CPC – an Eclipse framework for automated clone life cycle tracking and update anomaly detection

Valentin Weckerle
Freie Universität Berlin

Abstract

Copy-Paste-Change (CPC) [1] is a framework for copy and paste clone[1] tracking and update anomaly[2] warnings within the Eclipse IDE.

CPC represents the first step towards an integrated and feature rich clone tracking environment which increases the general awareness of clones in a software system and provides notifications and warnings about potential clone related errors.

It is our hope that *CPC* will provide in-depth data about the day to day copy and paste habits of programmers in real environments which can help to improve our overall understanding of the 'Micro-process of Software Development', the small day to day activities of a developer.

CPC is written in *Java* 1.5 and is licensed under the *GPL*. It can be obtained from `http://cpc.anetwork.de` [2].

1 Introduction

The 'Micro-process of Software Development' represents one of the areas of interest of the software engineering research group of the Department of Computer Science at the Freie Universität Berlin [3]. Sometimes also called 'Actual Process', the research focuses on the small, every day actions of the programmer *(i.e. browsing code or documentation, modifying a method, copying text)*. Copy and paste actions are of particular interest as they can introduce source code duplications *(clones)* into a software system.

The presence of clones in applications has long been considered to be an indication of poor software quality [4, 5]. Yet past and current research findings strongly indicate that cloning in software applications is a pervasive phenomenon [6, 7]. Some studies report cloning rates of more than sixty percent in certain applications [8]. Aggressive removal of clones with support of automated clone detection and refactoring approaches has thus received a lot of attention.

In light of newer research findings this tough stance on cloning has been reconsidered and there are now many advocates of a more lenient approach to clone removal. It is argued that limitations of the programming language often make cloning impossible to avoid [9, 10, 11, 12], that cloning can be a viable design decision to attain specific design goals [12, 13, 14], that the majority of clone groups tend to be maintained consistently [11, 15] and that there is a possibility that code sections are similar by 'accident' [16].

However, even the supporters of a more lenient approach acknowledge that clones, or at least certain types of clones, in software systems are likely to have negative effects on the long term maintenance effort. Findings show that source sections which contain clones are likely to require more modifications during maintenance than clone free segments [17].

Many potential remedies have been suggested, ranging from new language features to special preprocessors or meta-languages [4, 18]. Aside of these endeavours to eliminate cloning completely, a different approach suggested by many studies, would be to add special clone tracking features to the software development environments [11, 12, 14, 19]. The idea is to retain the convenience and positive aspects of copy and paste cloning while at the same time trying to ease or prevent some of its major pitfalls. This perceived need for tool support is one of the key motivations for *CPC*.

In comparison with static clone detection techniques, copy and paste actions by individual programmers have received only little attention by the research community. The amount of available empirical findings is thus very limited. Kim et al. conducted a number of studies which highlighted the extend of copy and paste cloning by professional programmers [11, 12]. They observed an average of four non-trivial copy and paste actions per hour. This lack of data on copy and paste activities in real environments was another motivation for the development

[1] Clones are duplicated source code fragments within a software application.

[2] Update anomalies can occur if a modification to the content of a clone is not propagated consistently to all other copies of the source code fragment. A typical example are defect corrections which usually need to be applied to all copies of the defective code section. A developer can easily forget to update some of the copies.

of *CPC*. We're hoping to obtain a large, heterogeneous data set of copy and paste activities and clone evolution data for future clone research.

2 Related Work

In late 2007 Jablonski et al. started the work on *CnP*, a copy and paste clone tracking plug-in for the Eclipse IDE with very similar goals to those of *CPC* [20]. Currently only a prototype implementation of one aspect of *CnP*, consistent renaming of local identifiers in copied source segments, is available [21]. The high overlap between the goals of *CPC* and *CnP* makes it likely that knowledge exchange and software reuse opportunities will arise in the near future.

In mid 2007 Duala-Ekoko and Robillard presented a clone tracking tool called *CloneTracker* [22]. Similarly to *CPC*, it is an Eclipse plug-in which is aiming at supporting the developer during software maintenance by highlighting clones and issuing warnings when a member of a clone group is modified. *CloneTracker* employs a 3^{rd} party static clone detection utility to identify duplicated source code segments and does not take copy and paste activities of the developer into account. Duala-Ekoko et al. use a very interesting clone tracking approach. Instead of storing line or character offsets and updating them during source modifications, they try to extract a robust meta description of the clone segment from the surrounding source code, the so called *Clone Region Descriptor*.

Aside of these two a number of other clone tracking and editing tools less similar to *CPC* exist. However, none of these is making use of the copy and paste activities of the developer. Instead they rely on static clone detection approaches and thus face some of the typical problems like low precision and low recall [23, 24, 25, 26, 27, 28, 29, 30].

3 Objective

CPC tries to provide a versatile and highly flexible framework for clone tracking within the Eclipse IDE. It was designed as a base for future work in the area of clone tracking and facilitates the collection of data on typical copy and paste cloning activities of programmers. It furthermore improves the general awareness of cloning in an application by providing basic visualisations of clone data and establishes a basis for future notifications of the developer about potential update anomalies.

