Dead Code Detection On Class Level

Fabian Streitel, Daniela Steidl, Elmar Jürgens

CQSE GmbH, Garching bei München, Germany
{streitel, steidl, juergens}@cqse.eu

Abstract

Many software systems contain usused code. While unused code is an unnecessary burden for maintenance, it is often unclear which parts of a modern software system can actually be removed. We present a semi-automatic, iterative, language-independent approach to identify unused classes in large object-oriented systems. It combines static and runtime information about an application and aids developers in identifying unused code in their system with high precision. A case study on three real-life systems shows its effectiveness and feasibility in practice.

1 Introduction

Unused code in software systems can be problematic, as Eder et al. noted in [2], since it has to be maintained by the developers along with the actually used code. Therefore, it creates an unnecessary maintenance overhead that could be avoided, were it known which parts of the system are no longer necessary.

Unfortunately, determining if code is inused is in general undecidable statically [3], due to the way programming languages evolved over the years. Features such as inheritance and virtual method calls make it impossible to know statically which exact pieces of code will be executed at runtime when a certain method is called. Furthermore, reflection allows not only to construct classes at runtime which are not referenced in the source code at compile time, but also to call arbitrary methods on objects the same way. Static analysis alone is thus not sufficient to solve the problem.

Using only dynamic information instead is, in our view, also not an adequate solution. Firstly, as stated in [2], profiling an application to obtain such information has a performance impact. Furthermore, to get an accurate picture of the usage of classes, a system has to be profiled for a long time. Secondly, even runtime traces that were collected over such extended periods of time may not cover all used classes, due to e.g. exception handling mechanisms that were not triggered or important features that were by chance not used in the considered time period, e.g. since they are only necessary once every year.

In this paper, we propose a semi-automatic, language-independent, iterative approach that combines static and dynamic information. The needed dynamic information can be obtained in a relatively short amount of time and with little overhead. With a case study we show that this analysis can with high precision identify unused classes in large systems.

2 Approach

Our approach consists of a manual iterative procedure that is aided by tool support. It works on the class-level to identify classes that can be removed entirely from the system. We use static analysis to create a class dependency graph of the system, including the known entry points of the source language, e.g. main methods in Java. With this data, we can compute a set of definitely used classes by finding all classes that are transitively reachable in the dependency graph from at least one entry point. All other classes are *possibly* unused.

Some of these classes may, however, be loaded at runtime via a reflection mechanism. Each time a class is loaded this way, this corresponds to a missing link in the dependency graph between the class that performs the loading and the class that is being loaded. To increase the precision of our analysis, we try to identify these links in the next step and improve our dependency model with them.

This requires knowledge about how the class loading mechanisms work. A developer can often supply this information directly. Large systems can, however, employ many different such mechanisms and it is therefore easy to forget some. Thus, we use runtime data to assist in this step. During the execution of the application, we gather a list of all classes that contain at least one method that was executed. These classes are obviously used. Gathering this list can be achieved easily with a profiler. Techniques such as ephemeral profiling [4] even allow such profiling on production systems without a major performance impact. Furthermore, even running such a profiler on a test system may be sufficient to record the necessary information.

If we compare this list to the list of classes that are statically not reachable from an entry point, we get a list of classes that were loaded via reflection. For at least one such class, the developer must manually search the source code for the mechanism that loaded it.

Once it is found, the information which classes may be loaded by that mechanism can be fed back into the static analysis. All of these classes are simply treated as entry points and reachability is recalcluated. Starting again with comparing the resulting list of unused classes to the list of classes executed at runtime, the iterative procedure begins anew. The list of possibly unused classes is thus narrowed down in each step, until no more classes remain that were executed at runtime and are identified as unreachable statically.

Note that our approach cannot find classes that are unused but are still referenced in the source code, e.g. if all references to the class are enclosed within `if (false)`. This would require a more in-depth analysis of the source code, e.g. with a data flow analysis.

We implemented our approach in Java using the quality analysis engine ConQAT [1]. It facilitates the analysis of quality characteristics of a software system using a pipes and filters approach.

3 Evaluation

To evaluate our approach in practice, we performed the analysis on three different systems: JabRef[1] (81 kLOC), an Open Source Java reference manager; ConQAT (191 kLOC), which we had used to implement the approach; A business information system written in C# at Munich Re Group (360 kLOC). For these systems, we answered the following research questions:

How many different mechanisms for reflection do we find? In JabRef we identified 2, in ConQAT 3 different class loading mechanisms, and 7 different ones in the business information system.

How much unused code do we find with our approach? We found that among the three tested systems, unused code as identified by our approach ranged between 1.7 for JabRef and 9 percent of the system size for the business information system. This corresponded to between 2 and 22 percent of the classes of those systems, since unused classes are often shorter than used ones.

What are the precision of our approach? To answer this question, we compiled a sample of classes from ConQAT's code base, which our approach had identified as used or unused. We showed them to a group of 6 ConQAT developers and asked them to rate these as correct or incorrect findings. Using the false positives (classes wrongly categorized as unused), we calculated a precision of 72 percent.

How much of the unused code can actually be removed from the system? There may be good reasons for keeping unused code in a system. Therefore, the amount of code that can be removed from a system is not equal to the amount of unused code. We asked the same ConQAT developers whether a class they had previously identified as unused could also be deleted. For about half of those classes, the study subjects answered positively. Reasons for keeping a class included a possible future use of the class, e.g. for debugging, or that another system, which was not considered in the analysis, used that code.

[1] `http://jabref.sourceforge.net/`

4 Conclusion

We presented a language-independent, iterative, semi-automatic approach to detect unused classes in software systems which combines static information obtained from the source code and binaries of a system with dynamic information obtained during its execution. Our analysis also deals with the problems posed by the use of reflection in modern software systems.

Our studies showed that real-life software systems can contain a large number of such class loading mechanisms, making it hard to identify unused code without tool assistance. Runtime information about the analyzed application can help with the identification and improve analysis results.

In a case study, our approach categorised up to 9% of the analysed systems' code base as unused. Of these classes, about half could actually be removed from the affected system, saving about 16 kLOC. We therefore estimate that for large systems, a significant amount of code can be removed after applying our analysis.

Due to its semi-automatic nature, however, the approach cannot replace an expert as he is still needed to accurately detect all reflection mechanisms used in the application. In the future, more accurate runtime information and language specific knowledge could be used to further automate this step, giving the developer more guidance as to where the reflection occurrs in the source code.

5 Acknowledgement

We would like to thank Sebastian Eder, Moritz Beller and Thomas Kinnen for their helpful comments, the ConQAT team for participating in the study, as well as the Munich Re Group for providing one of their systems as a study object.

References

[1] F. Deissenböck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *STEP '05*, 2005.

[2] S. Eder, M. Junker, E. Jürgens, B. Hauptmann, R. Vaas, and K. Prommer. How much does unused code matter for maintenance? In *ICSE '12*, 2012.

[3] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical Experience with an Application Extractor for Java. In *OOPSLA '99*, 1999.

[4] O. Traub, S. Schechter, and M. D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Harvard University, 2000.