An Exploratory Study of Interface Redundancy in Code Repositories

Adriano Carvalho de Paula*, Eduardo Guerra*, Cristina V. Lopes†, Hitesh Sajnani§, and
Otávio Augusto Lazzarini Lemos†‡
*Instituto Nacional de Pesquisas Espaciais, São José dos Campos, Brazil
{adriano.carvalho, eduardo.guerra}@inpe.br
†Donald Bren School of Information and Computer Sciences – University of California, Irvine – USA
lopes@ics.uci.edu
‡Science and Technology Department – Federal University of São Paulo at S. J. dos Campos – Brazil
otavio.lemos@unifesp.br
§Microsoft, Inc.
hsajnani@uci.edu

***Abstract*—An important property of software repositories is their level of cross-project redundancy. For instance, much has been done to assess how much *code cloning* happens across software *corpora*. In this paper we study a much less targeted type of replication: Interface Redundancy (IR). IR refers to the level of repetition of whole method interfaces – return type, method name, and parameters types – across a code corpus. Such type of redundancy is important because if two non-trivial methods ever share the same interface it is very likely that they implement analogous functions, even though their code, structure, or vocabulary might be diverse. A certain level of IR is a requirement for approaches that rely on the recurrence of interfaces to fulfill a given task (*e.g.*, interface-driven code search – IDCS). In this paper we report on an experiment to measure IR in a large-scale Java repository. Our target corpus contains more than 380,000 methods from 99 Java projects extracted randomly from an open source repository. Results are promising as they show that the chances of an interface from a non-trivial method to repeat itself across a large repository is around 25% (*i.e.*, approximately 1/4 of such interfaces are redundant). Also, more than 80% of the target projects contained IR (with the average percentage of redundant interfaces for these projects being above 30%). As additional analyses we investigated the distribution of the different types of redundant interfaces (*e.g.*, intra- vs inter-project); characterized the redundant interfaces and show that such a knowledge can help improve IDCS; and provided evidence that only a very small part of IR refers to method cloning (around 0.002%).**

I. Introduction

Recent studies have shown that large-scale software repositories contain significant amounts of redundancy[1] [1–3]. From the viewpoint of software reuse, such a property is important as it is an evidence of how much of the code contained in the repository is intentionally or unintentionally rewritten across projects. Also, if the level of redundancy is high (*e.g.*, per project) it is likely that a particular piece of code that a developer is about to write is already present in a code repository somewhere and could be reused.

One of the most popular types of replication investigated in code *corpora* is *code cloning* [4]. Cloning happens when similar or identical code fragments repeat across software projects [5]. Although such a property has received a lot of attention in the past, recently researchers have been looking at other types of replication, such as vocabulary or temporal redundancy. The idea is that sometimes two fragments of code can be similar with respect to other aspects besides text (think about different implementations of sorting algorithms, for instance, which can have the same function but different structure [6]). Vocabulary redundancy appears when different pieces of code share similar words [6], while temporal redundancy is concerned with the amount of code commits that are composed of previous commits [7].

Another type of replication that appears in software repositories and has received less attention is when whole method interfaces – return type, method name, and parameters types – repeat exactly or similarly across projects (called here *Interface Redundancy* – IR). This type of repetition is important because if two methods share similar interfaces it is very likely that they implement analogous functions (entirely or at least partially). This might be true even though their code, structure, or vocabulary is diverse. However, to the best of our knowledge to date there are no studies that focus on IR. A study that reveals properties of redundant interfaces can also support better tools that rely on such an information to fulfill their tasks (*e.g.*, interface-driven code search discussed below).

While IR and classical clone detection might be related, they are essentially different, as methods can possess redundant interfaces and at the same time significantly different implementations. For instance, consider the code fragments in Figures 1a and 1b which refer to methods with redundant interfaces from classes `FileTransferHandler` (project Free Instant Messeger Server[2]) and `MainFrameTransferHandler` (project Lilith[3]), both included in our target repository. While the methods contain redundant interfaces according to our definition, clone detection tools would probably not classify them as clones, because their texts are significantly different.

[1] Pre-print. This is the author's version of the work. Not for redistribution. Once copyright information is available, it will appear here.

[2] https://sourceforge.net/projects/fim1/ - 6/24/2016.

[3] https://sourceforge.net/projects/lilith/ - 6/24/2016.

```
public boolean canImport(JComponent comp,
        DataFlavor flavor[]) {
  for (int i = 0, n = flavor.length; i < n; i++) {
    for (int j = 0, m = flavors.length; j < m; j++) {
      if (flavor[i].equals(flavors[j])) {
        return true;
      }
    }
  }
  return false;
}
```

(a) Method `canImport` from project Free Instant Messenger.

```
public boolean canImport(JComponent comp,
        DataFlavor[] transferFlavors) {
  if(comp != desktop) {
    return false;
  }
  if(transferFlavors != null) {
    for(DataFlavor current : transferFlavors) {
      if(DataFlavor.javaFileListFlavor.equals(current)) {
        return true;
      }
    }
  }
  return false;
}
```

(b) Method `canImport` from project Lilith.

Fig. 1: Two methods with redundant interfaces and significantly different code.

However, it is clear that the functions are similar and it could be useful for the developer of one of the methods to see the other's implementation (for instance, some possibly important checks performed by the code in Figure 1b are absent in the one from Figure 1a).

A considerable level of IR in code repositories is also an important requirement for recent code reuse and repair approaches. For instance, in interface-driven code search (IDCS) [8–10], developers can search for code by using information from the interface of the desired method. IDCS will only be successful if method interfaces repeat across software projects, at least to a certain extent. Automated software repair based on semantic search [11], on the other hand, works under the assumption that if a project contains a buggy method, other software projects likely implemented a bug-free version of the same or similar method with analogous type signature.