Aside of its framework aspects, *CPC* furthermore provides a ready to use Eclipse plug-in which can be used to reliably track copy and paste clones during the development of *Java* applications.

4 Framework

During the development of *CPC* the framework aspects were of key importance. As it represents the first step towards all encompassing copy and paste clone tracking within the Eclipse IDE, a highly modular and flexible framework architecture which enables 3^{rd} parties to easily extend or modify the existing functionality was crucial.

This was complicated considerably by the fact that most potential future uses of *CPC* and thus their API requirements are still largely unknown. 3^{rd} parties may furthermore need to reuse parts of *CPC* at different levels of granularity. One contributor might just want to provide a new clone visualisation, another might want to reuse only the clone tracking functionality of *CPC* and another party might want to reuse everything 'just' with another definition of a clone.

The *CPC* architecture currently consists of 28 highly independent plug-ins, 14 service providers, 101 interfaces, 355 classes and 66,573 lines of code. A detailed description can be found in the *CPC* thesis [1].

5 Heuristics

Heuristics for the classification of clones, the calculation of the similarity between two clones and the evaluation of clone modifications to identify update anomalies represent the heart of *CPC* and directly determine the accuracy of the clone update anomaly detection. The basic heuristics shipped with *CPC* can easily be extended, modified or removed by contributors in order to improve the overall performance. The *CPC* framework ensures that contributions from multiple parties can work together seamlessly.

CPC currently uses the size, complexity and the type of content of a clone for its classification. The content analysis is based on an abstract syntax tree representation of the clone's content. Each clone is assigned a number of classifications according to the language elements which are contained within it. I.e. a clone which contains an entire method would be classified as 'method'.

The similarity between two clones is based on the Levenshtein distance between their contents after a number of normalisation steps. During the preprocessing step whitespaces are normalised and *Java* language elements which are semantically equivalent, i.e. comments, are normalised or removed.

Each modification of a clone is evaluated by the clone modification heuristics in order to detect potential update anomalies. The evaluation is based on the clone classification, the nature of the change, the difference between the content of all members of a clone group as well as their age and location.

6 Challenges

A number of factors complicated the development of *CPC* considerably. Some were related to the complexity of the Eclipse platform others to defects and inconsistent or inadequate APIs. The inherent complexity made long term planning all but impossible as it often proved very hard to estimate the time needed for a specific part of the implementation. The complexity together with *CPC*'s rather uncommon requirements and its need for low level access to the internals of the Eclipse platform represented another serious concern. Considerable amounts of time were spent on exploration of the Eclipse source code due to lack of documentation in some areas.

Initial attempts to reuse existing clone tracking software at our working group were stifled by performance and dependency problems and some of the relevant functionality provided by the Eclipse platform could not be used due to inherent restrictions or other shortcomings. In other areas inconsistent behaviour of the Eclipse platform made extensive workarounds necessary and some intended functionality like undo/redo support could not be realised due to its negative performance impact.

However, the biggest problem proved to be the Eclipse team repository provider APIs. Some crucial aspects like registration of listeners for team operations were simply not supported at all and in other areas existing APIs were either not implemented or were not able to provide all the data required by *CPC*. All in all this made it very hard to achieve one of the original goals of *CPC*, the support of clone tracking in distributed development teams. While possible, the API shortcomings limit the reliability of these remote synchronisation aspects of *CPC* to a degree which prevented their inclusion into the shipped version.

7 Data

The evaluation of copy and paste cloning data collected during earlier experiments at the Freie Universität Berlin and the data collected during the testing phase of the *CPC* development confirmed findings made by Kim et al. and others [4, 12].

During the analysed experiments and observations, programmers created an average of 24.76 new clones per hour. Most clones were very small, however, on average, 7.25 of these were larger than 80 characters and 2.7 were even larger than 250 characters. An 8 hour working day would thus result in more than 50 large clones, a week in more than 250 and a month in more than 1000.

The complete modification history which *CPC* maintains for each clone instance provides further insights into the evolution of copy and paste clones. Only about 26 percent of all created clones were ever modified. However, this figure includes all the very small clone instances *(median clone size was 25 characters)*. When limiting the analysis to clones larger than 80 characters more than 64 percent were modified. The majority of modified clones were changed shortly after their creation and remained static from that point onwards. The median delay between the creation of a modified clone and its last modification lies at two minutes and 37 seconds.

While some clone groups had up to 72 members, most clone groups remained very small *(group size: average 2.39, median 2)* and groups with more than two members were mostly created within a short period of time. The median delay between the creation of the first and last member of such a clone group was just 19 seconds.

So far the wealth of data collected by *CPC* could only be analysed very superficially. The development of tools and approaches to handle the large amount of data is likely to prove fruitful. We also expect further interesting insights once we collect copy and paste data from a larger, heterogeneous developer base.

