Linear Complexity Object–Oriented Similarity for Clone Detection and Software Evolution Analyses

E. Merlo* and G. Antoniol** and M. Di Penta** and V. F. Rollo**
ettore.merlo@info.polymtl.ca antoniol@ieee.org dipenta@unisannio.it f.rollo@unisannio.it

*École Polytechnique de Montréal
Montréal, Canada

** RCOST - Research Centre on Software Technology
University of Sannio, Department of Engineering
Palazzo ex Poste, Via Traiano 82100 Benevento, Italy

Abstract

With the widespread adoption of object–oriented technologies, the lack of computationally efficient and scalable approaches is limiting the ability to model and analyze the history of large object–oriented software systems. This paper proposes an approximate representation of object–oriented code characteristics, inspired by pattern recognition centroids for clustering.

An interesting application of such a representation is a linear–time complexity algorithm to detect duplicate or nearly duplicated code in object–oriented systems. The algorithm accuracy and time complexity were assessed on 11 releases of a large software system, the Eclipse Framework.

Keywords: Object–Oriented Software Evolution, Clone Detection, Source Code Analysis

1. Introduction

Object–Oriented (OO) technologies are changing the software development landscape. Although introduced over 30 years ago, only recently their adoption in the development of software systems is clearly visible.

Despite the significant progress in the adoption of OO technologies, there is a lack of low complexity approaches to analyze large OO systems. The lack of scalable, computationally efficient approaches is limiting the ability to model the evolution of large OO systems, the possibility to efficiently support refactoring or help program understanding activities.

This paper proposes an algorithm for approximate similarity computation with linear time and memory complexity to study large OO software system. The key idea to avoid quadratic cost stems from the pattern recognition approaches in clustering which represent and reason about clusters of elements by replacing them with a proper representative one.

In analogy with astrophysics, as presented in [8], pattern recognition suggests the use of center-of-mass and centroids to represent clusters of elements which can be ranked by some distance measures. In an astrophysical interpretation of OO systems, objects and classes can considered as "bodies" in a $n - bodies$ gravitational problem like those which happen in galaxies; properties such as the number of attributes or the number of methods in a class directly influence the center of mass of a class. For any given class their centers of mass can be easily computed with linear complexity. Once the centers of mass are available, they can be used to find with linear complexity, classes whose centers of mass are close one to another. This proposed linear algorithm has several foreseeable applications such as program understanding, refactoring or clone detection. As an application example this paper focuses on OO clone detection within and between software releases.

The proposed approach overcomes the intrinsic limitation of applying clone detection methods, developed for procedural code [2, 3, 5, 6, 7], to OO code. Moreover, such approaches are not applicable to OO as they are. Only few contributions addressed the problem of detecting duplicated code in OO systems [1, 4], and unfortunately, published contributions do not propose efficient computation approaches. On the contrary, the newly proposed algorithm compares thousands of classes in few seconds.

After providing an original operational definition of OO similarity, this work applies the similarity definition to code duplications identification (i.e., clone detection) using a novel linear time algorithm stemming from Newton mechanics. The accuracy and the time complexity of the approximate algorithm are then compared with those of an exact algorithm, by analyzing 11 releases of a large the open source software system, the Eclipse Framework.

The remainder of the paper is organized as follows.

Section 2 proposes a definition of OO clones, as well as the galaxy approach for studying software evolution, from which stems the algorithm for linear OO clone detection. Section 3 describes the empirical study performed, detailing the context, the hypotheses, discussing the validity threats and, finally, presenting results. Section 4 concludes.

2. Object-Oriented Similarity and Clone Detection

Any code duplication detection algorithm relies on an operational definition of similarity; in other words, the core of the problem is to exhibit criteria or algorithms to effectively decide whether or not two fragments of code are to be considered indistinguishable (or nearly indistinguishable).

The operational definition of clones adopted in this paper stems from some considerations and definitions: code fragments are clones under a given similarity measure if and only if the associated similarity measure is one or, relaxing the criterion, above a threshold of interest.

