2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

A Large-Scale Study On Repetitiveness, Containment, and Composability of Routines in Open-Source Projects

Anh Tuan Nguyen
ECpE Department
Iowa State University, USA
anhnt@iastate.edu

Hoan Anh Nguyen
ECpE Department
Iowa State University
hoan@iastate.edu

Tien N. Nguyen
ECpE Department
Iowa State University
tien@iastate.edu

ABSTRACT

Source code in software systems has been shown to have a good degree of repetitiveness at the lexical, syntactical, and API usage levels. This paper presents a large-scale study on the repetitiveness, containment, and composability of source code at the semantic level. We collected a large dataset consisting of 9,224 Java projects with 2.79M class files, 17.54M methods with 187M SLOCs. For each method in a project, we build the program dependency graph (PDG) to represent a routine, and compare PDGs with one another as well as the subgraphs within them. We found that within a project, 12.1% of the routines are *repeated*, and most of them repeat from 2–7 times. As entirety, the routines are quite project-specific with only 3.3% of them exactly repeating in 1–4 other projects with at most 8 times. We also found that 26.1% and 7.27% of the routines are *contained* in other routine(s), i.e., implemented as part of other routine(s) elsewhere within a project and in other projects, respectively. Except for trivial routines, their repetitiveness and containment is independent of their complexity. Defining a subroutine via a per-variable slicing subgraph in a PDG, we found that 14.3% of all routines have all of their subroutines repeated. A high percentage of subroutines in a routine can be found/reused elsewhere. We collected 8,764,971 unique subroutines (with 323,564 unique JDK subroutines) as basic units for code searching/synthesis. We also provide practical implications of our findings to automated tools.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, Enhancement**]: Extensibility

Keywords

Repetitiveness, Containment, Composability, Code Reuse

1. INTRODUCTION

Source code in software projects has a considerable level of repetitiveness [21, 23]. There are several reasons for that. Developers write their code in programming languages, which have strict, well-defined syntax and semantics. Syntactic rules in a programming language defined by its grammar enforce the presence of certain syntactic structures in the programs. This creates the repeated sequences of code tokens that include required keywords and separators [21]. Several studies have confirmed such repetitiveness. A study by Hindle *et al.* [23] has showed that source code has high repetitiveness. Gabel and Su [21] reported *syntactic redundancy* at the granularity level of 6–40 tokens (they treat source code as a sequence of syntactic tokens with the abstraction on variables' names).

Naming conventions and style conventions utilized by teams in software projects are also a reason for repeated code portions within a project or across multiple ones [6]. Pragmatic software reuse is another reason. The practice of copying-and-pasting source code leads to cloned code [12, 57]. Software reuse could occur at a higher level of abstraction, such as the reusable software libraries with APIs, third-party components, and frameworks, or the reuse of design patterns or structures. Moreover, software engineering (SE) tasks have commonality, which can be implemented with similar algorithms. That may lead to similar source code as well. Code portions involving the usages of related APIs have been shown to be repetitive [52].

Those aforementioned traits are good evidence of similarity in source code. While there exist prior studies on the similarity at the *lexical* and *syntactical* structure levels in source code [6, 21, 23, 56], none has shed insights on the **repetitiveness, containment, and composability** of source code at a higher level of abstraction such as **the semantic level** of routines. A routine is a portion of code (within a program) that performs a specific task, and independent and called by the remaining code [55]. In programming languages, routines are manifested as *procedures, functions, methods*, etc.

A (new) requirement might drive developers to implement a new task as well. However, is it possible that a routine that realizes that task already occurred elsewhere in the same project or a different one? Is that routine part of a larger routine in the same or a different project? If not, what portions of the new routine can be reused from other places? Do some (sub)routines often go together? Can they be reused together? Are there any parts of a routine with a certain size or complexity repeated/reused more than others? Those are fundamental questions in SE towards a more general picture on a point of convergence: whether all the building blocks (routines) of projects for all tasks will have been written.

This paper presents a large-scale study towards answering those questions. The answers for them will not only advance the state of the knowledge on SE, but also have practical implications on SE applications. First, the automated program repairing approaches [22, 54] involve searching a large space of programs in the codebase with the assumption that a fix might already occur in the same program or other ones [22]. FixWizard [51] assumes that similar fixes often occur at similar code. Thus, finding a similar routine or its portions could allow automated program repairing tools to expand their pools

MSR'16, May 14-15, 2016, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2901739.2901759

of potential fixes. Second, in program synthesis research, genetic programming [19, 32] is used to synthesize a program via genetic algorithms involving the search space of large code corpus. Our results will shed insights on the characteristics of routines that should be explored more in the search space (e.g., with high repetitiveness and containment). Thus, the genetic programming algorithms would have higher probability of finding proper code. Automated tools, e.g., code completion and clone detection can leverage our result to suggest better code examples by exploring search spaces.

In this paper, we have the following key research questions:

RQ1. How likely a routine for a task is repeated exactly elsewhere as an entirety;

RQ2. How likely a routine is repeated as part of other routine(s);

RQ3. What percentage of a routine is repeated from other places;

RQ4. How often portions of a routine are repeated or repeated together; what is the unique set of all of such portions, and

RQ5. How the repetitiveness of (parts of) routines involving common libraries is.

We used a large data set from SourceForge consisting of 9,224 Java projects with 2.79M class files, 17.54M methods with 187M SLOCs. We limit our study to Java projects. We collected the last snapshot (version) of each project. For each project, we collected all the methods and consider each method as a routine. For each method, we parse the code and build the program dependency graph (PDG). The rationale is that a PDG has been shown to represent the semantics in source code for comparison [20]. We also extend PDGs with API elements to study Java programs with *libraries*. In PDGs, we perform alpha-renaming for variables and literals. Two routines are considered matched if their PDGs are isomorphic. If a PDG of a method m is isomorphic to a subgraph of the PDG of a method m', we consider m as *contained* within m'.

For question 3, we consider the PDG of a method as a composition of multiple *subroutines*, each of which is defined by a per-variable slicing subgraph in a PDG, i.e., a subgraph in the PDG built by *slicing* from one variable v to get all the nodes having (in)direct dependencies with v via its edges. We measure how many subroutines of each method can be found from other places. We define *composability* of a method as the percentage of such per-variable subroutines in it that match a subroutine in the search space. For question 4, for a pair of per-variable subroutines, we count the number of methods in which both of them occur and then measure the Jaccard similarity coefficient between them as the likelihood of co-occurrences of those two subroutines. For question 5, we limited to the routines with API usages for Java Development Kit (JDK) library. We made our experiments parameterized over the variables on sizes and complexity of the PDGs, the number of APIs used, etc.

Generally, our study has three novel aspects: 1) **scalability**, 2) **three dimensions of repetitiveness, containment,** and **composability**, and 3) the **semantic level**. Our key findings include:

1. Within a project, 12.1% of the routines are *repeated* with mostly 2-7 times. The program auto-repair techniques that are based on the principle "similar code has similar fixes" [51] should set the threshold of less than 7 for the occurrence frequencies of similar methods in the same project. To increase the chance to find similar code for similar fixes, the tools need to explore the code context finer-grained than the method level. The number 12.1% also reflects the level of cloned code at the method level in open-source projects.
2. As entirety, the routines are quite project-specific with 3.3% of them exactly repeating in 1-4 other projects with at most 8 times. The repeated routines across projects often involve common libraries. The auto-repair tools (e.g., GenProg [22]) have a higher probability to find a fix within the same project.

