

Code Clone Detection in Practice

Florian Deissenboeck, Benjamin Hummel, Elmar Juergens
Institut für Informatik, Technische Universität München
Garching b. München, Germany
{deissenb,hummelb,juergens}@in.tum.de

ABSTRACT

Due to the negative impact of code cloning on software maintenance efforts as well as on program correctness [4–6], the duplication of code is generally viewed as problematic. However, the techniques and tools developed by the research community in the last decade have not found broad acceptance in software engineering practice yet. This tutorial contributes to a more widespread application of existing approaches by illustrating where cloning comes from, what its consequences are, and how it can be detected.

Categories and Subject Descriptors

D.2.9 [Software Engineering]: Management—SQA

Keywords

Redundancy, Clone Detection

1. CODE CLONING

While there are different definitions of the term *code clone*, this tutorial assumes that a *clone* is a duplicated, and then potentially modified, piece of code. The duplication of code has the following ramifications:

First, it increases program size and thereby raises the efforts for all activities that are affected by the size of a system. This includes inspections, compilation, test and various other maintenance activities. Many real-world software systems exhibit a cloning-induced size increase of more than 10%. However, size increases of more than 25% or even 50% are sometimes encountered.

Second, it increases software maintenance efforts as modifications to cloned code usually need to be applied to all copies of the code. Additional efforts are caused by the identification of the copies, the repeated modification as well as the required quality assurance for all modified pieces of code. Currently, there is no reliable data on the actual amount of additional effort caused by cloning. However, it is expected to be proportional to the size increase discussed above.

Third, inconsistent changes to cloned code can lead to unexpected program behavior. A particularly dangerous type of change to cloned code is the *inconsistent bug fix*. If a fault was found in cloned code but not fixed in all clone instances, the system is likely to still exhibit the incorrect behavior. A recent study has shown that inconsistent changes to clones represent faults in 50% of the cases if the change was introduced unintentionally [4].

Code cloning almost always is the result of copy&paste operations carried out by developers. The reasons why developers use these operations so frequently are manifold: Reusing existing code without copying it, often requires developers to apply abstraction mechanisms to generalize functionality. As generalizing code can be a substantial challenge and does not directly contribute to the functionality of a product, developers under schedule pressure often apply copy&paste operations as a very efficient form (on the short term) of reuse. Moreover, developers are sometimes not familiar with the appropriate abstraction mechanisms that are required for reusing existing code without copying it or are unaware of the consequences of code cloning. In some situations, creating proper abstractions would require changing modules a developer cannot change due to technical or organizational reasons. Finally, limitations of the used programming language prohibit the creation of proper abstractions and force developers to apply copy&paste.

2. CLONE DETECTION

Over the last decade a plethora of clone detection algorithms has been proposed and implemented by the research community as well as, in a few cases, by commercial tool providers. Most of them follow a pipeline-like pattern with the following core stages. The problems and questions arising at each of these stages are mostly independent of the actual tool used.

Loading The first stage of the clone detection is responsible for loading the analyzed source code from the hard disc.

Segmentation In the second stage, the source code is broken down into *units*, on which the actual detection is performed. Depending on the detection approach, these units can *e. g.*, be source lines, tokens created by a lexer, parse tree fragments or parts of a program dependence graph.

Normalization Most approaches account for minor modifications between clones by performing a set of normalization operations on the units generated by the last stage. One example, is the abstraction from concrete identifier names by replacing them with suitable substitutes.

Detection The detection stage is responsible for the actual detection of duplication. Depending on the algorithm used, it either compares sequences of units for real equality or allows minor differences between them. There is a wealth of algorithms for this problem, including those based on suffix trees, locality sensitive hashing, or graph search on a program dependence graph.

Post-processing The results of the detection are usually run through a post-processing stage that, among others, filters out clones known to be irrelevant, *e. g.*, because they are too short.

Presentation Finally, the last stage is responsible for presenting the detection results to the user. Depending on the users and their intended goals, different metrics, formats and visualizations are applied here.

Using state-of-the-art clone detection tools, code duplication can be detected in code bases in virtually all languages and of all realistic sizes. However, it must be pointed out, that clone detection is currently limited to finding *clones*, *i. e.*, it is ill-suited to detect redundant functionality that was not cloned but developed independently by different developers [3].

