empirical study," *Empirical Software Engineering*, vol. 19, no. 1, pp. 1–38, 2014.
[4] S. Eder, M. Junker, E. Juergens, B. Hauptmann, R. Vaas, and K. H. Prommer, "How much does unused code matter for maintenance?" in *Proceedings of the International Conference on Software Engineering*. IEEE/ACM, 2012, pp. 1102–1111.
[5] D. Steidl, B. Hummel, and E. Juergens, "Using network analysis for recommendation of central software classes," in *Proceedings of the Working Conference on Reverse Engineering*. IEEE, 2012, pp. 93–102.
[6] I. Şora, "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, 2015, pp. 495–500.