Table 1: Collected Dataset

Total projects	9,224
Total classes	2,788,581
Total methods	17,536,628
Total SLOCs	187,774,573
Total extracted PDGs	17,536,628
Total extracted subgraphs	1,615,050,988

3. 26.1% and 7.27% of the routines are *contained* in other routine(s) within a project and in different projects. Thus, 92.73% of all routines are unique across all projects. The probability to find a routine in another project is small. This result shows that we have not reached the point of convergence where all the routines as the building blocks can be found elsewhere.
4. A very small percentage of routines is contained more than 8 times in other routines. The pattern mining approaches for a method should not use a threshold of greater than 8.
5. Except for trivial (sub)routines, the complexity of (sub)routines in terms of sizes, cyclomatic complexity, control units, and nested structures does *not* affect much repetitiveness and containment. Thus, in the empirical studies on repetitiveness and containment of (sub)routines, sampling strategies on (sub)routines can be independent of their sizes and complexity. Moreover, the program auto-repair or synthesis methods can explore repeated routines with similar likelihoods at any levels of sizes and complexity if the routines are non-trivial.
6. The routines with no control and nested structure, and no API call are more likely to repeat than the ones with either of them. However, among the routines with either of them, the numbers of occurrences of those do not affect repetitiveness.
7. 14.3% of the routines have all of their subroutines repeated. 15.6% of them have at least 90% of their subroutines repeating elsewhere. This shows a promising foundation for search-based code synthesis [19, 32]. We collected 8,764,971 distinct subroutines (with 323,564 distinct JDK subroutines) as basic units for code completion/searching/synthesis.
8. Focusing further on JDK, a popular library for Java applications, we reported that a small percentage (5%) of JDK API usages are much more frequently used than all other usages across all sizes, and 25% of other JDK usages are rarely used. Code completion tools [48] could use our collection of those commonly used JDK API usages for better recommendation.

2. DATA COLLECTION AND CONCEPTS

We collected source code at the latest revisions of Java projects on SourceForge (Table 1). We filtered out the toy projects with short histories (< 50 revisions) and small numbers of files (< 50 files). Overall, we selected a large number of well-established projects with long development histories. We kept only the main trunk of the latest revision of a project since the branches have large portions of duplicate code. Let us present the concepts used in our study.

A **routine** is a portion of code that performs a specific task and independent of and is called by the remaining code within a program [55]. In programming languages, a routine is often manifested as a *procedure, function, method*, etc. A routine expresses a functionality in a program and is assigned with a name to describe the task/procedure. A routine can be viewed as the code for a method-level algorithm (i.e., an algorithm is realized as a method). A routine is an important level in a program because programmers often break

```
1  int foo (int i) {
2     int k;
3     int j;
4
5     j = 9;
6     while (j < i)
7        j = j + 2;
8
9     k = add(i,j);
10    return k;
11 }
```

Figure 1: Example of a routine

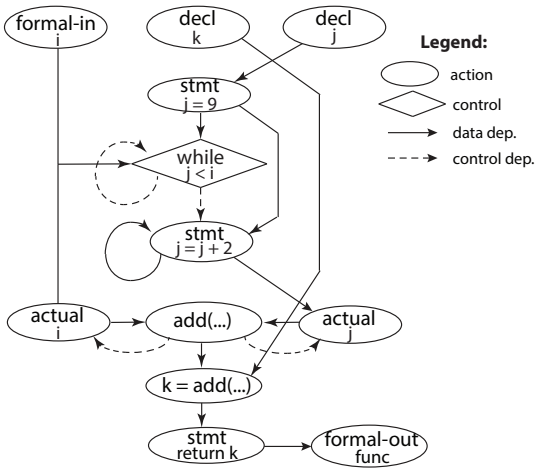

Figure 2: Program Dependence Graph (PDG) for code in Figure 1

down their program into classes, each of which in turn are broken into methods; each of them realizes a complete task. When starting to write a routine/method, they aim to have it to achieve a complete functionality. Therefore, we are interested in the repetitiveness of source code involving this level of method/routine.

2.1 Program Dependence Graph

Prior research has used *program dependence graph* (PDG) [17] to model the semantics of source code for comparison [20, 36, 30]. PDG enables an abstraction that represents the relevant statements and program entities and abstracts away the detailed syntactic differences. Thus, we use a PDG to represent the semantics of a routine.

A *Program Dependence Graph* (PDG) is a graph representation of a routine in which the nodes represent declarations, simple statements, expressions, and control points, and edges represent data or control dependencies [17].

Those declarations, simple statements, expressions, and control points are called action points and constructed from source code. A control point represents a program point where there are branches, iterations (loops), entering and exiting a routine/method. A control point is labeled with its associated program predicate.

For example, in the PDG in Figure 2 for the code in Figure 1, the regular nodes include formal-para for int i, the declaration node decl for int k, the statement node j=9, the method call add, etc. The while node is a control point and labeled with the guard expression 'j < i'.

The edges in a PDG represent the data and control dependencies between program points represented by the nodes. A *directed data dependency edge* connects two points if the execution of the second point depends on the data computed directly by the first point. For example, the node for j=9 connects directly to the node for j=j+2 because the second statement does computation involving a value that is initialized in the first statement.

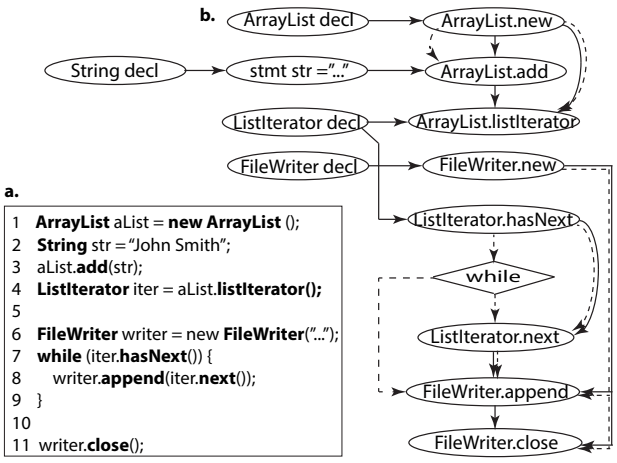

Figure 3: Enhancing PDG with API nodes and dependency edges

The node for j=j+2; has both a self data dependency edge and outgoing one because j appears in both sides of the assignment and the value of j affects the execution of the next statement.

A directed control dependency edge connects from p to q if the choice to execute q depends on the test in p. For example, the while node has a control dependency edge to the statement j=j+2 in its body. The while node also has a self control dependency edge because the test at while affects the next iteration.

In a PDG, a function call has its own node linking to the nodes for the expressions of the computation of the actual parameters, e.g., the node add connects to two actual parameter nodes for i and j with both types of dependencies. A PDG also represents the assignment of the returned value to the out parameter, e.g., to the variable k.

Since we are interested in the PDG within a function/method, we will not use the other types of nodes representing the entry, exit, function body control points, which are used to connect the PDGs for methods together to form the system dependency graph.

2.2 Extension with API Nodes

In our study, we also aim to analyze the methods involving software libraries with Application Programming Interfaces (APIs), e.g., the Java code using JDK. Figure 3 shows a code fragment that uses the APIs in JDK for the task of reading and writing data to a file. To do that, developers use API elements (or APIs for short), which are the API classes, methods, and fields provided by a framework or a library. A usage of APIs (as in Figure 3), called an *API usage*, is for an intended use to achieve a task. An API usage could involve APIs from multiple libraries or frameworks.

Since we use PDGs within methods and we match an API usage in one method to another usage in another method, we enhance the traditional PDG with *three types of nodes for three basic API usages*: 1) API **object instantiations** (e.g., new ArrayList()), 2) **API calls** (e.g., Scanner.next()), and 3) **field accesses** (e.g., LinkedList.next). Those three types of nodes are adopted from our prior work, Groum [52], an extension to PDG to support object-oriented code with libraries via APIs. Groum is also called *API usage graph representation* [52]. Note that the data and control dependencies among API variables, API calls, and field accesses are considered in the same manner as the dependencies among the other nodes in a traditional PDG.

A **usage graph** [52] is a graph in which the nodes represent API object instantiations, API calls, field accesses, and control points (i.e., branching points of control units, e.g., if, while, for). The edges represent the control and data dependencies between the nodes. The nodes' labels are from the qualified names of APIs and control units.