In this paper we report on an experiment to measure IR in a large-scale Java repository. Our target corpus contains more than 380,000 methods extracted from 99 random SourceForge projects. We focus on the universe of *reusable methods*, that is, non-trivial functions that have a more defined interface – do not return `void` and contain at least one parameter –, have some implementation (that is, are not `abstract`) and are `public`. For these methods, when interfaces are similar, it is more probable that their functions are also analogous. This is opposed to *getter*, *setter*, and simple Java API methods, whose interfaces can repeat often but are less likely to be reused from one project to the other for being too simple. To overcome the vocabulary mismatch problem – or *synonymy* – we apply a simple vocabulary and type expansion to measure IR. Such an approach can help match similar interfaces that nevertheless have different but synonymous terms (*e.g.*, `String stripAccents(String)` and `String cleanAccents(String)`, actual interfaces found in a Java code repository). To show the impact of the application of such an expansion approach, we also present results for when it is not used.

In particular, we were interested in the overall redundancy of each target interface contained in the repository, but also on how much of the reusable method interfaces inside a given project tended to reappear in other projects. This second measurement can give an idea of how non-trivial methods in a given project could have been reused from other projects instead of being written from scratch, or copied manually. As additional analyses, we also wanted to characterize the redundant and non-redundant interfaces of reusable methods contained in the repository in terms of some attributes. Such results can help improve tools that are directed towards method interfaces (*e.g.*, IDCS) by supporting, for instance, prediction of query performance or automated query refinement approaches. To show evidence that IR is significantly different from classical method clone detection, we also measured how much of IR did not refer to clones.

Our results show that more than 25% of the reusable method interfaces repeat at least once across projects in the repository. This is an important result, as it shows that the level of IR in large code corpora tend to be significantly high. Moreover, more than 80% of the analyzed projects had at least one redundant interface and, on average, around 30% of the target methods of projects with IR possessed redundant interfaces. We also found that IR is much more prevalent inter-project than intra-project, and gathered interesting characteristics from the reusable method interfaces. For instance, we noticed that most methods names in these interfaces are formed by only two terms. A replication of a recent IDCS experiment [9] with a simple heuristic based on such an information can help significantly improve this approach. Finally, we show that only a very small part of IR happens due to cloning: approximately 0.002% of the inter-project-project redundant interface pairs found in our target repository did refer to method clones.

The remainder of this paper is structured as follows. Section II presents background information about the topics necessary to understand our study, such as IDCS and Sourcerer [12] (the infrastructure used to run our experiment). Section III lays down details about our study, such as the target research questions and used repository. Section IV presents our results and discusses them in the light of our research questions, and Section V examines the limitations of our study. Section VI discusses related work and, finally, Section VII concludes the paper with our main findings.

II. Background

Code redundancy has been a popular topic among software engineering researchers lately [1–4]. Such a property, if proven to be high, is an indication of how much rework has been going on among software developers. In fact, most studies point out that the level of redundancy tends to be significantly high in software *corpora* [1–4].

A different but nevertheless important type of redundancy that has received less attention from researchers is related to method interfaces. In this case we are interested in the extent to which the combination of return type, name, and parameters types of methods repeat across software repositories. As commented earlier, recent code search and repair approaches rely on such a replication to be able to perform well. Next we discuss Interface-Driven Code Search (IDCS), an approach to code search that utilizes interface descriptions to search and reuse source code. IDCS is important in our study because it was used to measure IR in our experiment. We also show that the results of our study can in turn help improve the effectiveness of IDCS (see Section IV).

A. *Interface-Driven Code Search*

To support sophisticated code search, recently researchers have proposed advanced infrastructure to analyze and index code. One of these efforts is Sourcerer [12]. Given a set of Java projects with available source code, Sourcerer can do the following: (1) automatically collect and store the code in a repository with a standardized directory structure; (2) find and perform an in-depth structural analysis of the available source code (*e.g.*, dependence analyses); and (3) store the code structure information in a centralized database. With this information it is then possible to perform several types of mining tasks within the target repository, including code search.

Sourcerer stores essential structure information of the target source code in a relational database. For example, by running queries in the database, we can tell which packages contain which classes, which classes contain which fields or methods, which methods are called by which methods, and what are the return and parameter types of a given method. Therefore, by using Sourcerer it is possible to perform searches in which information such as the return and parameter types of the desired function are specified in the queries. For instance, a user interested in the implementation of a function to *merge* a list of strings into a single string can search for a method that, given a *list of strings* (in Java: `List<String>`) returns the merged `String` and contains the term *merge* in its identifier [9]. We have been calling this type of search Interface-Driven Code Search (IDCS) [8, 9].

B. *Query Refinement*

The relative ineffectiveness of information retrieval systems is mainly caused by the inaccuracy with which a query formed by few keywords can model the actual user information need [13]. Another known issue is that, in most collections, the same concept may be referred to with different words. This issue, known as *synonymy* (or vocabulary mismatch problem), impacts the recall of most retrieval systems [14]. This is also true in the context of code search, because keywords are still required.

Query refinement (QR) methods have been proposed to deal with this problem. In general, QR methods are classified as *global* or *local*. Global methods expand query terms independently of the results returned from the original query, so that changes in its wording will cause the new query to match other terms with similar semantics. Local methods, on the other hand, adjust a query relative to the documents that originally appear to match the initial query [14].

Automatic query expansion (AQE) is a well known global method to overcome synonymy [13]. It augments the user's original query with new features with similar semantics. In order to perform the queries in our experiment to measure IR, we also included a query refinement approach to IDCS that was used before. The approach used in the experiment described in this paper [8] is a global approach that does not require running the query prior to expanding it: the query is first expanded with similar terms, and then executed to find relevant functions. This strategy generally improves performance, as queries are run only once. Another advantage is not requiring additional user input [14].