To nail down operational definitions of similarity suitable for the OO paradigm the following definition was given:
Exact-clone: *two entities X and Y are exact-clones if they have exactly the same number of properties (e.g., attributes, methods, inheritances, aggregations and associations), and each method of X has a corresponding clone in Y.*

In turn, two methods are considered indistinguishable if they exhibit the same software metrics values.

In general, the above definition requires to compare objects property sets ending up in a complexity $\mathcal{O}(n^2)$ where n it the number of classes. Much alike, the approach presented in [1] the complexity of finding a similar property in a given set is linear in the set size, however, if the computation of all pair similarity is required the complexity is tied to the square of the set cardinality.

2.1 Galaxy clone detection approach

As a first approximation in astrophysics, when modeling the interactions between distant galaxies, the position of asteroids and planets within a galaxy can be disregarded by simply considering all the galaxy mass concentrated in the center-of-mass.

This Newton's mechanical metaphor has to be adapted to OO software; in an OO system, objects (classes) may be considered as galaxies or solar systems. Properties such as the number of attributes or the number of descendants in a class hierarchy directly determine the center of mass (as the mass of the sun does for the solar system), while methods (the planets or stars) contribute to it. For any given entity its center of mass can be easily computed with complexity $\mathcal{O}(n)$; in other words, the center of mass of the galaxies (i.e., classes) contained in the universe (the software system under analysis) is computed in linear time. More formally, the set of properties $P(x)$ has been divided into two sets:

- Elementary properties (the *sun*) such as the number of attributes, the number of associations, aggregations, etc; and
- Complex properties (the *planets*): the methods' metrics, i.e. the number of statements, the cyclomatic complexity, the number of parameters, etc.

Methods play the role of asteroids and planets they influence center of mass; their contribution to the center of mass is computed as software metric descriptive statistics (e.g., min, max, median). Thus, if methods are summarized with n features (metrics) and the object has also associated m elementary properties, the object center of mass is specified by $m + n$ real numbers. Clearly, summarizing the methods of any given object by taking software metric descriptive statistics may be too coarse. Going back to the astrophysics metaphor, extra features, such as the diameter of the solar system or its angular velocity, may be used to refine the model. This means, going back to the OO evolution domain, that a larger set of statistics may be needed. The essential fact is that each added statistics can be computed in linear time with the system size. Furthermore, it is necessary that the statistics is compatible with the scale of the metrics on which it is applied. For example, the metric needs to be in a *ratio* scale when computing the average.

Once a summary description of each entity in the system is available (i.e., the center of mass), code duplication identification is based on a similarity measure. In this paper, the similarity definition was inspired by the clone detection metric based approaches by Mayrand et al. [7] and Kontogiannis et al. [6].

Let $\mathbf{M}_X = < m_1(X), \ldots, m_n(X) >$ be the tuple of metrics characterizing the entity X; $m_i(X)$ $(i = 1 \ldots n)$ stands for the i-th software metric chosen to describe X, and n is a fixed number metrics describing the entities. Notice that a method is regarded here as an entity.

Any other entity Y described by the same set of software metrics will be considered indistinguishable from X *iff*:

$$m_i(X) = m_i(Y) \;\; i = 1 \ldots n \qquad (1)$$

The definition requires that classes have the same number of methods, aggregations, etc. plus it implies that pairs of methods (in the different classes) must obey the same rule.

Imposing the identity of metrics leads to a linear complexity algorithm for clone detection once the metrics describing the entity are interpreted as the key of a hash table. Hashing with chaining allows writing an efficient linear complexity algorithm to detect indistinguishable entities. However, to relax the imposed similarity criterion 1, some kind of approximation is needed. To retain linearity complexity, the approach adopted here was inspired by voice coding in telecommunication where voice samples are quantized and a controlled error introduced. In other words, the array of real numbers describing the given entity is first quantized and then used to construct the key of an hash table.

