

# Complexity and Feasibility Issues in Object Oriented Clone Detection

Ettore Merlo<sup>(\*)</sup>, Giuliano Antoniol<sup>(\*\*)</sup>, Massimiliano Di Penta<sup>(\*\*)</sup>  
ettore.merlo@polymtl.ca, antoniol@ieee.org, dipenta@unisannio.it

<sup>(\*)</sup> École Polytechnique de Montréal - Montréal, Canada

<sup>(\*\*)</sup> RCOST - Research Centre on Software Technology  
University of Sannio, Department of Engineering  
Palazzo ex Poste, Via Traiano, I-82100 Benevento, Italy

## 1. Introduction

Large multi-platform software systems are likely to encompass a variety of programming languages, coding styles, idioms and hardware-dependent code. Analyzing multi-platform source code, however, is a challenging task. Assembler code is often mixed with high-level Object Oriented (OO) or procedural programming languages. Furthermore, scripting languages, configuration files, and hardware specific resources are typically used.

Often, systems were originally conceived as a single platform application, with a limited number of functionalities and of supported devices. Then, they evolved by adding new functionalities and were ported on new product families. In other words, new devices and target platforms were added. When writing a device driver or porting an existing application to a new processor, developers may decide to copy an entire working subsystem and then modify the code to cope with the new hardware. This approach is believed to increase the chances that the developers' work will have little unplanned effect on the original piece of code they have just copied. However, this evolving practice promotes the appearing of duplicated code snippets, also called *clones*.

There have been many publications proposing various ways of identifying similar code fragments and components in a software system [14, 9, 6, 2, 10, 12, 3]. However, those publications essentially focused on procedural programming languages such as C and only few contribution addressed the problem of detecting duplicated code in OO systems [8, 1]. Moreover, algorithms presented and published to detect duplications in procedural systems exhibit linear or almost linear complexities whereas published similarity computation in OO has higher complexities (see [13]). For example, the complexity of similarity computation between C++ classes published in [1] is  $O(n^2)$ , where  $n$  is the number of classes. To compare two subsequent releases, each one composed of about 200 classes, of the same sys-

tem, several hours were required. Complexity issues raise from the the nature of properties of objects, as it will be discussed in Section 2, which are characterized by methods, which are procedural in nature, and by state variables of different types. With the increased adoption of OO technologies, the lack of computationally efficient approaches to identify duplicated code on OO systems is limiting the ability to analyze large systems and the scalability of known techniques.

## 2. The Paradigm and the Problem

We believe that the complexity of detecting duplicate code in OO system is essentially related to the OO paradigm and its available representations.

Bunge's ontology [4, 5] has been a source of inspiration in the OO domain. According to this ontology objects can be viewed as *substantial individuals* which possess *properties*. Chidamber and Kemerer [7], proposed a representation of *substantial individuals*, objects, as a finite collection of properties:

$$X = \langle x, P(x) \rangle \quad (1)$$

where the object  $X$ , is identified by its unique identifier,  $x$ , and  $P(x)$  is its finite collection of properties. If similarity between two individuals is defined as the intersection of their sets of properties, two individuals are indistinguishable if and only if they share the same name and possess the same collection of properties.

In general, two objects  $X$  and  $Y$  may possess different properties. Comparing individuals for similarity translates into checking the similarity of the individuals' properties. The similarity between two individuals is related to their properties, in that it seems reasonable that the more properties they share, the more similar they are. If two individuals are identical then no matter how we measure their sim-

ilarity, similarity must evaluate to 1; conversely, the more differences two individual exhibit, the less similar they are [11].

The idea is easily generalized to represent all the artifacts which characterize a software system or which are produced along the software life-cycle. Since Bunge's ontology does not specify the nature of the represented object, it can be applied to the different software artifacts (i.e., requirements, design, user manual or source code), once an identity and a set of properties are defined. However, a criterion imposing *substantial individuals* identity is unnecessarily stringent and may lead to unsatisfactory results.

Nevertheless the ontology clearly state the double nature of entities: they possess properties and an identity. Thus, due to the intrinsic nature of entities any similarity definition must account for the identity (i.e., the class name) and properties (i.e., methods and attributes), see for example [1].

Approaches developed to detect duplicated code with linear complexity in procedural systems can be applied to the methods of an OO system. However, the extracted information accounts only for one aspect of the OO nature, the methods, and they somehow neglect the object identity and the attributes.

In other words we believe that in the OO area there is the need of a substantial research effort to define new similarity measures and new approaches aiming at reducing the computational cost, while, at the same time, fully considering the substantial individuality of objects together with their properties. It is well known that in several fields such as graph theory, artificial intelligence and pattern recognition heuristics and opportunistic strategies often allow to reduce the average case complexity when linear solution exists for specific sub-problems. Indeed, exact solutions may or may be not computationally feasible, whereas approximated solutions with low or almost linear complexity may ensure scalability to multi-million LOCs OO systems. Such systems, which have been developed with OO programming languages, could be analyzed at the cost of slightly reducing the accuracy of the computed solution.

## References

- [1] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Maintaining traceability links during object-oriented software evolution. *Software - Practice and Experience*, 31:331–355, 2001.
- [2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of IEEE Working Conference on Reverse Engineering*, July 1995.
- [3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 368–377, 1998.
- [4] M. Bunge. *Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World*. Reidel, Boston MA, 1977.
- [5] M. Bunge. *Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems*. Reidel, Boston MA, 1979.
- [6] E. Buss, R. D. Mori, W. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. Muller, J. M. S. Paul, A. Prakash, M. Stanley, S. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the cas program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.
- [7] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [8] F. Fioravanti, A. Migliarese, and P. Nesi. Duplication analysis of object oriented systems by using trend tool. In *Proceedings of the International Conference on Software Engineering*, pages 577–586, Toronto, May 12-19 2001. IEEE CS Press.
- [9] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*, pages 171–183, October 1993.
- [10] K. Kontogiannis, R. D. Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, March 1996.
- [11] D. Lin. An information-theoretic definition of similarity. In *the 15th ICML*, pages 296–304, Madison WI, 1998.
- [12] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 244–253, Monterey CA, Nov 1996.
- [13] F. Van Rysselberghe and S. Demeyer. Evaluating duplicated code detection techniques. In *Evolution of Large-scale Industrial Software Applications (ELISA)*, pages 1–12, 2003.
- [14] T. J. McCabe. Reverse engineering reusability redundancy: the connection. *American Programmer*, 3:8–13, October 1990.