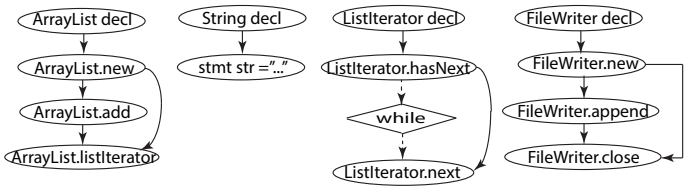

Figure 4: Per-variable slicing subgraphs in a PDG

```
1 context . select (PDG.GRAPH, PDG.FREQUENCY)
2   .from(PDG)
3   .where(CGQLDSL.nSize(PDG.GRAPH).gt(4))
4   .orderBy(PDG.FREQUENCY.desc()).limit(5).fetch();
```

Figure 5: Example of gOOQ query

Figure 3 illustrates API nodes and their edges. For clarity, we keep in the figure only the elements' names. We also keep the parameters' types and return type for a method call for matching.

For example, the nodes ArrayList.new and FileWriter.close are the action nodes representing a constructor and an API call, while the node while represents the control unit. Both data and control dependency edges connect ArrayList.new to ArrayList.add because the former method call must occur before the latter one for the ArrayList variable to be used in the latter call. Moreover, if a method call is a parameter of another, e.g., m(n()), the node for the method call in the parameter will be created before the node for the outside call (i.e., the node for n comes before that of m). The rationale is that there is a data dependency from n to m. For example, a data dependency edge connects ListIterator.next and FileWriter.append, since the former one has its return value to be used as the argument for the latter. The while node has control dependency edges to both API nodes in its body. Note that, ListIterator.hasNext in the condition of the loop must be executed before the control point while, thus, its node comes before the while node. More details on usage graphs and how to build them for methods are available in [52].

In this work, we use those API nodes and their dependency edges in a usage graph as an extension to PDG to support object-oriented source code involving APIs in libraries. For a method, we build an intra PDG. A regular function call is represented as a regular node. However, if it is an API call, a constructor call, or a field access, we will create an API node in one of those three types and its dependency edges. The dependency edges among regular nodes (e.g., statements, formal inputs, function calls) and API nodes are built as usual. For example, in Figure 3, a data dependency edge connects the statement str="John Smith" to API node ArrayList.add.

Slicing on PDG. To answer RQ3 and RQ4, we assume that a PDG for a routine can be built from subgraphs, each of which has nodes that have (in)direct data/control dependencies with a single variable via its edges. We call such a subgraph *per-variable slicing subgraph* (PVSG). To build a PVSG, we perform standard static slicing [53] on a PDG to collect for each variable v the related nodes having data and control dependencies with v to form that subgraph. Figure 4 shows the PVSGs for the PDG in Figure 3. The rationale of using PVSG with slicing is that its nodes will be interrelated via data/control dependencies, which could form a more meaningful subgraph than an arbitrary subgraph with any size in a PDG.

Normalization. In different methods, repeated code might have different variables and literal values. Thus, we need to perform normalization to remove those differences before matching. To do that, we use the same procedure as in Gabel and Su [20] for clone detection on PDG. Specifically, each statement is first mapped back to its AST node. The subtree in AST for the statement is then normalized by re-labeling the nodes for local variables and literals. For a node of a local variable, its new label is the node type (i.e., ID) concatenated with the name for that variable via alpha-renaming within the method. For a literal node, its new label is the node type (i.e., LIT) concatenated with its data type. For a PVSG, we do not need to maintain the variable's name since there is only a single variable.

Table 2: Graph operators and functions in gOOQ

Syntax	Semantics
nAction(graph)	Number of action nodes of a graph
nControl(graph)	Number of control nodes of a graph
nData(graph)	Number of data nodes of a graph
nSize(graph)	Number of nodes of a graph
nCCount(graph, label)	Number of nodes starting with label
lStartWiths(label)	Whether a graph contains node starting with a specific label
glDistance(graph1, graph2)	Number of different nodes (labels)
gMatch(graph1, graph2)	Whether a graph is isomorphic of another
gContains(GraphDesc)	Whether a graph contains another

3. EXPERIMENTAL METHODOLOGY

3.1 Graph Querying Infrastructure

To enable the querying on PDGs, we developed *graph-based Object-Oriented Query infrastructure (gOOQ)*. It was extended from the Java Object-Oriented Query framework (jOOQ) [26], to support *querying on PDGs*. Generally, jOOQ is an OO framework that allows a client Java program to place SQL queries via regular Java method invocations and field accesses. The keywords in SQL are represented by method calls such as select, from, where, and orderBy in jOOQ. The tables and fields' names are specified via objects' fields or string literals/variables. In our gOOQ, we extended jOOQ with domain-specific APIs for querying graphs. Figure 5 shows a query to list top-5 PDGs with more than 4 nodes.

To support PDGs, we added to jOOQ a new set of graph operators (Table 2). The operators gMatch, glDistance and gContains are used to search for graphs that exactly match, resemble, or contain a given graph. To enable the description of a graph in a query, we use dot [1], a text graph description language. More details can be found in [3].

3.2 Vector Representation

In gOOQ, we use our prior work Exas [50], a vector representation for graphs. Exas can approximate the structure within a graph. A (sub)graph is characterized by a vector whose elements are the occurrence counts of the selected structural features within the (sub)graph.

Exas considers two kinds of structural information in a (sub)graph, called *(p,q)-node* and *n-path*. A (p,q)-node is a node having p incoming and q outgoing edges. An n-path is a directed path of n nodes, i.e. a sequence of n nodes in which any two consecutive nodes are connected by a directed edge. *Structural feature* of a (p,q)-node is the label of the node and two numbers p and q. For an n-path, it is a sequence of labels of nodes and edges along the path.

We use the occurrence-count vector of the features extracted from a (sub)graph as its characteristic vector. Table 3 partially shows the indexes of the features, which are global across all vectors, and their occurrence counts for the graph in Figure 3b. The vector is (1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...). We can choose n as the diameter of the graph of a method. Thus, the length of a vector is equal to the number of all possible n-paths and (p,q)-nodes. In [50], we proved:

THEOREM 1. *If graph edit distance of G_1 and G_2 is λ, then $\|v_1 - v_2\| \leq \|v_1 - v_2\|_1 \leq (2P+4)\lambda$ with $P = \sum_{l=1}^{N} l.b^{l-1}$.*

G_1 and G_2 are two subgraphs of G. b is the maximum degree of nodes in G (i.e., branching factor), and N is the maximum size of

Table 3: Example of n-path features and indexes

Feature	Index	#	Feature	Index	#
StrDdecl	1	1	ArrDecl-ArrNew	9	1
ArrDecl	2	1	ArrNew-ArrAdd	10	1
LIDecl	3	1	StrDecl-StrAsn	11	1
FWDecl	4	1	StrAsn-ArrAdd	12	1
ArrNew	5	1	ArrAdd-ArrLI	13	1
StrAsn	6	1	LIDecl-ArrLI	14	1
ArrAdd	7	1	LIDecl-LIhasNext	15	1
ArrLI	8	1	FWDecl-FWNew ...	16	1

n-paths of certain sizes. This result means that, the vector distance of two subgraphs is bounded by their edit distance, i.e. similar subgraphs (having small edit distance) will have small vector distance.

THEOREM 2. *Two isomorphic graphs have the same feature set, thus, have the same vector.*

THEOREM 3. *If a graph A is a subgraph of a graph B, then the vector of A is also a sub-vector of the vector of B. A vector v is called a sub-vector of another vector v′ if all occurrence-counts in all elements of v are smaller than or equal to those of v′.*

3.3 Matched and Contained Routines

a. Repeated Routines. *Two routines are considered as repeated* if their PDGs are exactly matched after normalization. Unique routines are those with unique PDGs, which do not match with other PDGs. The number of **repetitions** of a routine A is the number of *other* repeated routines whose PDGs match with its PDG.

DEFINITION 1 (REPETITIVENESS OF A ROUTINE). *Repetitiveness is measured by the percentage of the repetitions of that routine over the total number of routines in the search space under study.*

Examples of search spaces are the entire corpus or the set of routines with a certain size. Repetitiveness of a routine A represents the percentage of the routines (in the search space) that are the repeated routines of A. The higher the repetitiveness of A, the higher chance we can find a repeated routine for A. If A and B are repeated routines, each will be counted toward the repetitiveness of each other. We also need a definition for repetitiveness of all routines in a set to compare the repetitiveness of a set with that of another, e.g., a set of routines with control nodes and another set without them.

