Static Bug Detection Through Analysis of Inconsistent Clones

Elmar Juergens, Benjamin Hummel, Florian Deissenboeck, Martin Feilkas

Institut für Informatik
Technische Universität München
{juergens,hummelb,deissenb,feilkas}@in.tum.de

Abstract. Existing software systems contain a significant amount of duplicated code. Such redundancy can negatively impact program correctness, since inconsistent updates to duplicated code fragments are prone to introduce subtle bugs. This paper outlines our work-in-progress to statically detect inconsistencies in duplicated code fragments in order to find clone-related bugs. We illustrate the problem of clone related bugs with the example of such a bug in Eclipse, outline our algorithm for detecting inconsistencies in clones and report initial experiences from an industrial case study conducted with the Munich Re Group.

1 Introduction

Existing research in clone detection indicates that software systems typically contain a significant amount of duplicated code. Of all lines of code in the respective system, 8,7% of GCC [DRD99], 19% of X Windows [Ba95], 22,7% of Linux and 29% of JDK [Li06] have been found to be part of at least one duplication. Code duplication can be observed independent of programming language, target platform or application domain [Ko07].

Cloning renders maintenance difficult, since changes to one instance of a duplication (clone) potentially affect the other clones as well. Or, if a bug is contained in a duplicated code fragment, it needs to be removed from all its clones. If one or more clones are overlooked by the developer while he performs the change operation, subtle bugs can be introduced, or known bugs can remain in a system. In response to this, numerous clone detection approaches and tools that help to identify regions of duplicated code have been proposed by the research community [Ko07]. Such tools are valuable in reducing the risk of introducing inconsistencies during maintenance of duplicated code, since they help programmers in identifying all potentially affected code fragments during a change operation. Their shortcoming is that they cannot detect clone-related bugs once they have entered a system. To mitigate this, we outline an analytic approach that helps to detect faults causing such bugs contained in a system.

Contribution: We illustrate the risk cloning poses to program correctness and propose an approach to statically detect clone-related bugs. Our approach is applicable to both modern and legacy programming languages, for which few bug detectors exist. We show preliminary results from an industrial case study that indicate that we can detect a significant number of defects in practice.

2 Detecting Inconsistently Evolved Clones

A clone is a sequence of statements that can be found more than once in a software system[1]. An inconsistent clone is a sequence of statements that would become a clone, if we inserted or deleted a small number of statements. An example of this is illustrated in Figure 2 that displays a code fragment from Eclipse 3.2.2, in which the statement `target.appendChild(export)` is missing on the left side[2]. As can be seen, the fragments of an inconsistent clone that are separated by gaps (insertions or deletions), are shorter clones themselves.

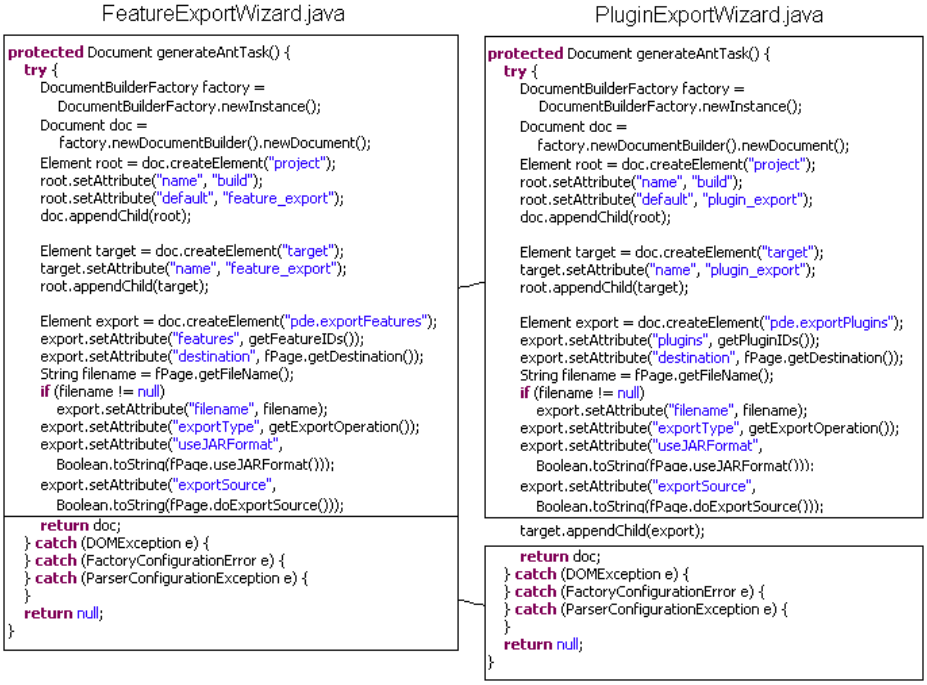

Fig. 1. Bug 155070 in Eclipse 3.2.2

This observation leads to a first simple algorithm for finding inconsistent clones: Firstly, it determines all consistent clones using one of the existing clone detection algorithms. Then it looks for pairs of clones which, together with their siblings, are sufficiently close to each other. These are then assembled into a single inconsistent clone. We implemented this detection algorithm as part of our open source software *Continuous Quality Assessment Toolkit ConQAT* [DPS05] that implements an extensible clone detection tool. Due to the language independence of the inconsistency detection, our implementation can be used with all languages that ConQAT supports, namely C/C++, Java, VB, C#, PL/I and Cobol.