The AQE approach used in our experiment can be configured with different thesauri [8]. In this paper, we configured it to be used with WordNet [15], a lexical database for the English language, and a simple type thesaurus that expands numerical and Java collection types. In the above example where the user is looking for a function to merge a list of strings into a single string, the expanded IDCS query with the two selected thesauri would look like the following [9]:

return_type: *(String)*
keywords: *(merge^10 ∨ unite ∨ ...) ∧*
!(divide ∨ disunify)
parameter_types: *(List<String> ∨ List ∨ ...)*

Note that the query field where keywords are used is expanded with synonyms. More weight is given to the original terms selected by the user with the Lucene[4] boost factor, so that candidates that possess such terms are given a higher rank (in the example, *merge^10*). When there are multiple keywords, a conjunction is used among them, and a disjunction is used among the synonyms (*e.g.*, *merge* OR *unite*). In this way, a candidate is matched when it contains at least one term related to each of the concepts used in the query. The type thesaurus allows for a relaxed matching of the types. In the example, `List<String>` is expanded with other types of lists contained within the Java API (including the alternative where the type parameter `<String>` is removed). In queries, antonyms are also used to filter out unwanted results that might implement the opposite of the desired function (*e.g.*, *unzip* when *zip* is being searched)[5] [9].

Recently we conducted an experiment that points to the general effectiveness of IDCS, especially when compared to keyword-based code search and using AQE [9]. However, that study involved only 16 auxiliary functions that might not be representative of all types of functionality contained in a large repository. In this paper we go a step further and measure the general level of IR in a large corpus that includes several types of functions implemented in more than 300,000 methods.

III. Study Setup

Our goal is to investigate the level of redundancy of method interfaces inside source code repositories. In particular, we are interested in *reusable methods*, that is, non-trivial methods that have a more concrete interface (return a type other than `void`, have at least one parameter, are not *abstract*, and are `public`).

[4] http://lucene.apache.org - 06/25/2016.

[5] In this paper we do not use Lucene to run the queries, but a simplified version of IDCS ran inside the relational database. The main differences are that we do not use antonyms to filter out results, but only synonyms to broaden our result set, and the boost factor for original terms is not applied.

We did this because trivial methods such as *getters* and *setters* would hardly be interesting from a reuse perspective.

Our main question is whether or not the interfaces of such methods repeat significantly across projects. Such a measure is an indicator of how much reuse could – or did – take place by means of referring to the methods by their interfaces. If such a redundancy level is high, as discussed before, we have evidence that approaches like IDCS can be successful. If method interfaces do repeat often, say, per project, it is also a sign that when a developer is about to write a method, it is likely that the method is already present in the repository with similar interface. As secondary investigations we wanted to profile several features of interface redundancy in a large Java repository. Such a profiling can help improve tools that target method interfaces (*e.g.*, it can help predict or even automatically improve the successfulness of a given query). We also wanted to verify the differences between IR and classical code clone detection. Thus, in our study, we are interested in the following research questions:

In the context of large-scale Java source code repositories:

RQ1. What is the extent of interface redundancy in the repository, in general and per project?
RQ2. What is the distribution of the different types of interface redundancy in the repository (intra-project vs inter-project IR and project-project vs project-library IR)?
RQ3. How are redundant interfaces characterized in terms of different features (*e.g.*, most frequent number of parameters / number of terms used for method names)? Can this type of information help improve IDCS?
RQ4. How does interface redundancy relates to classical code clone detection? More specifically, how much of interface redundancy is not due to textual clones?

Since this is an exploratory study and we are not making direct comparisons between different approaches, in this paper we do not formally define hypotheses for our experiment. Rather, we decided to discuss our results descriptively, leaving more formal comparisons for future work.

A. Repository

Recently, Fraser and Arcuri [16] argued that several novel test data generation techniques are proposed without proper empirical studies to provide evidence for their usefulness. Case studies on such a topic tend to be either small or biased toward a specific kind of software [16]. To cope with this problem, they have randomly selected 100 Java projects from SourceForge[6] in what was called a first attempt to run a statistically sound experiment in test data generation. According to them, the resulting benchmark, called SF100, is statistically sound and representative of open source projects[7]. This issue also affects the evaluation of code search techniques: many times the target repositories are not a random sample of software projects, but a specific set of projects, which can introduce bias to the study. Concerned with this problem, and to construct a manageable and unbiased code base, we used Sourcerer to index Fraser and Arcuri's repository[8]. SF100 thus forms the target repository for the experiment described in this paper.

TABLE I: Filters applied to the `entities` table to reach the desired search space for our experiment. (Total = total of method interfaces after applying the filter)

Filter	Total
none	6,064,495
`entity_type = METHOD`	2,003,480
`modifiers like %PUBLIC%`	1,602,644
`modifiers not like %ABSTRACT%`	1,380,031
`params <> ()`	835,327
`return <> void`	401.749
`project <> 075 openhre`	385,078

As commented before, Sourcerer stores all code metadata in a relational database. The central element of the Sourcerer's model is the `entities` table, which contains information about all source code elements of a given code base (*e.g.*, classes, methods, fields). Once we had a Sourcerer database containing the SF100 metadata, we constructed the target repository to form the basis of our experiment. Table I presents the set of filters we applied to the `entities` table to reach the search space set of reusable method interfaces contained in SF100. The last line of the table is the result of applying all filters containing the desired interfaces for our experiment, that is, all non-abstract, public methods that contain at least one parameter and return a non-`void` type.