DEFINITION 2 ((AGGREGATE) REPETITIVENESS OF A SET). *Aggregate repetitiveness of all routines in a set S with a criterion is measured by the percentage of the routines repeated (at least once) over all routines in S in the search space.*

Two isomorphic graphs have the same vector. However, even two vectors of two graphs are the same, they still might be different. Thus, we hash the PDGs with the same vectors into the same bucket using LSH [7], a vector hashing algorithm. Then, our algorithm for gMatch compares the graphs in the same bucket by the graph isomorphism algorithm, Ullman's [58], to find the matched graphs.

b. Containment. A routine appears as part of another routine if the PDG of the first one is *isomorphic* to a subgraph of the PDG of the second routine. In our containment checking function, we also build vector representations for PDGs and hash them into buckets using LSH [7]. The vectors of all the buckets are then compared to find the pairs of buckets (b_1, b_2) in which the vector for one bucket is a sub-vector of another bucket. Then, we perform pairwise matching between every PDG in b_1 and that in b_2 to find the real isomorphic subgraphs among PDGs in b_1 and b_2 using Ullman's algorithm [58].

The degree of **containment** of *a routine* and *of a set of routines* are defined in the same manner as the repetitiveness except that the relation considered between routines now is containment, instead of "repeated" (B contains A, i.e., A is contained in B).

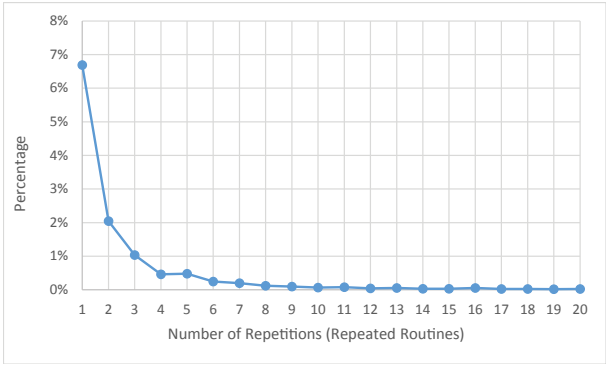

Figure 6: % of entire routines realized elsewhere within a project

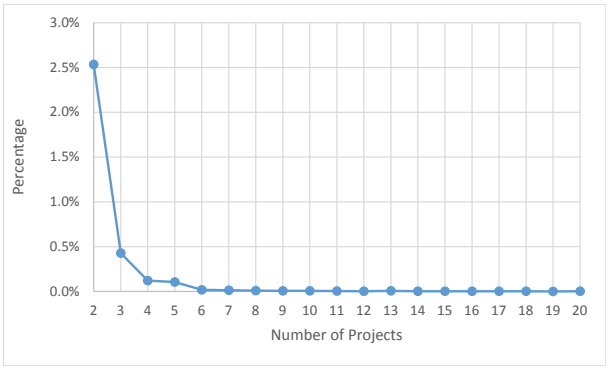

Figure 7: % of entire routines realized in more than one project

DEFINITION 3 (CONTAINMENT OF A ROUTINE). *The degree of containment of a routine is measured by the percentage of the routines contained in other routines elsewhere over the total number of routines in the current search space.*

3.4 Per-variable Slicing for Subroutines

In this study, we consider the PDG of a method as the composition of multiple portions, each of which is built by slicing the PDG to get a subgraph for a variable. We call each portion a **subroutine**. We measure how many subroutines of each method are repeated.

DEFINITION 4 (COMPOSABILITY). *Composability of a routine r is defined via the percentage of the per-variable subroutines in r that match a subroutine in the current search space. We also measure the percentage of a routine repeated elsewhere in term of the nodes in those subroutines.*

For co-occurring subroutines, for each pair of them, we determine the number of methods in which they co-appear, and the number of methods in which only one of them appears. We compute the sharing portion using Jaccard index [24]. It equals 0 if there is no sharing and 1 if two subroutines co-occur in all methods using them.

4. REPEATED ENTIRE ROUTINES (RQ1)

4.1 Routines Repeated Within a Project

First, we study the repetitiveness within a project. Figure 6 displays the repetitiveness of a routine within a project. As seen, 6.7% of the routines in the dataset repeat exactly once within a project; 2% of them repeat twice; 1.1% of them repeat 3 times, etc. The percentages of routines repeat more than 7 times are less than 0.1%. Within a project, 12.1% of the routines are repeated with mostly 2-7 times.

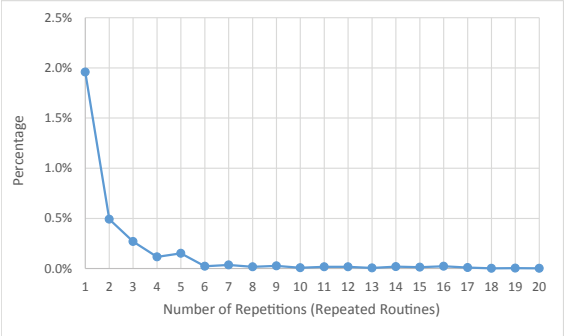

Figure 8: % of entire routines realized elsewhere in other projects

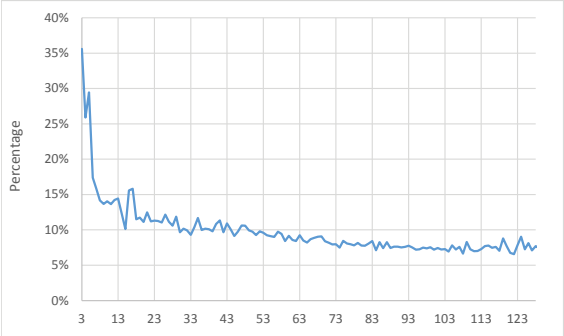

Figure 9: Repetitiveness by graph size ($|V|+|E|$) in PDGs

Implications. The program auto-repair techniques [51] that aim to find similar fixes from similar code should set the threshold of less than 7 for the occurrence frequencies of similar methods in the same project. The result of 12.1% is also consistent with a report by Roy and Cordy [56] that said cloned code at function level within a project is 7.2–15%. This shows an opportunity for clone detection/management tools at the method level.

4.2 Routines Repeated across Projects

Figure 7 shows the percentage of entire routines realized in more than one project. As seen, 2.53% of all routines in the dataset repeat in exactly 2 projects. Only 0.43% of the routines repeat in 3 projects. Figure 8 shows the repetitiveness of routines across projects.

Implications. Despite similar trends in Figure 6 and Figure 8, the actual percentages of routines repeated across projects are smaller than those repeated within a project, i.e., *as entirety, routines are quite project-specific*. 3.3% of them repeat at most 8 times across projects.

Examining the reasons for such repetitiveness, we found that those repeated routines across projects often involve the common APIs, e.g., JDK. We will give examples on such repeated routines in Section 8. Another type of repeated routines involves common control flows, e.g., *"checking a condition to break out of a loop"*:

```
1 for (init; expr1; update) {
2   if (expr2) break;
3   expr3;
4 }
```

As an implication, the program auto-repair tools could have higher probability to find a fix within a project. The fixes to incorrect usages of common API libraries could be found across projects.

4.3 Repetitiveness by Complexity in PDGs

Next, we measure repetitiveness of sets of routines (Definition 2) by the complexity of PDGs. We consider all routines in all projects.