8 Summary

CPC represents the very first copy and paste clone tracking utility available for the Eclipse platform which is ready for general use. It supplies the Eclipse IDE with a central integration point for clone tracking activities and represents an ideal base for all kinds of tools which require clone or position tracking functionality. *CPC* provides such extensions with a degree of resilience against external file modifications which has so far not been available within the Eclipse platform. The very open and loosely coupled nature of the *CPC* framework enables 3^{rd} parties to reuse or exchange arbitrary parts of the implementation easily and ensures that contributions from different parties can coexist within the same *CPC* installation.

The evaluation of existing copy and paste data as well as new data collected during the testing phase of *CPC* yielded results on average cloning rates and the pervasiveness of copy and paste clones which confirmed earlier published findings. The detailed clone data collected by *CPC*, especially the clone modification histories and general clone evolution information, provides a new level of granularity much finer than available before. The employed copy and paste based approach furthermore provides a much higher precision than any static clone detection approach. Application of *CPC* might thus be able to provide new insights into copy and paste operations and the micro-process in general.

However, *CPC* represents only the first step on a

long road towards all encompassing clone tracking. It currently ships with basic clone visualisations and heuristics which need to be improved and extended in order for *CPC* to become truly useful. Synergies between *CPC* and *CnP* might prove fruitful in this regard. Future improvements in the Eclipse team provider APIs would furthermore open up new possibilities.

References

[1] Valentin Weckerle. CPC – an Eclipse framework for automated clone life cycle tracking and update anomaly detection. Master's thesis, Freie Universität Berlin, 2008.

[2] Valentin Weckerle. Official CPC website. `http://cpc.anetwork.de`.

[3] Sebastian Jekutsch. Micro-process of software development website. `https://www.inf.fu-berlin.de/w/SE/MicroprocessHome`.

[4] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, Queen's University at Kingston, 2007.

[5] Robert Tairas. Clone detection literature overview. `http://www.cis.uab.edu/tairasr/clones/literature/`.

[6] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.

[7] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

[8] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In David Lowe and Martin Gaedke, editors, *ICWE*, volume 3579 of *Lecture Notes in Computer Science*, pages 252–262. Springer, 2005.

[9] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the stl and some general implications. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 451–459. ACM, 2005.

[10] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. An empirical study on limits of clone unification using generics. In William C. Chu, Natalia Juristo Juzgado, and W. Eric Wong, editors, *SEKE*, pages 109–114, 2005.

[11] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[12] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE '04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[13] J.R. Cordy. Comprehending reality: Practical challenges to software maintenance automation. In *Int'l Workshop on Program Comprehension*, pages 196–206. IEEE Computer Society Press, 2003.

[14] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE*, pages 19–28. IEEE Computer Society, 2006.

[15] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In René L. Krikhaar, Chris Verhoef, and Giuseppe A. Di Lucca, editors, *CSMR*, pages 81–90. IEEE Computer Society, 2007.

[16] Raihan Al-Ekram, Cory Kapser, Richard C. Holt, and Michael W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE*, pages 376–385. IEEE, 2005.

[17] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *MSR*, page 18. IEEE Computer Society, 2007.

[18] Hongyu Zhang and Stanislaw Jarzabek. XVCL: a mechanism for handling variants in software product lines. *Science of Computer Programming*, 53(3):381–407, 2004.

[19] Zoltan Adam Mann. Three public enemies: Cut, copy, and paste. *Computer*, 39(7):31–35, 2006.

[20] Patricia Jablonski. Managing the copy-and-paste programming practice in modern ides. In Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr.,

editors, *OOPSLA Companion*, pages 933–934. ACM, 2007.

[21] Patricia Jablonski and Daqing Hou. Cren: A tool for tracking copy-and-paste code clones and renaming identifiers consistently in the ide. OOPSLA - Workshop: Eclipse Technology Exchange, 2007.

[22] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE*, pages 158–167. IEEE Computer Society, 2007.

[23] Simon Giesecke. Clone-based Reengineering für Java auf der Eclipse-Plattform. Master's thesis, Carl von Ossietzky Universität Oldenburg, 2003.

[24] Udo Borkowski. C4d website. `http://www.udo-borkowski.de/C4D/`, 2004.

[25] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Trans. Software Eng.*, 32(3):176–192, 2006.

[26] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VL/HCC*, pages 173–180. IEEE Computer Society, 2004.

[27] Thomas Dudziak and Jan Wloka. Tool-supported discovery and refactoring of structural weaknesses in code. Master's thesis, Technical University of Berlin, 2002.

[28] Blue Edge Bulgaria. Simscan for eclipse. `http://blue-edge.bg/simscan/simscan_help_r1.htm`.

[29] PMD plug-in for Eclipse. `http://pmd.sourceforge.net`.

[30] Iryoung Jeong and Seunghak Lee. Sdd: high performance code clone detection system for large scale source code. In Ralph Johnson and Richard P. Gabriel, editors, *OOPSLA Companion*, pages 140–141. ACM, 2005.