Project `075 openhre` was removed from our target repository because it contained a disproportional number of source interfaces (superior than the sum of all other interfaces from the remaining 99 projects), which could bias our results. To give an idea of size of the target repository, Figure 2 shows the remaining top 16 largest projects in terms of number of interfaces from our repository. Note that `openhre` is an outlier as it contains 16,671 source interfaces while all others have less than 1,800 (in fact, it contains more interfaces than the sum of the interfaces of all other projects). Therefore, the search space of our experiment was formed by the 385,078 interfaces of the reusable methods found in SF100 (minus the `openhre` project). The search space includes all reusable methods referred to in the projects of the repository, including the ones from libraries and the Java API[9].

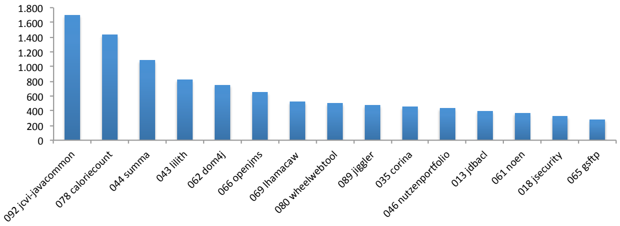

Fig. 2: Top 15 largest projects of our target repository in terms of number of source interfaces.

[6] http://sourceforge.net/ - 06/25/2016.

[7] http://www.evosuite.org/sf100/ - 06/25/2016.

[8] SF100 was used in other recent code repository experiments [8, 9].

[9] Although our repository could be considered small compared to large-scale code *corpora* recently used in some experiments (*e.g.*, [4]), it can represent a local forge from a company, for instance. We discuss such a limitation in our study in Section V.

From the set of 385,078, we then conducted a further filtering to reach the interfaces of reusable methods that were actually developed inside the projects contained in the repository (that is, excluding the ones from libraries and the Java API). With this further filtering we reached the number of 14,547 *source interfaces*, which were then used to form the base queries to search the whole search space in order to measure IR[10]. It is important to note that the set of 14,547 interfaces is used to form the queries, but the search space also includes the remaining interfaces of the reusable methods in the repository. We did this to also target the situation where a developer implements a method that could be reused from a library or the Java API, instead of being written from scratch. Some projects did not have any interfaces that matched our criteria (non-trivial and actually developed in the project). Finally, the number of projects that contained at least one source interface was 92.

B. Redundancy detection method

To measure interface redundancy in our target repository, we ran searches using IDCS for each of the 14,547 source interfaces. We ran intra-project and inter-project searches, to measure the level o IR both within projects and across projects. It is also important to note that, as commented in Section II, we applied the basic WordNet and type query expansion approach to measure IR in our study. We did this because in our previous studies we showed that such a type of expansion in this context generally improves recall without hurting precision. Since we are more interested in high recall in this study, that is, if an interface ever refers to a similar method we want it to be returned by our query, we believe it makes sense to always apply the expansion. In any case, to be able to see how the expansion impacts our results, we also show results when we ran the queries without applying it (see the variations of the similarity functions defined below). The differences are discussed in Section IV.

Another important aspect related to how we measured interface similarity is parameter ordering. We decided not to take into account the order in which parameters are defined for a method when judging whether or not a pair of interfaces is similar. This means that interfaces `int sum(int, float, int)` and `int sum(float, int, int)`, for instance, are regarded as similar in our experiment. We believe this is quite intuitive as the number and types of inputs received by a method are more important than the order in which they are received. However, it must be noted that only a few instances of interface pairs with different ordering of parameters were observed in our experiment (less than 1% of the total).

The metrics we used to measure IR are the following. First, for each source interface i_x, we run an IDCS query in the repository to check how many times i_x appears in other projects with similar interface. For each pair of candidate interfaces i_x and i_y, they are deemed similar ($S(i_x, i_y) = 1$) when the following is true: (1) return types are similar (either same or compatible according to our type thesaurus [9]), (2) method name is similar (either same or synonymous according to WordNet); and (3) number and types of parameters are the same or similar (regardless of the order in which they were defined). We defined a variation of the similarity function S to check the impact of turning on and off the type and WordNet expansion. S_{ne} refers to the similarity function when no expansion is used. For a given pair of interfaces (i_x, i_y), we say that i_x is the *source interface* and i_y is the *target interface*.

i_x is redundant if the number of other interfaces similar to it in the repository is larger than one according to the similarity function used. The redundancy of a given interface i_x ($R(i_x)$) is then defined as follows – for all other i_y target interfaces in the search space:

$$\mathbf{R(i_x)} = \begin{cases} 0 & \text{if } S(i_x, i_y) = 0 \text{ for any } i_y; \\ 1 & \text{if } S(i_x, i_y) = 1 \text{ for at least one } i_y. \end{cases}$$

We also define the number of redundant interfaces (#RI) for the whole repository according to a given similarity function:

#RI $= \sum_{x=1}^{n} R(i_x)$, where n is the number of source interfaces (14,547 in this study).

Moreover, in order to look at the distribution of redundancy of the target interfaces and the characteristics of the most redundant ones, we compute the number of interfaces similar to a given interface i_x in the target repository (Sim(i_x)). Such computation is defined as follows:

Sim$(i_x) = \sum_{y=1}^{n} R(i_x, j_y)$, where n is the number of interfaces in the search space (385,078 in this study).

To facilitate understanding how we compare interfaces, Table II shows actual examples of pairs of interfaces and how each similarity function would classify it (as similar or non-similar).