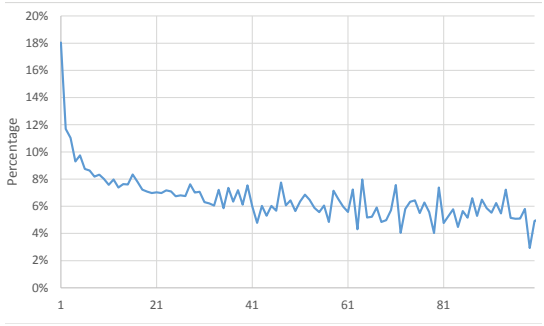

Figure 10: Repetitiveness by cyclomatic complexity

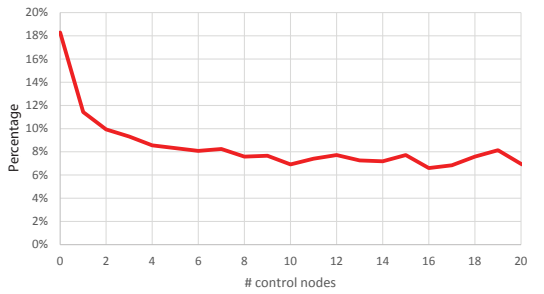

Figure 11: Repetitiveness by number of control nodes in PDGs

4.3.1 Repetitiveness by Graph Properties of PDGs

We measured (aggregate) repetitiveness (Definition 2) of a set of routines by the *size of the PDGs in term of nodes and edges*. Figure 9 shows the percentage of repeated routines that have different sizes. As seen, the routines with small sizes (3-5 nodes and edges), which correspond to a trivial PDG with a couple of statements and formal arguments, are more repetitive than the routines with larger sizes. We found that they correspond to many getters/setters or a routine whose body contains exactly a method call. Except those trivial routines, repetitiveness is not affected much by the size of the PDG.

4.3.2 Repetitiveness by Code Complexity

Cyclomatic Complexity. Figure 10 shows the percentage of repeated routines by their cyclomatic complexity, which is measured as $M = |E| - |V| + 2 * |P|$ where $|V|$, $|E|$, and $|P|$ are the numbers of nodes, edges, and connected components in the CFG of a routine. This graph has the same trend as the one in Figure 9. At the smaller complexity levels, the repetitiveness of routines is higher, however, the routines themselves are quite trivial. The repetitiveness does not change much as cyclomatic complexity increases.

Number of Control Nodes. The number of control nodes in a PDG is also an indicator of a routine's complexity. Figure 11 shows the percentages of repeated routines among the routines with one or multiple control nodes such as for, while, if, etc. For example, about 8% of routines having 6 control nodes of any type are repeated. The trend of repetitiveness when complexity is measured by the number of control nodes is the same as the ones when we measure complexity by graph sizes (Figure 9) and by cyclomatic complexity (Figure 10).

Moreover, as shown in Table 4, the routines having control node(s) of any type are less likely to be repeated than the ones without them. The same observation can be made for individual types of control nodes. However, as shown in Figure 11, the repetitiveness of routines does not depend much on the number of control nodes.

Table 4: Repetitiveness without and without control nodes

	for	while	do	if	switch	any
With	8.5%	9.1%	9.2%	10.2%	9.0%	10.1%
Without	16.2%	15.7%	15.4%	17.7%	15.5%	18.3%

Table 5: Repetitiveness by number of nested control structures

# nested struct	0	1	2	3	4	5	6	7	8	9	10
Percentage %	15.6	9.3	10.7	7.6	9.4	8.4	8.9	7.9	7.2	7.2	7.1

Repetitiveness by Number of Nested Structures. Nested structures of control units are a good indicator for code complexity. As seen in Table 5, the routines with no nested structure are repeated the most (15.6% among all such routines). Similar to the cases of other complexity metrics, repetitiveness decreases abruptly and then does not change much as the number of nested structures increases. Overall, 9.2% of the routines with nested structures are repeated.

4.3.3 Repetitiveness by Method Calls

We also found that 11.8% of routines with method calls are repeated, while 29.4% of routines without method calls are repeated.

4.3.4 Implications to SE Applications

An interesting observation is that despite using different metrics to measure code and graph structure complexity of routines, the trend on their repetitiveness is the same (Figures 9–11). First, for the simple routines with a couple of statements in their bodies and a couple of formal arguments (graph size is less than 5), their repetitiveness is higher than more complex ones. However, except for those simple routines, the complexity does not affect much repetitiveness for other routines. Thus, as an implication, a program auto-repair tool can explore repeated routines with the similar likelihoods at any levels of sizes and complexity if the routines are non-trivial (i.e., a PDG has more than 5 nodes and edges). Moreover, in the empirical studies concerning the repetitiveness, the sampling strategies on routines can be independent of their sizes and complexity if the chosen routines are non-trivial.

As seen in Tables 4 and 5, the routines without nested structures or without control nodes are more likely to be repeated than the ones having them. However, among the routines with either of them, the repetitiveness does not depend much on the number of nested structures nor the number of control nodes in the PDGs. Thus, in the empirical studies concerning repetitiveness, the strategies for sampling the routines need to distinguish the cases of having or not nested structures and control nodes. However, it does not need to do so for different numbers of nested structures and control nodes.

5. ROUTINE CONTAINMENT (RQ2)

In this study, we are interested in degree of containment, i.e., to see how likely a routine is repeated as part of other routines.

5.1 Containment Within and Across Projects

Figure 12 displays the percentage of routines that are implemented with a PDG that is a sub-graph of a PDG of another routine(s) in some other places within the same project. There are 26.1% of the routines that are contained in some routines elsewhere in the same project. 12.8% of them are contained in exactly one routine.

Figure 13 shows the percentage of routines that are implemented as a PDG that is a subgraph of another PDG of a routine in other project(s). In total, there are only 7.27% of the routines that are

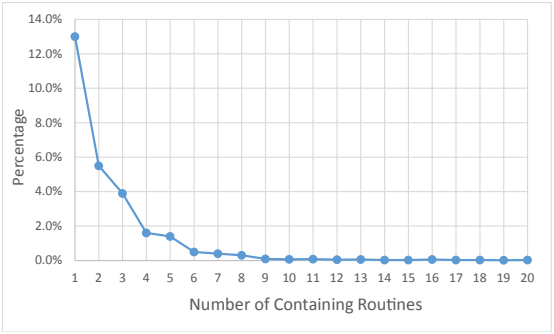

Figure 12: % of routines realized as part of other routine(s) elsewhere within a project. Horizontal axis shows number of containers.

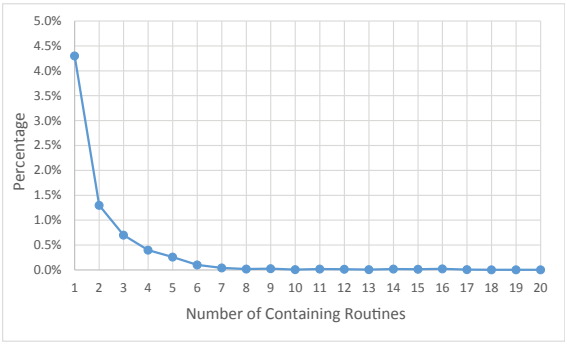

Figure 13: % of routines realized as part of other routine(s) elsewhere in other projects. Horizontal axis shows number of containers.

contained in other routine(s) in more than one projects. There are 4.3% of routines that are contained in exactly one routine in a different project. Almost all of the contained routines occur within 1–6 routines in different projects.

5.2 Containment by Complexity

We also aim to study the containment of routines by their complexity. We consider all routines in all projects.

Figure 14 shows the percentage of routines (over all routines) with different sizes that are contained in other routine(s). Figure 15 shows the percentage of routines that are contained within another one by different levels of their cyclomatic complexity.

As seen, the graphs for containment in Figures 14 and 15 exhibit the same trend as the graphs for repetitiveness. Thus, the implications listed Section 4.3.4 are also applicable to containment. For example, except for trivial routines, containment of routines is not affected much by their sizes and complexity. Thus, a program auto-repair tool could explore similar code with similar PDG with the similar likelihoods at any sizes and complexity levels if non-trivial routines are considered. In the empirical studies for containment, sampling strategies can be independent of the sizes and complexity.

5.3 Implications

First, a high percentage of routines (92.73%) are unique across all projects. That is, only 7.27% of them are contained in other routines in other projects. Thus, as developers, we have not reached the point of convergence where all the routines as the building blocks can be found elsewhere. This suggests us to explore a finer-grained unit than a routine as building blocks (Section 6).

Second, a very small percentage of routines (0.01%) is contained more than 8 times in other routines. Thus, pattern mining approaches [52] for a method should use a threshold of less than 8 occurrences.