3. CLONE DETECTION IN PRACTICE

The biggest challenges for an effective and efficient application of clone detection in practice are detection precision and the interpretation of the detection results.

Detection Precision When a clone detection tool is run on a system's source code without further configuration, the initial detection results are often disappointing. While the longest clones are usually perceived as interesting, many of the shorter ones are viewed as *false positives* as they are irrelevant for the user of the detection tool. The reasons for this are manifold. For example, the detection tools often report clones in parts of code that are generated and never modified manually. Or, the reported "clones" are not actually clones as perceived by developers; the seeming similarity is caused by over-eager normalization applied by the detection tool. Most importantly, the clone detection results often do not support the goals of the user of the clone detection tool. For example, a user that aims to remove the found clones is only interested in the clones that are actually removable. Hence, he might, for instance, not be interested in clones that cross method-boundaries although these clones might be valuable for another user that uses the tool for a quality assessment of the same code.

However, the reasons why clone detection tools generate false positives, do not generally make a case against the tools. Rather, they stress that these tools need to be carefully *tailored* to match a specific project and user situation to be beneficial. Our experience with various industrial and open-source projects showed that, given a certain amount of tailoring, the rates of false positives can be reduced to 5% or less. Experience also shows that such low rates are imperative if clone detection should be applied in a continuous manner as developers are not willing (and not paid for) to consult clone lists containing false positives on a daily basis.

The mechanisms to reduce rates of false positives are almost as diverse as their causes: Filtering can be applied at various levels to exclude irrelevant, *e. g.*, generated, code from the analysis. Filters can be applied to whole files, based either on the names of the files or their content, or on parts of files, *e. g.*, by excluding specific methods that are generated by a GUI editor. Moreover, *Context-sensitive normalization* can be applied to tame an overeager normalization stage of a clone detector. For example, normalization of constant values can be turned off for sequences of constant definitions. Finally, if, for whatever reasons, certain false positives cannot be eliminated systematically, a technique

called *blacklisting* can be applied. Blacklisting allows to specifically mark certain clones and thereby exclude them from the results of future clone analyses.

Result Interpretation The interpretation of clone detection results is made challenging simply by the amount of data that a clone analysis of a large system generates. Users can be assisted by dedicated visualizations that convey clone information on different levels of granularity. Examples are *tree-maps* that are capable of visualizing the amount of cloning for an entire system in one screen and the *Seesoft* view that shows only dedicated subsets of the systems but provides more details. On the lowest level, a *compare view* can be used to inspect individual clones. These visualizations are accompanied by appropriate clone metrics that help to evaluate the extent of cloning.

4. CLONING IN OTHER ARTIFACTS

Cloning is not limited to source code but can be found in virtually all software engineering artifacts. For example, recent studies have shown that many (natural language) software requirement specification documents contain a significant amount of cloning [2]. Similarly, graphical models that are used in the design or as an input for code generators contain high levels of duplication [1]. Obviously, duplication in non-code artifacts leads to similar problems as code cloning. Even worse, cloning in pre-code artifacts can lead to clones or re-implementation in the source code that is based on the artifacts. The clone detection tool used in this tutorial can also be applied to detect duplication in textual documents and Matlab/Simulink models that are often used in the development of embedded systems.

5. SUMMARY

Cloning of source code is widespread and has severe consequences for maintenance effort and program correctness. Today's clone detection approaches and tools can be applied to detect cloning in practice if they are sufficiently tailored for the specific project situation. This tutorial transfers the insights that were gained in the clone detection community, and experiences that we collected during several years of applying clone detection in industry, to practitioners who want to apply clone detection in practice.

6. REFERENCES

- [1] F. Deissenboeck, B. Hummel, E. Juergens, B. Schätz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *ICSE'08*, 2008.
- [2] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In *ICSE'10*, 2010.
- [3] E. Juergens, F. Deissenboeck, and B. Hummel. Code similarities beyond copy & paste. In *CSMR'10*, 2010.
- [4] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE'09*, 2009.
- [5] R. Koschke. Survey of research on software clones. In *Duplication, Redundancy, and Similarity in Software*. Dagstuhl Seminar Proceedings, 2007.
- [6] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University at Kingston, 2007.