C. Classifying redundant interface pairs

To address research question 2, we classified pairs of redundant interfaces with respect to two aspects: (1) whether the source interface reappeared in the same project or in a different project (intra-project vs inter-project redundancy), and (2) whether the source interface reappeared in code developed inside a project, and not in libraries (project-project vs project-library redundancy). Such classifications are important as they show whether possible reuse happened across or within projects, and also whether the possible reuse is from code implemented by the developers of the projects themselves or from libraries or the Java API.

Here we formally define each type of redundant interface pair. For a given pair of interfaces i_x and i_y:

- (i_x, i_y) is called *intra-project redundant* if $S(i_x, i_y) = 1$ and i_x and i_y are located in the same project;
- (i_x, i_y) is called *inter-project redundant* if $S(i_x, i_y) = 1$ and i_x and i_y are located in different projects;
- (i_x, i_y) is called *project-project redundant* if $S(i_x, i_y) = 1$ and i_x and i_y are located in code inside projects and not in libraries; and
- (i_x, i_y) is called *project-library redundant* if $S(i_x, i_y) = 1$ and i_x is located in code inside a project and i_y is located in a library.

[10] It is clear that library methods are much more likely to repeat across projects, that is why we decided not to include them in the set of *source interfaces*. If we did so, we would probably introduce bias to the experiment. Since we are more interested in code that was actually developed inside projects by the developers, and not inside libraries, the natural choice was to include only those for the searches.

TABLE II: Examples of comparisons between interfaces and their similarity outcomes.

Interface 1 (i_1)	**Interface 2** (i_2)	$S(i_1, i_2)$	$S_{ne}(i_1, i_2)$
`String stripAccents(String)`	`String cleanAccents(String)`	1	0
`boolean matchesPrefix(char[],Pattern)`	`boolean matches(char[],Pattern)`	0	0
`Document read(File)`	`Document read(File)`	1	1
`int read(byte[], int, int)`	`int read(int, int, byte[])`	1	1

library-library redundant pairs could also be defined, but since we are more interested in pairs of interfaces for which at least one of the interfaces refers to a method implemented inside some project in the repository, we did not measure such a type of redundancy.

IV. Results and Analysis

To evaluate the extent of interface redundancy in a large open source Java repository, we applied the redundancy detection method described before to our target repository.

A. Level of interface redundancy in a large Java source code repository

To answer **RQ**1, we analyze results presented in Table III and Table IV, which shows our main redundancy results of our study, for all source interfaces and per project.

TABLE III: Redundant interfaces found in our target repository (#I - number of interfaces; %I - percentage of interfaces).

	#I	**%I**
Total	14,547	100.00
Redundant with expansion (#RI with S)	3,892	26.75
Redundant without expansion (#RI with S_{ne})	3,755	25.81
Difference	137	0.94

By analyzing Table III, we can see that a large portion of the 14,547 source interfaces considered in our experiment are redundant (3,892 – 26.75% – with expansion, and 3,755 – 25.81% – without expansion). Even when expansion is not used, more than 25% of the interfaces repeat in the repository.

Results per project are also telling (Table IV), as 86 out of the 92 projects (94.48%) that contain at least one source interface have at least one redundant interface. Even when we take the 99 initial projects considered in our experiment, these numbers are high: 86 out of 99 – more than 85%. Another interesting result is the average percentage of redundant interfaces from the projects with IR: 40.93%. This means that, on average, more than 30% of all the interfaces from non-trivial methods find at least one similar interface in the interfaces' search space.

With respect to research question 1, our results indicate that the level of IR in a large open source Java repository tends to be significantly high. This outcome is an evidence that approaches such as IDCS can be successful, as it shows that the chances of finding a related non-trivial method by using its interface are higher than 25% when no expansion is used. Moreover, our results also indicate that a large portion of the non-trivial methods contained in a given project have interfaces that reappear in other projects or libraries (more than 30%, on average).

B. Distribution of the different types of redundant interface pairs

In research question 2, we evaluate two types of characteristics of IR in our repository: intra-project vs inter-project IR and project-project vs project-library IR. For this research question we decided to follow a more conservative approach and present results only when not applying expansion. In this way, redundant pairs that are reported here refer only to interfaces that match exactly – same return type, same methods names, and same parameters types (regardless of the order in which they are defined).

With respect to intra-project vs inter-project IR, Table V shows results from our experiment regarding such an aspect. It can be observed that there are significant more inter-project redundant interface pairs than intra-project ones. This means that inter-project IR tends to be higher than intra-project IR. Although this is somehow expected – there is much more code outside a project than within it in a large repository –, the difference is quite telling: 300% more. This is an evidence that although projects might deal with diverse functionality there is still much commonality in terms of method interfaces. Such a result also points to the usefulness of reuse and repair approaches that rely on interface redundancy (such as IDCS), as for a given interface there might be a considerable number of candidates for reuse outside a given project in the repository.

We also measured IR with respect to project-project vs. project-library redundancy. Such a measure shows how much interfaces repeat from projects to other projects vs. from projects to libraries. It is important to note that Sourcerer considers each library a different project, so all project-library pairs are also inter-project. Table VI shows our main results for such an aspect. We can see that there is a very large number of project-library redundant interface pairs: more than 4M. This is an evidence that either many methods are being copied from libraries to projects for reuse, or developers might be missing chances to reuse methods from libraries instead of writing them from scratch. The third option is that the majority of redundant interface pairs reported here refer to methods that are not at all related in terms of functionality (*i.e.*, implement completely different functions). This is quite improbable as the interface pairs reported here are only exact matches (*i.e.*, no expansion is used).

Although the number of project-project redundant pairs is much lower than the project-library ones, there are still a lot of redundancy happening across projects without the involvement of libraries – almost 73,000 pairs in total and 1,000 pairs on average per project. This is an evidence that also points to the feasibility of method reuse across projects through approaches such as IDCS (as we show in results for research question 4, a significant part of these pairs do no refer to textual method clones).