To have a more realistic idea of a "galaxy", Figure 1 depicts center of mass and method metrics for four classes, C_1, C_2, C_3 and C_4. To make two-dimensional representation possible, this example only uses two metrics, namely LOCs and cyclomatic complexity. Crosses, stars and circles represent metric values for methods, while the same bold symbols represent the centers of mass. Finally, the grid performs a quantization on the center of mass values (to perform a comparison and thus the clone detection). In our examples, classes C_1 and C_2, although having different centers of mass, will be considered as clones due to the quantization effect (their centers of mass are in the same cell of the grid).

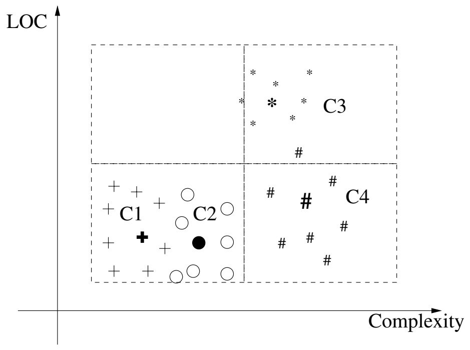

Figure 1. Class methods and their center of mass.

2.2 Exhaustive clone detection approach

In the *precise* approach, the comparison of each pair of entities in turn requires quadratic cost comparison of methods. Classes and methods comparison via brute force equality test on metric values has an additional linear cost.

A first improvement may be obtained via a *smart* comparison of classes and methods inspired from database compression techniques for retrieving identical elements. Instead of comparing metrics value by value, a *smart* but still quadratic algorithm compares two hash codes derived from the arrays of metrics. The overall complexity is still $\mathcal{O}(n^2)$ with the number of classes, but at least each class and method comparison has a constant cost.

2.3 Scalability

As stated above, the galaxy approach and the precise approach have complexity respectively of $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$ over the number n of compared classes.

Center of mass computation involved descriptive statistics extracted from methods, that can be carried out with linear complexity or $\mathcal{O}(n\ log(n))$ with the size of the input set. Therefore, center of mass computation is linear with the number of entities in the system, and approximately constant with the number of methods. Increasing the number of classes, does not usually impact the number of operations performed by any given class. Finally, hash table accesses have complexity $\mathcal{O}(1)$.

3. Empirical Study

In this section we compare accuracy of the galaxy approach versus a quadratic algorithm. Differences between the two approaches may be due to two possible phenomena: false positives and false negatives. These phenomena are usually tied to the granularity of the applied processing and, above all, to the similarity definition.

About false positives, it may happen that small classes with a similar number of attributes, and only access (*get* and *set*) methods exhibit the same metrics profile, thus quantization may lead to false positive. False negatives may occur when semantics is the same but the metric profile is different. Here the metric profile involves both the class level and the summary information (the center of mass). Clearly, the accuracy is tied to the center of mass approximation and the quantization function.

3.1 Study description

The context of the empirical study is to perform clone detection on Eclipse releases. Eclipse is an open source platform developed by the Object Technology International, Inc. Such a platform is designed to build integrated development environments (IDEs) for different languages (Java, C++, Web scripting languages and many others).

11 releases of Eclipse, ranging from 1.0 to 3.0M4, have been analyzed. The system size varies from 710 kLOCs, 5 kclasses and 46k methods of release 1.0, to 1771 kLOCs, 11 kclasses and 107 kmethods or release 3.0M4. The Eclipse evolution followed a sequential version numbering until release 2.1. Then, there has been a parallel development, consisting in *stable* releases (2.1.x) and *experimental* releases (3.0Mx). Further details are shown in Table 1.