Third, comparing Figures 6 and 12, Figures 8 and 13, we can see

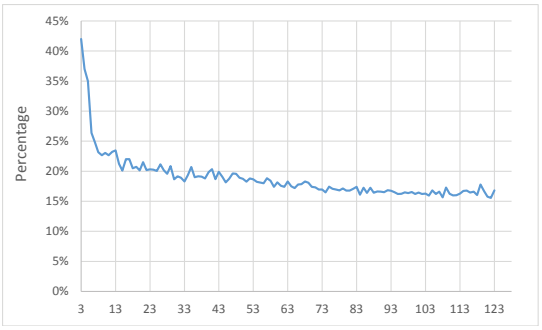

Figure 14: Containment by graph size ($|E| + |V|$) in PDGs

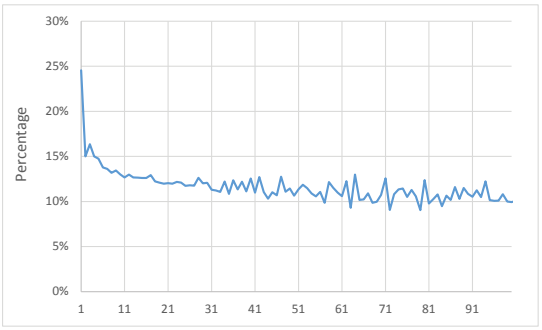

Figure 15: Containment by cyclomatic complexity

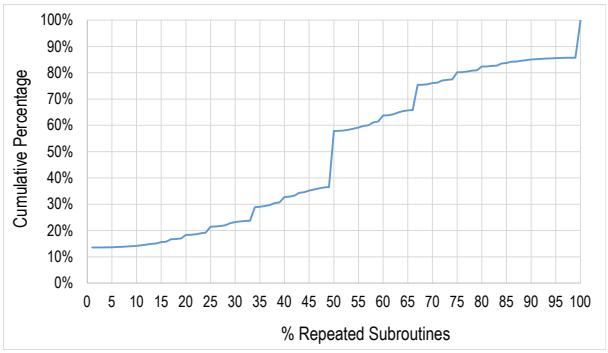

Figure 16: Cumulative distribution of routines with respect to percentage of their repeated subroutines

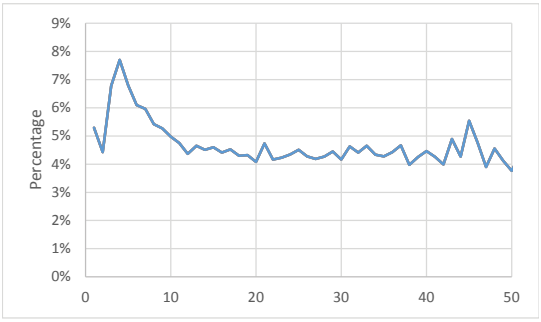

Figure 17: Repetitiveness of subroutines by size ($|V| + |E|$)

that repetitiveness and containment have the same trend (with the percentage for contained routines is higher). Moreover, there is a notable percentage of routines that are contained, but not exactly matched in other projects. This suggests that the automated tools to find a similar fix should search for the similar routines, rather than for the exactly matched ones.

6. COMPOSABILITY OF ROUTINES (RQ3)

In this experiment, we measured the percentage of subroutines in a routine that are repeated in other places. In Figure 16, 13.5% of the routines have no subroutine repeated elsewhere, i.e., **86.5% of them have at least one subroutine repeated**. 84.4% of the routines have less than or equal 90% of their subroutines having been repeated elsewhere, i.e., **15.6% of the routines have at least 90% of their subroutines repeated elsewhere**. Interestingly, there are **14.3% of the routines having 100% of their subroutines repeated**.

Implications. In the previous sections, we see that the probability to find an *entire routine* elsewhere (as exactly or as part of others) in the same and different project(s) is small. That suggested us to explore the subroutine level. *This result at the subroutines provides a promising foundation on which the program synthesis approaches can rest. That is, a reasonable percentage of subroutines in terms of PDG's subgraphs of a routine can be found in existing code*. Thus, in many cases, a large percentage of a routine might be constructed/synthesized from the subroutines elsewhere. In Section 8, we will explain our study on the repetitiveness/uniqueness of subroutines, and the synthesis approaches could use our collected unique subroutines as basic units for searching and synthesizing.

7. REPEATED AND CO-OCCURING SUBROUTINES (RQ4)

Next, we study the repetitiveness of subroutines, defined as PVSG and built by slicing via individual variables on a PDG. We used similar measurements as in the previous experiments except that each PVSG is a basic unit, instead of a routine. Figure 17 shows the percentage of repeated subroutines over the total subroutines with respect to different sizes.

Implications. As seen, the small subroutines are repeated more. However, when considering non-trivial subroutines with 10 or more nodes and edges in their PDGs, we can see that their repetitiveness does not change much when size varies. That is, such subroutines have equally repeated in term of percentages over the total subroutines at certain sizes. This phenomenon for subroutines is similar to that of the repetitiveness and containment of entire routines (Sections 4 and 5). Thus, the implications listed in Section 4.3.4 are applicable to subroutines.

Compared to repetitiveness of entire routines (Figure 9), that of subroutines is smaller due to the much larger numbers of subroutines at certain sizes. The average size of repeated subroutines is 4.3.

Among 9,269,635 subroutines, 5.4% of them are repeated. *The program synthesis tools could use our collection of 8,764,971 distinct subroutines as basic units for searching and combining*. Examining the repeated ones, we found that they are mostly involved common libraries such as JDK. Some examples on repeated subroutines are shown in Table 6 and Figure 20. Thus, the code completion tools could explore those subroutines for better recommendations.

Figure 18 shows the repetitiveness of subroutines involving JDK. As seen, subroutines (with JDK APIs) with smaller sizes are more repetitive. For larger sizes (>10), the repetitiveness of subroutines just slightly changes. In general, the percentages of repeated subroutines with JDK are higher than the ones for all subroutines shown in Figure 17. The average size of repeated JDK subroutines is 8.7. Generally, 28.6% of JDK subroutines are repeated; that number is much higher than that of general subroutines. Our collection of distinct 323,564 JDK subroutines can be used as basic ones for synthesis and code completion tools.

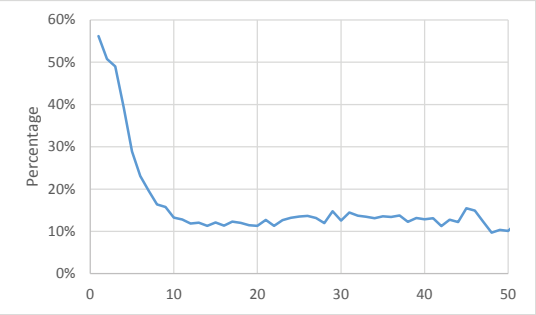

Figure 18: Repetitiveness of JDK subroutines by size ($|V|+|E|$)

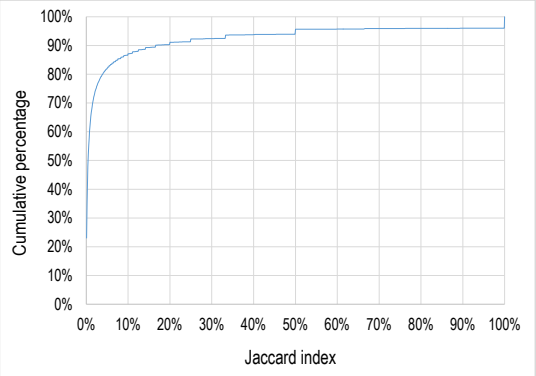

Figure 19: Cumulative distribution of pairs of subroutines w.r.t. their Jaccard indexes

We are also interested in the subroutines that frequently go together. If a pair of subroutines occurs in the same routine frequently, they can be used to improve efficiency of code search and suggestion tools. Figure 19 shows the cumulative distribution of co-occurring pairs according their Jaccard indexes. As seen, in 87% of the pairs, Jaccard indexes are less than 10%, i.e., the pairs of subroutines co-occur in a small number of routines, in comparison to the total number of routines containing each subroutine. Only 6.1% of the pairs have Jaccard indexes higher than or equal to 50%. **4% of them (115,034 pairs) have Jaccard indexes of 100%, i.e., those pairs of subroutines always co-occur in all methods using them. Thus, if seeing one routine, a tool can suggest the other routine.**

We wrote a tool to check them and found that in 101,792 pairs, the two subroutines in a pair are used together in only one method. Interestingly, **13,242 pairs always go together in multiple methods**. Table 6 lists some pairs with high Jaccard indexes. For example, the first subroutine involves a `XMLStreamWriter` variable with its functions to get/set a prefix and write the namespace. Thus, the subroutine for that variable has been used with the subroutine that uses `NamespaceContext` in 2,125 methods.