TABLE IV: Redundancy results per project.

	With expansion	Without expansion	Difference
Total number of projects considered (containing at least one source interface)	92	92	–
Number of projects containing redundant interfaces	86	86	0
Percentage of projects containing redundant interfaces	93.48	93.48	0.00
Average # of redundant interfaces per project with redundancy	45	44	1
Average % of redundant interfaces per project with redundancy	33.76	32.53	1.23

TABLE V: Intra-project and inter-project redundant interface pairs (Diff.% = Different in percentage; Avg. = Average per project).

	Intra-project	Inter-project	Diff.%
Total	16,101	4,860,294	300.86
Avg.	309.63	56,515.05	181.52

TABLE VI: Project-project and project-library redundant interface pairs (Diff.% = Different in percentage; Avg. = Average per project).

	Project-project	Project-library	Diff.%
Total	72,741	4,787,553	64.82
Avg.	956.49	55,669.22	58.20

> With respect to research question 2, our results indicate that inter-project IR is significantly higher than intra-project IR (300% more common for our target repository). Our outcomes also point out that the largest portion of IR is related to interfaces that repeat from projects to libraries. On the other hand, there is also a significant number of redundant interface pairs across projects that do not refer to libraries (almost 73,000 pairs in total and 1,000 pairs on average per project).

C. Knowledge about redundant interfaces and its impact in the improvement of software reuse

We collected information about the characteristics of redundant interfaces found in our target repository. In particular, we measured the following: (1) the most common number of parameters used in the interfaces (*#param*); and (2) the most common number of terms used in the method name (*#mterms*)[11]. Table VII shows results for metric #param while Table VIII shows results for metric #mterms.

TABLE VII: Most common number of parameters (#param) used in redundant interfaces.

#param	Total	%
1	3,461	71.08
2	1,041	21.38
3	188	3.86
4	69	1.42
6	59	1.21
5	46	0.94
7	5	0.10
Total	4,869	100,00

By looking at Table VII we can see that the vast majority of redundant interfaces (92.46%) contain only one (71.08%) or two parameters (21.38%). Redundant interfaces with more parameters make up for only the remaining 7.54%. On the other hand, note that results in Table VIII shows that 90.02% of redundant interfaces have one, two, or three terms in their method names, with 34.55% of them containing two terms. Approaches that are targeted at interfaces could take advantage of such information to help improve query successfulness.

TABLE VIII: Most common number of terms used in method names (#mterms) in redundant interfaces.

#mterms	Total	%
2	1,682	34.55
1	1,401	28.77
3	1,300	26.70
4	328	6.74
5	151	3.10
6	4	0.08
7	3	0.06
Total	4,869	100,00

To check whether such type of knowledge could indeed have an impact on code search application, we devised a simple heuristic to improve IDCS. The idea is that since most redundant interfaces possess method names with only two terms, if a user ever submits a query with three or more terms, less significant terms exceeding two might be decreasing the likelihood of returning relevant code. In order to evaluate this simple query refinement approach, we replicated a recent IDCS experiment [8, 9] implementing such an heuristic. The original experiment involved 36 subjects (24 senior Computer Science undergraduate students and 12 professional developers) who guessed interface queries for 16 target functions contained in the SF100 repository. In our modified version of IDCS, if a query ever contains more than two terms, we remove the exceeding shortest terms used by the developer. In fact, when we ran such a modified version of IDCS, there is an increase in 8% in the total number of queries that return at least one relevant function: they go from 59 to 64, so there are 5 additional queries that become successful with the heuristic (a 5.6% increase in average recall). Interestingly, no queries that were successful with the original version of IDCS become unsuccessful by the modified version. This outcome shows the potential of using the results reached in our experiment to improve approaches targeted at method interfaces.

We have also gathered other types of information from redundant and non-redundant interfaces contained in our target repository. As an example of such type of information, Table IX shows the most common return types found in the entire repository, including in library interfaces. For instance, we can see that primitive types are much more frequent than the Java API wrapper types. Also, the most common Java API type used as return type in our repository was `String`. This outcome also points to Java developers' common use of types in interfaces. For instance, it can be seen that there are much more generic Collection interfaces used as return types (*e.g.*,

[11] To separate terms in the method name, we use the camel case naming convention frequently adopted by Java developers.

TABLE IX: Frequency of different return types used in interfaces in our target repository.

Category		Type	Subtotal	% (Subtotal)	Total	% (Total)
Primitive		`boolean`	68,334	17.75		
		`byte`	1,067	0.28		
		`char`	768	0.20		
		`double`	3,615	0.94		
		`float`	1,859	0.48		
		`int`	32,620	8.47		
		`long`	5,084	1.32		
		`short`	1,587	0.41		
		Total			114,934	29.85
Java API	String	`String`	45.256	11.75	45,256	11.75
	Wrappers	`Boolean`	683	0.18		
		`Character`	55	0.01		
		`Double`	206	0.05		
		`Float`	131	0.03		
		`Integer`	516	0.13		
		`Long`	220	0.06		
		`Short`	98	0.03		
		Total			1,909	0.50
	Collection	ArrayList	215	0.06		
		`Collection`	731	0.19		
		`HashMap`	90	0.02		
		`HashSet`	13	0.00		
		`HashTable`	96	0.02		
		`LinkedHashMap`	4	0.00		
		`LinkedList`	9	0.00		
		`List`	10.047	2.61		
		`Map`	1.067	0.28		
		`Set`	557	0.14		
		`SortedMap`	294	0.08		
		`SortedSet`	216	0.06		
		`TreeMap`	2	0.00		
		`Vector`	621	0.16		
		Total			13,962	3.63
Other popular		byte[]	3,949	1.03		
		`Object`	31.113	8.08		
		`String[]`	3.184	0.83		
		Other	60.101	15.61		
		Total			98,347	25.54
User defined		`Value`	1,478	0.38		
		`Expression`	764	0.20		
		`XObject`	721	0.19		
		Other	107,707	27.97		
		Total			110,670	28.74
		Grand Total			385,078	100.00

`List`, `Set`) than concrete implementations of such interfaces (*e.g.*, `ArrayList` or `HashSet`). Such type of information, besides being informative in itself, could also be used in the future to improve approaches that target method interfaces.