As also briefly discussed in the introduction, the case study presented in this paper aims at analyzing perfor-

Release		KLOC	Classes	Methods	Date
Stable	1.0	719	5,013	46,608	Nov 01
	2.0	1,112	7,524	70,867	Jun 02
	2.0.1	1,115	7,533	71,062	Aug 02
	2.0.2	1,116	7,552	71,312	Nov 02
	2.1	1,596	10,007	95,244	Mar 03
Stable	2.1.1	1,598	10,014	95,312	Jun 03
	2.1.2	1,599	10,016	95,349	Nov 03
Exp.	3.0M1	1,654	10,484	99,204	Jun 03
	3.0M2	1,685	10,768	101,449	Jul 03
	3.0M3	1,715	10,994	103,247	Aug 03
	3.0M4	1,771	11,451	107,418	Oct 03

Table 1. Eclipse key features; LOCs, number of classes and methods

mance improvement of the proposed galaxy approach and the tradeoff between the timing improvement and the number of false positive obtained. At this purpose, the empirical study aims at confuting the following *null hypotheses*:

1. H_0 There is no significant improvement in terms of time performance when adopting the galaxy approach;
2. H_0 There are significantly more false positives/false negatives when adopting the galaxy approach than when applying the precise approach.

The empirical study needed, above all, to implement a toolkit composed of Java Parsing and Metrics Extraction Tools and Clone Detection Tools. Computations were carried out on a Pentium 4 Mobile 1.6 GHz laptop with 512 Mbytes of RAM running RedHat 9 Linux (kernel 2.4.20-28.9). Time reported and shown in tables were collected via the `time` command. The *user* CPU time required to run the command was reported in the results.

3.2 Empirical study results

Releases	Galaxy	Smart Quadratic
1.0 2.0	23	985
2.0 2.0.1	29	1548
2.0.1 2.0.2	28	1527
2.0.2 2.1	33	1997
2.1 2.1.1	39	2630
2.1.1 2.1.2	38	2589
2.1 3.0M1	41	2715
3.0M1 3.0M2	41	2889
3.0M2 3.0M3	42	3047
3.0M3 3.0M4	43	3227

Table 2. Clone detection time performance (sec) on a P4 running Linux and Perl 5.8.

3.2.1 Timing comparison

Source code parsing and metric extraction as well as center of mass computation were performed off-line. Parsing and extracting class and method level software metrics required about 45 minutes. Since the overall system size is of about 20 MLOCs, the throughput of this phase was about 10 kLOCs/sec.

Time required to compute the center of mass ranges almost linearly from 33 sec. (release 1.0) to 76 sec. (release 3.0M4). In other words, in agreement with the claimed $\mathcal{O}(n)$ complexity, doubling the number of classes doubles the time needed to extract the center of mass. Overall, about 680 sec. were necessary for the 11 release with an average of 30 KLOCs/sec.

Table 2 shows the time required by the galaxy approach compared to the *smart* quadratic algorithm. Doubling the number of classes and methods almost doubled the galaxy time while it almost multiply by four the time required by the precise approach. This result agrees very much with the complexity of the algorithms and the expected behavior. The overall time required on the 11 releases by galaxy was about 360 sec. with an average of 55 KLOCs/sec.

It is worth noticing that the *really* quadratic approach gave performance substantially worse then data reported in Table 2; for example a non *smart* quadratic algorithm, on the first release it required about 1320 sec. and, on the last release, 4300 sec.

Releases	Galaxy	Quadratic
1.0 2.0	2723	2718
2.0 2.0.1	9697	9693
2.0.1 2.0.2	9762	9758
2.0.2 2.1	6916	6912
2.1 2.1.1	14575	14575
2.1.1 2.1.2	14717	14717
2.1 3.0M1	13582	13582
3.0M1 3.0M2	13774	13774
3.0M2 3.0M3	14676	14676
3.0M3 3.0M4	15132	15132

Table 3. Clone pairs detected by different approaches

3.2.2 Accuracy comparison

Clones detected by the different algorithms are reported in Table 3. As expected the number of clones detected by galaxy was always greater or equal than the number identified by the precise approach. This is in agreement with the quantization effect masking differences between classes when summarizing methods with a center of mass representation. We experienced that classes classified as clones by the precise method were always classified as cloned by galaxy; as expected, the contrary was never observed. Indeed due to the independence of descriptive statistics from permutations $precise \subseteq galaxy$. Also, differences between galaxy and the precise algorithm was negligible and, in most cases, null.