8. REPETITIVENESS OF JDK APIS (RQ5)

This section describes our study on the repetitiveness of the code involving JDK. First, we collected the subgraphs involving JDK APIs by performing slicing on a PDG to get JDK elements and dependent nodes/edges with one or multiple variables. Then, we collected the connected subgraphs in those subroutines with different sizes. Let us call such subgraphs *JDK usages* since they involve JDK APIs. Table 7 shows the statistics on the frequencies of JDK usages. A row shows the percentage of JDK usages. For example, 25% of all JDK usages with size 2 have occurred at least 8 times.

Implications. First, comparing the first row to others, we can see that a **small percentage (5%) of JDK usages are much more frequently used** than all other JDK usages across all sizes. Figure 20 displays a sample set of those popular JDK usages. The result implies that *the tools such as auto-completion, pattern mining, auto-patching, could focus on that small percentage of heavily used JDK usages, rather than evenly selecting from the entire pool of usages*.

Table 6: Frequent (Sub)routines and Co-occurring Routines

Subroutine	Co-occuring Subroutine	Freq
XMLStreamWriter#var XMLStreamWriter.getPrefix XMLStreamWriter.getNamespaceContext XMLStreamWriter.writeNamespace XMLStreamWriter.setPrefix IF XMLStreamWriter.getNamespaceContext	NamespaceContext#var NamespaceContext .getNamespaceURI	2,125
org.omg.CORBA.TypeCode#var org.omg.CORBA.TypeCode.equivalent	org.omg.CORBA.INTERNAL#var org.omg.CORBA.INTERNAL.new	300
java.security.cert.X509Certificate .checkValidity	java.security.Principal#var java.security.Principal.equals IF	188

Table 7: Statistics on frequencies of JDK API usages

	Size									
	1	2	3	4	5	6	7	8	9	10
5%	1,832	94	35	18	12	10	8	8	6	5
25%	53	8	5	4	3	2	2	2	2	2
50%	8	3	2	2	2	2	2	2	1	1
75%	2	2	1	1	1	1	1	1	1	1
95%	1	1	1	1	1	1	1	1	1	1

The cell c at k% row means k% of usages occurring **at least** c times

Second, we can see that for that those 5% popular JDK usages are quite highly repeated even at larger sizes. For example, 5% of the usages with size 6 have repeated at least 10 times. Finally, in contrast to the 5% popular JDK usages, there are another set of least popularly used usages: about **25% of JDK usages occur only once or twice** (repeat once or no repetition). This least frequently used set requires more investigation from library designers (Figure 20).

Figure 21 shows the percentage of JDK usages repeated at various average numbers (per project) of their frequent occurrences. The shapes of graphs for different usage sizes exhibit the same trends. For each size, a **reasonably large percentage (42–80%) of JDK usages occur once per project**. Moreover, the percentage of usages repeated twice over the total number of usages (with the same size) in a project is smaller (12–22%), and that for more than 3 times is very small. Thus, *API suggestion tools should also rely on the usages across projects, rather than on one project*.

Importantly, we found that on average, the number of repeated JDK usages for each size is from 2-4 times per project (not shown). From the large numbers of popularly used usages from Table 7, we can see that the set of most popular JDK API usages (**5%**) has been **used in multiple projects**, rather than in only a few ones.

We also studied the usages of different JDK packages (Figure 22). As seen, some packages (`java.lang`, `java.util`, `java.awt`, `java.io`) have been more frequently used than others (`java.rmi`, `java.applet`).

Threats to Validity. Regarding construct validity, our dataset does not represent all code in existence. We used a large corpus in hope that it provides an accurate estimation. Our dataset is large enough since we report quite similar trends with very little fluctuation despite of different metrics. Our methodology does not consider the time of creation of routines and the reasons of repetitions. However, we just aim to study repetitiveness, rather than actual code reuse. Moreover, we use only *intra* PDG. We argue that a method with its PDG is a *basic unit* to achieve a task in a program. Another point is that to measure composability, we break a PDG into subgraphs

```
Most Frequently Used APIs in JDK
Size 1
java.lang.String.equals; java.io.PrintStream.println ;
java.lang.StringBuffer.append; java.awt.Container.add ;...
Size 2
java.lang.StringBuilder#var java.lang.StringBuilder.append;
java.util.Iterator#var java.util.Iterator.next;
java.util.Map#var java.util.Map.get;
java.lang.Object#var java.lang.Object.getClass;
java.util.Map#var java.util.Map.put; ...
Size 3
java.lang.String.equals IF RETURN;
java.util.Iterator.hasNext WHILE java.util.Iterator.next;
java.util.Iterator.hasNext FOR java.util.Iterator.next;
java.io.File#var java.io.File.exists IF; ...

Least Frequently Used APIs in JDK
java.util.Scanner.locale ; java.sql.SQLInput.readAsciiStream;
java.sql.SQLInput.readRef; javax.persistence.OneToOne.optional;
org.omg.CORBA.WrongTransactionHelper.read;
javax.time.calendar.format.DateTimeFormatterBuilder.parseStrict ;...
```

Figure 20: Most and least frequently used JDK APIs

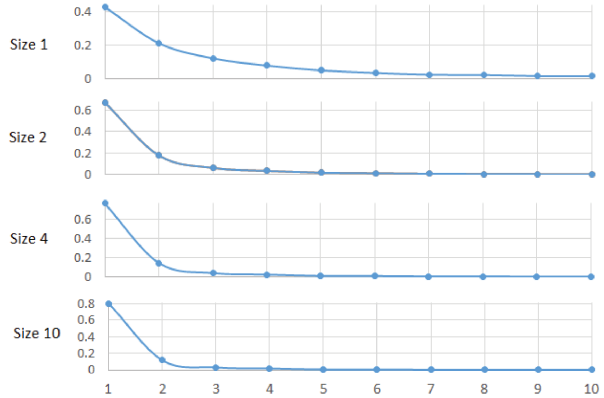

Figure 21: Percentage of JDK usages repeated at various average numbers from 1–10 (per project) of their frequent occurrences

built by slicing for a variable. Such subgraphs might not represent well the basic units in a method for program synthesis. However, we aim to study the repetitiveness of parts of routines, rather than the feasibility to synthesize a routine. Regarding external validity, we studied only Java projects and those in other languages might exhibit different repetitiveness. The characteristics of open-source projects in terms of code reuse might affect repetitiveness.

9. RELATED WORK

There are several empirical studies on the **repetitiveness of source code**. Early research shows that a significant percentage (7–23%) of the source code in a project has been cloned [8]. Roy and Cordy [56] studied on 15 open-source projects and reported that 7.2–15% of code is clones at the function level. Our study is in a much larger scale. Moreover, we look at the PDG, rather than comparing only syntactic units as in their study. At the file level, Mockus *et al.* [45, 46] study on 13.2M source files, and report more than 50% of the files being used in multiple projects. At a finer granularity, Kapser and Godfrey [28] reported that up to 10–15% of source code in a project can be code clones. Gabel and Su [21] conducted a large study on uniqueness of source code at the token level. They reported that at the granularity level of 6 tokens, 50–100% of the code of a project is repeated. Hindle *et al.* [23] compute the cross-entropy for source code to show that code is repetitive at the lexical level. Barr *et al.* [10] reported a high degree of graftability of code changes, providing a foundation for program auto-repairing.