With respect to research question 3, we found several interesting properties of redundant and non-redundant interfaces in the repository. For instance, our results show that redundant interfaces tend to have few parameters – one or two – and few terms in the method name (two was the most common). By using such type of information we showed that it is possible to improve a code search approach targeted at method interfaces – IDCS. In particular, a simple query refinement approach based on the method names terms information commented before significantly improved results from a previous IDCS experiment (8% improvement on the number of relevant functions returned and 5.6% improvement on average recall).

D. Interface redundancy compared to classical clone detection

In research question 4, we want to investigate the relation between IR and classical clone detection (*i.e.*, method clone detection based on textual analysis). The main question is how much of IR is due to textual clones, and how much could be related to Type-4 clones (*i.e.*, methods that implement analogous functions but with diverse code). In this endeavor we focused on project-project redundant pairs, as these are the interfaces that refer to actual methods implemented by developers inside projects, and not in libraries[12].

SourcererCC is an accurate token based near-miss code clone detection tool. Given a similarity threshold, it identifies if two code blocks are clones by looking at the contents of the blocks (*e.g.*, method body in case of methods). We set the SourcererCC similarity threshold to 80% as it gave maximum recall and precision in our earlier experiments in comparison with other tools. More details about SourcererCC and its evaluation can be found elsewhere [4].

Table X shows the clone detection results for all inter-project-project pairs. Note that we found that only 143 of the redundant pairs referred to method clones (around 0.002%), while the majority of pairs (72,722 – approximately 99.998%) did not refer to methods that had similar code. This is a very

[12] Although it would also be possible and important to evaluate project-library pairs, in this first exploratory IR study we decided to target only project-project pairs. One of the reasons is that there are libraries for which the source code is not available. In those cases it would not be possible to check for method clones.

important result, as it shows that IR differs significantly from classical method clone detection.

TABLE X: Classical method clone detection vs. IR for project-project redundant pairs (Redundant pairs refer to inter-project-project redundant pairs found in our target repository).

	Total	%
Redundant pairs	72,741	100.00
Clones	143	0.0019
Non-clones	72,598	99.9991

With respect to research question 4, our results show that only a very small percentage of IR is due to cloning – 143 clones were found (around 0.002% of the inter-project-project redundant pairs) –, while the vast majority do not refer to methods with similar code (approximately 99.998%). This is an evidence that IR significantly diverges from classical method clone detection.

V. Study Limitations

The main limitation of our experiment is the size of our target repository, which is formed by 99 Java projects extracted from SourceForge. This might affect the generalization of our results to larger corpora. In any case we believe our results are important as a first exploratory study into IR, as it does involve a significant number of methods in its search space (more than 380,000; *i.e.*, it can be representative of a company's local repository, for instance). On the other hand, it is unclear whether IR will change significantly when the size of the repository increases. This is because it is not obvious whether IR is relative to the number of methods contained in the repository; that is, even though there might be more modules to be redundant with respect to a given method, the number of queries will also increase with a repository with larger number of modules. Nevertheless, we intend to replicate our experiment with bigger repositories curated by our research group in the near future, to verify how IR can be affected by the repository size.

Another limitation of our study is that although we hypothesize that methods with similar interfaces should probably implement analogous functions, we do not show evidence of whether or not such a hypothesis hold. In any case, since we are using whole interfaces – return type, method name, and parameters – from non-trivial modules to deem a pair of interfaces similar, we believe it is likely that methods from matching interfaces do implement similar functions, if not completely at least partially (an initial look at some of the redundant interface pairs found in our study do point for this assertion). Nevertheless, we are currently devising an approach to verify such a hypothesis, at least to a certain extent. The idea is to gather a set of redundant interface pairs and run the referred methods by sampling their input space and verifying whether outputs are coincident. If they ever are – at least for part of the tested inputs – we have evidence that their semantics are also similar. Although such an experiment requires sophisticated techniques to enable its execution (*e.g.*, running an arbitrary method from a given project would necessitate slicing such a piece of code with its dependencies), we believe it is possible to at least have initial evidence about the extent of semantic equivalence of methods with redundant interfaces. A similar experiment was conducted before but with arbitrary pieces of code, not necessarily methods [17].

VI. Related Work

The idea of looking at Interface Redundancy is closely related with many areas of software engineering research that have similar concepts.

Clone detection. Several techniques have been proposed for clone detection over many years. A comprehensive survey highlighting the strength and limitation of various clone detection techniques is available at [18]. These techniques differ in many dimensions ranging from the type of detection algorithm they use to the source code representation they operate on. Techniques using various representations include Tokens [19, 20], Abstract Syntax Trees (AST) [21, 22], Program Dependence Graphs [23, 24], Suffix Trees [19, 25], Text representations [26, 27], and Hash representations [28]. Each of these different approaches have their own merits and are useful for different use cases. For example, AST based techniques have been shown to have high precision, and are useful for refactoring of clones, but they may not scale. Moreover, token based techniques have high recall but may yield clones which are not syntactically complete [29]. They are useful where high recall is important. Interface Redundancy is one of the light-weight techniques that can easily scale to thousands of projects in cases where recall is important. As a result, the findings presented in this study can be useful to the design of future clone detection tools.