[1] Typically minor differences in identifier naming or literal values are ignored.

[2] The missing statement results in loss of information and leads to incomplete generated XML. It is the cause of Eclipse Bug 155070.

3 Case Study

We have performed a case study of clone-related bugs due to inconsistent clones on an industrial business information system from the financial domain that is developed and maintained at the Munich Re Group. The system comprises 400 kLoC of C#, of which 215 kLoC are handwritten[3].

Setup: We performed the analysis in three steps: Firstly, we applied our tool to detect inconsistent clones. Secondly, we manually inspected the detected inconsistent clones to exclude inconsistencies that obviously do not indicate bugs and categorized the remaining inconsistencies according to their defect probability. Thirdly, the remaining inconsistencies are currently being inspected by the developers of the system under investigation.

Results: The detection produced 106 inconsistent clones (comprising 272 code fragments) and took approx. 30s and 40MB on a 1.7 GHz notebook. Manual investigation identified 67 inconsistent clones to be false positives, typically due to purely syntactical differences (e.g. inconsistent use of braces surrounding single statement conditionals and loops, order of commutative statements or use of local or fully qualified names). Of the remaining 39 inconsistencies (comprising 89 code fragments), 11 could be classified as bugs for technical reasons due to missing or incomplete null checks or inconsistent use of break statements that cause different behavior of the same algorithm in different places of the system. 8 inconsistencies due to unused or commented code were classified as style issues that, although they do not present defects at present, should be mended in order to simplify future maintenance. The remaining 20 inconsistencies were mainly caused by differences in conditions or missing method calls and cause variations in the system's behavior. More detailed knowledge about the system than available to us is required to decide whether they are desired or erroneous. They are still under investigation by the system's developers, but preliminary results are very promising.

Discussion: Although the evaluation result of 20 inconsistent clones by the system developers is still pending, the 11 bugs and 8 style issues identified by our analysis already indicate a substantial value of our analysis in practice. The resulting precision of 18%-38% (depending on the 20 pending assessments) can, while acceptable, still be improved by ignoring purely syntactic inconsistencies such as those mentioned above.

The current analysis approach still has drawbacks, since it only detects inconsistencies with a single gap that are encased by sufficiently large clones. Hence, we might miss relevant inconsistencies that are encased by clones that are too small to be discovered by the initial consistent clone detection phase. We plan to improve on this by applying an algorithm that directly detects gapped clones in our future work. The parameters for minimal clone and maximal gap size heavily influence precision and recall of detection results. Our choice of 5 as threshold for both for our case study is based on our experience gained in early experiments and demands further research.

[3] The remaining code is generated and has been excluded from the analysis, since it is not maintained by hand and thus cannot suffer from inconsistent manual changes.

4 Related Work

A variety of approaches and tools that search for bugs by performing different types of static program analyses—such as syntactic pattern matching or data flow analysis—has been proposed, see e.g. [RAF04,Zh06] for a survey. Our approach is complementary to them in that it detects bugs that typically cannot be detected with existing tools, since it uses duplication inconsistencies as additional information source unavailable to them. A large body of research exists that deals with various aspects, i. e. origin, evolution, detection and removal of cloning. Due to space constraints, please refer to Koschke's comprehensive survey [Ko07]. Approaches to detect gapped clones exist [Ko07], but are not targeted at bug detection. To our knowledge, only two approaches exist that employ clone detection to detect bugs. In [Li06], Li et. al. search for inconsistently renamed variables to detect clone-related bugs. In [JSC07], Jiang et. al. extend that approach by searching for different contexts of duplicated code fragments that potentially indicate bugs. Both approaches focus on detecting bugs that have been introduced during clone creation. In contrast, our approach focuses on detection of bugs introduced by inconsistent evolution of clones, which we consider the bigger threat to program correctness in long lived software systems.

5 Future Work

Empirical Analysis: We plan to conduct a comprehensive industrial case study to better understand the significance of clone-related bugs in practice.

Algorithm: We are currently developing an extension of suffix-tree based clone detection that directly detects inconsistencies in clones. We expect this approach to yield better completeness than our current one.

Increasing precision: We plan to increase the precision of our approach to detect clone-related bugs by ignoring purely syntactic inconsistencies and by exploiting evolution information from the version control system.

References

[Ba95] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of WCRE 1995*.

[DPS05] F. Deißenböck, M. Pizka, T. Seifert. Tool Support for Continuous Quality Assessment. In *STEP 2005*.

[DRD99] S. Ducasse, M. Rieger, S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of ICSM 1999*.

[JSC07] L. Jiang, Z. Su, E. Chiu. Context-Based Detection of Clone-Related Bugs. In *FSE 2007*.

[Ko07] R. Koschke. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Proceedings, 2007.

[Li06] Z. Li, S. Lu, S. Myagmar, Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale SW. *IEEE TSE*, 2006.

[RAF04] N. Rutar, C. B. Almazan, J. S. Foster. A Comparison of Bug Finding Tools for Java. In *Proceedings of ISSRE 2004*.

[Zh06] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J.P. Hudepohl, M.A. Vouk. On the value of static analysis for fault detection in SW. *IEEE TSE*, 2006.