Table 4 shows the performance of the proposed approach in terms of the performance improvement factor, which is the ratio between the quadratic algorithm and the galaxy one, and in terms of precision, which is the number of excessive clones given by the galaxy approach with respect to the number of clones reported by the quadratic method. The execution time has been reduced by a factor between 40 and 70 times, while the precision remains within a difference of less than 0.2 % and 60 % of the times the precision is 100.0 %.

Releases	Performance improvement factor	Precision (%)
1.0 2.0	42.82	99.81
2.0 2.0.1	53.37	99.95
2.0.1 2.0.2	54.53	99.95
2.0.2 2.1	60.51	99.94
2.1 2.1.1	67.43	100.0
2.1.1 2.1.2	68.13	100.0
2.1 3.0M1	66.21	100.0
3.0M1 3.0M2	70.46	100.0
3.0M2 3.0M3	72.54	100.0
3.0M3 3.0M4	75.04	100.0

Table 4. Performance of the approach

Caution has to be paid when reading figures of Table 1 and Table 3. Figures are clones pairs, similarity mapping is a one–to–many mapping. Given a class in one release, there may be many classes in the subsequent release indistinguishable from it. By inspecting metrics values of class pairs indicated as clones by galaxy and not by the precise approach we observed that galaxy false positives (5 for the releases 1.0 and 2.0) were classes extended by adding new methods or symmetrically changed in two (or more) methods. For example, the class `ElementChangedEvent` (release 1.0) was matched with the class `SchemaEditorContributor` (release 2.0) since in release 2.0 `ElementChangedEvent` added one method. The class `SchemaEditorContributor` (release 2.0) has exactly the same number of methods with exactly the some metric profiles. An example of the second phenomenon was the class `JdwpID`. In release 1.0 the method `hashCode()` and `getConstantMaps(Object)` has no parameter and one parameter respectively. In release 2.0 the situation was symmetrically changed in `hashCode(Object)` and `getConstantMaps()`. As a result, the center of mass was unchanged. However the number of LOCs of those methods is different thus the classes were classified as clone by galaxy and not by the precise algorithm.

4. Conclusions and work-in-progress

We have presented a novel linear complexity algorithm to compute class similarity in OO systems. The algorithm was inspired by the Newton's mechanics; it approximates class representations with a model similar to the physical center of mass of a $n - bodies$ gravitational problem. Time performance and precision of the approximated algorithm were compared to the precise implementation of our definition of OO similarity. Empirical evidence on about 20 MLOCs of Java code (11 releases of Eclipse) supports the outstanding superiority of this novel approach.

The algorithm was conceived to ensure high precision while keeping low the computational costs. Suitable applications range from interactive support to program understanding, to software evolution modeling, to refactoring and maintenance support in multi million LOCs systems. Tasks such as supporting program understanding, refactoring via IDE environments poses severe challenges to algorithms: accuracy must very high while response time have to be almost real time. We believe that our approach to OO similarity computation is a step forward providing a means to rapidly analyze large OO systems.

We are currently enriching out test set to obtain performance evaluation on different OO systems and languages.

References

[1] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Software - Practice and Experience*, 31:1–25, 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] F. Fioravanti, G. Migliarese, and P. Nesi. Reengineering analysis of object-oriented systems via duplicated analysis. In *Proceedings of the International Conference on Software Engineering*, page 577586, Toronto, ON, Canada, May 2001. IEEE Computer Society Press.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, July 2002.

[6] K. Kontogiannis, R. De Mori, R. Bernstein, M. Galler, and E. Merlo. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, March 1996.

[7] 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.

[8] A. J. Quigley. Experience with "fade" for the visualization and abstraction of software views. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 11–20. IEEE Computer Society Press, 2002.