Clone Detector Use Questions: A List of Desirable Empirical Studies

Thomas Dean[1], Massamiliano Di Penta[2], Kostas Kontogiannis[3], and Andrew Walenstein[4]

[1] Queen's University, Department of Electrical & Computer Engineering, Kingston, ON, Canada, K7L 3N6
tom.dean@queensu.ca
[2] University of Sannio - Benevento, Department of Engineering, Palazzo ex Poste, Via Traiano, I-82100 Benevento, Italy
dipenta@unisannio.it
[3] University of Waterloo, Department of Electrical & Computer Engineering, Waterloo, ON, Canada, N2L 3G1
kostas@swen.uwaterloo.ca
[4] University of Louisiana at Lafayette, Center for Advanced Computer Studies, P.O. Box 44330, Lafayette, LA 70504-4330, U.S.A.
walenste@ieee.org

Abstract. Code "clones" are similar segments of code that are frequently introduced by "scavenging" existing code, that is, reusing code by copying it and adapting it for a new use. In order to scavenge the code, the developer must be aware of it already, or must find it. Little is known about how tools—particularly search tools—impact the clone construction process, nor how developers use them for this purpose. This paper lists five outstanding research questions in this area and proposes sketches of designs for five empirical studies that might be conducted to help shed light on those questions.

Keywords. code clone, clone detector, code search, reuse, scavenging, empirical study

1 Introduction

It is widely believed that code "scavenging" is a common practices, i.e., that searching for source code and then consequently copying it is an activity that many developers practice. In order to scavenge code, however, the developer must first be aware of it. One way is through search tools, such as the text search tools that are known to be well-used in development [1]. However these may not be ideal for scavenging code. A different scavenging-support idea was brought out in discussions at a recent Dagstuhl seminar [2]. The suggestion was to create a kind of "auto-complete" feature within the software development environment. In current auto-complete features the development envionment uses the current typing context to determine possible completions of the partially-finished phrase

Dagstuhl Seminar Proceedings 06301
Duplication, Redundancy, and Similarity in Software
http://drops.dagstuhl.de/opus/volltexte/2007/969

or item, possibly allowing the developer to select from a menu. The basic idea for scavenging auto-complete is to show the developer code fragments that might match the code at the current focus.

Whether or not a code-scavenging auto-complete tool would work well or not, a question is raised whether such scavenging-assisting tools could possibly influence the presence of clones into source code, and whether this should be subject of empirical studies. This question was discussed in a working session on "Empirical Studies of Clone Detectors" at the DRASIS seminar [2]. Based on these discussions, this paper presents: (1) an introductory overview of some code search techniques known already to exist that could be used for code scavenging, (2) a preliminary list of five research questions about these tools and their uses and impacts, and (3) five sketches of empirical studies that might be used to help answer these research questions.

2 Code Search Overview

During software development, developers may realize they need a piece of code, a function, or a class that implements a particular feature. Experience in the field suggests that, ordinarily, they would try to avoid reinventing the wheel and, above all, to reduce their effort. Thus it is plausible that in many circumstances they will search for pieces of code that provides what they need, or at least something that could be easily adapted to that purpose.

Several different mechanisms or tools might be employed to search for such code, including:

Component repositories: in many cases, it would be possible to find—either on the Web or inside a component repository—a component that encapsulates the needed feature. In other cases, a component realizing what needed might not be available. Alternatively, the available might not be the right solution for the particular problem; for example, the developer does not need a component but rather a piece of source code to be templated for a particular purpose, e.g., handling synchronizations, interrupts, adapting another component, visiting a data structure, etc.

Code search tools on the desktop: Information Retrieval (IR) techniques have been successfully applied to source code analysis, for example for traceability recovery [3,4] and for feature location [5,6]. The idea of IR-based feature location is to find source code fragments textually similar to a sentence describing the feature. Recently, these techniques have been applied to support source code browsing and, in general, to search into source code files. To this aim, Poshyvanyk *et al.* [7] developed an Eclipse plugin.

Web searching for source code: The simplest possible way is to use a search engine (e.g., Google) and try to search whether the piece of functionality to be realized is already somewhere on the Web. For example,

`http://www.google.com/search?hl=en&q=Handling+interrupts+in+C`

might be used to find interrupt handlers in C, or

http://www.google.com/search?hl=en&q=FFT+implementation+in+Java may be used to find Fast Fourier Transform implementations in Java. These queries would return, among other links, pointers to pages containing the pieces of code the developer need. Now Google also provides a specific search interface for code in their "Google Code Search" feature.

3 Empirically assessing the relationship between code search and cloning

Little is known about the impact of code search tools on their use in scavenging, or on their impact on clone construction. The following are questions that need answering:

RQ1: To what extent does the use of source code searching tools impact developer effort?