Code reuse. Gabel and Su conducted a large-scale study on the uniqueness of source code in a repository of 6,000 projects [2]. They found high syntactical similarity at various levels of granularity. Similarly Mockus conducted a large-scale study of file-level reuse and found evidence of high code reuse. Jiang and Su identify functionally identical code fragments based on test executions [17]. While their goal of exploring functional similarity was similar to ours, our approach are widely different as we do not rely on test executions but only the similarity of the interface parts (return type, method name, and parameter types).

Ossher et al. [30] looked at reuse in open source Java systems using hash based file level clone detection and classified the usage scenarios into good, bad, and ugly. These scenarios included good use cases like extension of Java classes and popular third-party libraries, both large and small. All the above studies noted that exploring code reuse at a large scale as a limiting challenge. Our experience with this study suggests that Interface Redundancy, has a potential to be effectively used as a practical and inexpensive technique to identify potential reuse at a large scale.

Code search & program repair. Recently researchers have started using code search for program repair [11, 31]. The idea is to identify a buggy chunk and then find relevant, probably correct code from large code repositories as the source for the patches. Since the success of such approach relies on how big the repository is, the search space of potential patches is thus greatly increased. Interface Redundancy can be used as a quick and cheap approach to narrow down to the potential candidates.

VII. Conclusions

We have conducted an exploratory study to evaluate interface redundancy (IR) in a large Java code repository. Our results point to essential findings in this topic. The main conclusions are the following: (1) the level of IR is generally high for the whole repository – above 25% of the interfaces from non-trivial methods are redundant in the repository –, and on average per project; (2) inter-project IR is much more prevalent than intra-project IR (approximately 300% more common), and redundant interfaces that involve libraries are more frequent than the ones that refer only to code from the projects themselves (although there are also several project-project pairs in the repository – around 73,000 and, on average, 1,000 per project); (3) knowledge from redundant interfaces in the repository can help improve code search application (in particular interface-driven code search); and (4) IR significantly diverges from cloning as only about 0.002% of the project-project redundant interface pairs found in our study referred to method clones.

Future work includes replicating our study with larger repositories and also devising other approaches that take advantage of the IR knowledge of the target code corpus.

Acknowledgements

Otavio Lemos would like to thank FAPESP for financial support (grant 2015/12787-0). The authors would also like to thank Pedro Martins and Vaibhav Saini for their support with the SourcererCC tool. This work was partially supported by a grant from the DARPA MUSE program.

References

[1] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016.

[2] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proc. of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 147–156.

[3] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 306–317.

[4] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1157–1168.

[5] J. R. Pate, R. Tairas, and N. A. Kraft, "Clone evolution: A systematic review," *J. Softw. Evol. Process*, vol. 25, no. 3, pp. 261–283, Mar. 2013.

[6] Y. Higo and S. Kusumoto, "How should we measure functional sameness from program source code? an exploratory study on java methods," in *Proc. of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 294–305.

[7] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proc. of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 492–495.

[8] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, "Thesaurus-based automatic query expansion for interface-driven code search," in *Proc. of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 212–221.

[9] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?" in *Proc. of the IEEE 15th International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM 2015. IEEE, 2015, pp. 41–50.

[10] O. Hummel, W. Janjic, and C. Atkinson, "Evaluating the efficiency of retrieval methods for component repositories," in *Proc. of SEKE 2007*. KSI, 2007, pp. 404–409.

[11] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun, "Repairing programs with semantic code search (t)," in *Proc. of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 295–306.

[12] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, Jan. 2014.

[13] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Comput. Surv.*, vol. 44, no. 1, pp. 1:1–1:50, Jan. 2012.

[14] C. D. Manning, P. Raghavan, and H. Schtze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[15] G. A. Miller, "Wordnet: a lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, Nov. 1995.

[16] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Proc. of the ICSE 2012*. IEEE Press, 2012, pp. 178–188.

[17] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *Proc. of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 81–92.

[18] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program*, pp. 470–495, 2009.

[19] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[20] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. of Working Conference on Reverse Engineering*, 1995, pp. 86–95.

[21] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Working Conference on Reverse Engineering (WCRE'06)*. IEEE Computer Society, 2006, pp. 253–262.

[22] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: Scalable and accurate tree-based detection of code clones," in *Proc. of ICSE*, 2007.

[23] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," in *Proc. of the 8th International Symposium on Static Analysis*. Springer-Verlag, 2001, pp. 40–56.

[24] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *Proc. of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. IEEE Computer Society, 2001, p. 301.

[25] R. Koschke, "Large-scale inter-system clone detection using suffix trees," in *Proc. of CSMR*, 2012, pp. 309–318.

[26] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proc. of ASE 2001*, p. 107.

[27] J. Cordy and C. Roy, "The nicad clone detector," in *Proc. of ICPC*, 2011, pp. 219–220.

[28] S. Uddin, C. Roy, K. Schneider, and A. Hindle, "On the effectiveness of simhash for detecting near-miss clones in large scale software systems," in *Proc. of Working Conference on Reverse Engineering*, 2011, pp. 13–22.

[29] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *IEEE Trans. Softw. Eng.*, vol. 33, no. 9, pp. 577–591, 2007.

[30] J. Ossher, H. Sajnani, and C. Lopes, "File cloning in open source java projects: The good, the bad, and the ugly," in *Proc. of ICSM 2011*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 283–292.

[31] F. Long and M. Rinard, "Automatic patch generation by learning correct code," *SIGPLAN Not.*, vol. 51, no. 1, pp. 298–312, Jan. 2016.