RQ2: To what extent does the use of source code searching tools impact on the presence of clones in the developed source code?

RQ3: To what extent does the use of source code searching tools impact the source code quality, e.g. on the presence of defects in the source code?

RQ4: If a developer makes his or her source code available on the Web, how frequently is the code found and used once it is indexed by search engines?

RQ5: How do developers actually go about searching for code? Are there differences in the behaviour based on developer contexts?

These research questions (RQ) are labeled for ease of reference below. To answer the aforementioned questions, a number of empirical studies would be necessary. The following subsections briefly outline the designs of some possible ones. These studies, clearly, are not the only possible ones. Rather, the point of listing them is to provide a way of envisioning some of the more promising ways of empirically exploring the research questions.

3.1 Study I - How developers use code search tools

The aim of this is to answer questions RQ1-3, and would, in general, help understand how the developers' behavior changes with the availability of source code searching tools. The envisioned study is a controlled experiment in which subjects will be asked to perform a series of development or maintenance/evolution tasks. The treatments would be the different search tools, i.e., the subjects would be divided into groups each having different search tools available (e.g., Internet access or also code search tools); the control group would have no such a facilities available.

The table below shows a possible, counter-balanced experiment design. To implement such an experimental design, subjects need to be split into four groups

(A, B, C, D) and they need to perform, over two labs, two different tasks (T1 and T2) with (Search) and without (No Search) the availability of searching tools.

	Group A	**Group B**	**Group C**	**Group D**
Lab 1	T1:Search	T1:NoSearch	T2:Search	T2:NoSearch
Lab 2	T2:NoSearch	T2:Search	T1:NoSearch	T1:Search

The independent variables include: Task, the Lab, the subjects' ability, and the availability of the code searching tool (main factor). Different dependent variables can be considered:

1. the effort;
2. the cloning percentage in the produced code; or
3. the code functionality and quality, e.g., assessed through some test suites.

To ensure the effects of scavenging are made plain it might be necessary to carefully select tasks in which an easy solution is available by cloning some part of the existing code.

3.2 Study II - How code posted on the Web is accessed or copied

This study is conceived of as a case study, and would aim to answer question RQ4. The protocol would require some fragments of source code realizing different functionalities to be posted on a Web server. Then, the server log is analyzed for access to the code. At the same time, queries related to the code posted are submitted to a search engine to analyze the hits related to the source code just posted. To avoid biasing the study, such queries need to be formulated from someone different from the code developers. For example, a pool of students might be used from a class.

3.3 Study III - What queries are made to search engines to find code to be cloned

This study, again a case study, aims to answer question RQ5. Search engine access logs could be mined to investigate how developers use the engines to search source code. Both search engine owners and researchers would benefit from such a study, since it aims to answer our research questions and, also, helps to tune the search engine to better search source code (which is for some search engines a specific feature—Google Code Search, for example).

The study can be performed by asking search engine owners to provide subsets of their logs that match some queries related to source code searching. Clearly, confidentiality constitutes a major issue for this kind of study. Nevertheless, the study can also be performed in a simpler way, i.e., within an organization using a proxy to access the Web. In such a case, it would suffice to properly configure the Web proxy and then use its log to perform the analysis.

4 Conclusions

At the present time it seems reasonable to suggest that the tools available can have an important impact on the scavenging practices of developers and therefore the properties of code clones within software systems. We presented five sketches for empirical studies that might be conducted to gain more knowledge about this topic. The hope of this paper is that these sketches can help organize the efforts of the research community, and to prompt and inspire work in this area.

References

1. Singer, J.A., Lethbridge, T.C.: (Studying work practices to assist tool design in software engineering) 173–179
2. Walenstein, A., Koschke, R., Merlo, E.: Duplication, redundancy, and similarity in software: Summary of Dagstuhl seminar 06301, Dagstuhl, Germany, Dagstuhl (2006) ISSN 1682–4405.
3. Antoniol, G., Canfora, G., Casazza, G., Lucia, A.D., Merlo, E.: Recovering traceability links between code and documentation. IEEE Transactions on Software Engineering **28** (2002) 970–983
4. Marcus, A., Maletic, J.I.: Recovering documentation-to-source-code traceability links using latent semantic indexing. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2003) 125–135
5. Antoniol, G., Gueheneuc, Y.G.: Feature identification: A novel approach and a case study. In: ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Washington, DC, USA, IEEE Computer Society (2005) 357–366
6. Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I.: An information retrieval approach to concept location in source code. In: WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04), Washington, DC, USA, IEEE Computer Society (2004) 214–223
7. Poshyvanyk, D., Marcus, A., Dong, Y.: JIRiSS - an eclipse plug-in for source code exploration. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), Washington, DC, USA, IEEE Computer Society (2006) 252–255