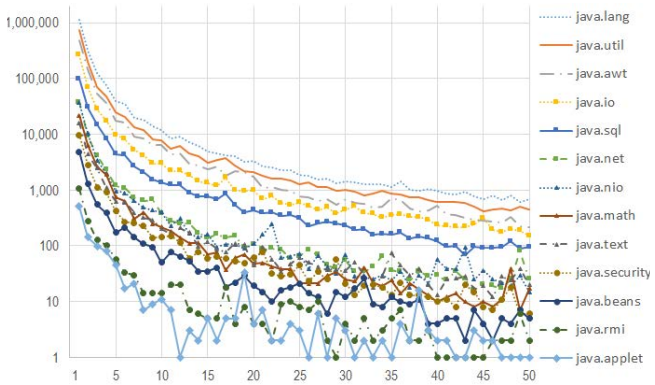

Figure 22: Usage comparison in JDK packages. The Y-axis shows the numbers of distinct usages occurring with specific frequencies.

Our prior study on **repetitiveness of changes** is at the AST level [49]. Our API usage graph (Section 2.2) and vector representation (Section 3.2) are re-used from our prior work [52, 50]. The graph query infrastructure (Section 3.1) was built for this study.

Our study is also related to **clone detection** [57, 12]. Clone detection approaches are classified based on their code representations. The typical categories are text-based [16, 38], token-based [9, 27, 34, 42], tree-based [11, 25, 18], and graph-based [31, 36]. Many clone detection tools focus on individual projects, rather than across projects as in our study. There have been several studies on clone changes [29], cloning across projects [5], API usages [59, 35, 47].

GPLAG [36] detects cloned code via mining from the PDGs with an approximated subgraph searching with a statistical lossy filter to prune the search space. Duplix [33] finds similar subgraphs in the PDGs to detect clones. Their approximated algorithm was run on 13 projects with 2K-24KLOCs. Komondoor and Horwitz [31] use program slicing and graph matching on PDGs. To scale up, we used hashing on vectors before pairwise comparison. In contrast, Gabel and Su [20] map the PDG subgraphs to structured syntax and reuse Deckard [25] to detect clones in AST. Portfolio [40] is a tool to find relevant functions and their usage. Mendez *et al.* [43] studied the diversity in how classes in API libraries are used.

Several approaches use the data structures such as pairs, sets, trees, and graphs to model abstractions in code and then **detect patterns in API usages** and examples [41, 47, 44, 14, 39]. Deterministic pattern mining methods are used, e.g., mining frequent pairs, subsequences [60, 4, 61], item sets [13], subgraphs [52, 15], association rules [37].

10. CONCLUSION

This paper presents a large-scale study on the repetitiveness, containment, and composability of source code at the routine level. We found that within a project, 12.1% of the routines are repeated. As entirety, the routines are quite project-specific with only 3.3% of them being exactly repeated. We also found that 26.1% and 7.27% of the routines that are implemented as part of others within and in other projects, respectively. Except the trivial ones, the complexity of the routines does not affect much their repetitiveness and containment. Moreover, 14.3% of routines have all of their subroutines repeated. We provide implications and practical use of our findings to the automated tools. Our data and results are available at [2].

11. ACKNOWLEDGMENTS

This work was supported in part by the US NSF grants CCF-1518897, CNS-1513263, CCF-1413927, CCF-1320578, CCF-1349153, TWC-1223828, CCF-1018600, and CCLI-0737029.

12. REFERENCES

[1] The dot language. http://www.graphviz.org/doc/info/lang.html.

[2] Large-scale empirical study on routines. home.engineering.iastate.edu/~anhnt/Research/Routine/.

[3] Usagebank website. http://home.engineering.iastate.edu/~anhnt/Research/UsageBank/.

[4] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE '07: Proceedings of the ACM SIGSOFT symposium on The foundations of software engineering*, pages 25–34. ACM, 2007.

[5] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *ISESE*, pages 376–385, 2005.

[6] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 281–293, New York, NY, USA, 2014. ACM.

[7] A. Andoni and P. Indyk. E2 lsh 0.1 user manual. http://web.mit.edu/andoni/www/LSH/manual.pdf.

[8] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering*, WCRE '95. IEEE Computer Society, 1995.

[9] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Comput.*, 26(5):1343–1362, Oct. 1997.

[10] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317. ACM, 2014.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98. IEEE CS, 1998.

[12] S. Bellon, R. Koschke, M.-G. Antoniol, M.-J. Krinke, and M.-E. Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[13] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222. ACM, 2009.

[14] R. P. L. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792. IEEE Press, 2012.

[15] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Trans. Softw. Eng.*, 34(5):579–596, Sept. 2008.

[16] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance*, ICSM '99, pages 109–118. IEEE CS, 1999.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[18] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, Nov. 2007.

[19] E. M. Fredericks and B. H. Cheng. Exploring automated software composition with genetic programming. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '13 Companion, pages 1733–1734, New York, NY, USA, 2013. ACM.

[20] M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[21] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 147–156. ACM Press, 2010.

[22] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[23] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 837–847. IEEE Press, 2012.

[24] Jaccard index. http://en.wikipedia.org/wiki/Jaccard_index.

[25] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[26] Java object oriented querying. http://www.jooq.org/.

[27] 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.*, 28(7):654–670, July 2002.

[28] C. J. Kapser and M. W. Godfrey. "cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, Dec. 2008.

[29] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196. ACM, 2005.

[30] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56, London, UK, 2001. Springer-Verlag.

[31] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis*, SAS '01, pages 40–56. Springer-Verlag, 2001.

[32] J. R. Koza. *Genetic programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[33] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, WCRE '01, pages 301–309, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, Mar. 2006.

[35] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 2–11, New York, NY, USA, 2014. ACM.

[36] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 872–881, New York, NY, USA, 2006. ACM.

[37] D. Lo and S. Maoz. Mining scenario-based triggers and effects. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, ASE '08, pages 109–118. IEEE CS, 2008.

[38] A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE international conference on Automated software engineering*, ASE '01. IEEE CS, 2001.

[39] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie. Exemplar: A source code search engine for finding highly relevant applications. *IEEE Transactions on Software Engineering*, 38(5):1069–1087, 2012.

[40] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[41] C. McMillan, D. Poshyvanyk, and M. Grechanik. Recommending Source Code Examples via API Call Usages and Documentation. In *Proceedings of the 2nd Int. Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 21–25. ACM, 2010.

[42] T. Mende, R. Koschke, and F. Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *J. Softw. Maint. Evol.*, 21(2):143–169, Mar. 2009.

[43] D. Mendez, B. Baudry, and M. Monperrus. Empirical evidence of large-scale diversity in api usage of object-oriented software. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, volume 0, pages 43–52, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[44] A. Mishne, S. Shoham, and E. Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016. ACM, 2012.

[45] A. Mockus. Large-scale code reuse in open source software. In *Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, FLOSS '07. IEEE CS, 2007.

[46] A. Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, MSR'09, pages 11–20. IEEE CS, 2009.

[47] E. Moritz, M. Linares-Vasquez, D. Poshyvanyk, M. Grechanik, C. McMillan, and M. Gethers. Export: Detecting and visualizing api usages in large source code repositories. In *Proceedings of the 28th International Conference on Automated Software Engineering*, ASE'13, pages 646–651. IEEE, 2013.

[48] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 69–79. IEEE Press, 2012.

[49] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proceedings of the 28th International Conference on Automated Software Engineering*, ASE, 2013.

[50] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. In *FASE '09*, pages 440–455. Springer Verlag, 2009.

[51] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 315–324. ACM, 2010.

[52] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[53] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *SIGPLAN Not.*, 19(5):177–184, Apr. 1984.

[54] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 87–102, New York, NY, USA, 2009. ACM.

[55] Routine. http://stackoverflow.com/questions/6885937/whats-the-technical-definition-for-routine.

[56] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 81–90, Washington, DC, USA, 2008. IEEE Computer Society.

[57] 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.*, 74(7):470–495, May 2009.

[58] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, Jan. 1976.

[59] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang. Mining succinct and high-coverage api usage patterns from source code. In *Mining Software Repositories (MSR), 10th IEEE Working Conference on*, pages 319–328, May 2013.

[60] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC-FSE '07: Proceedings of the ACM SIGSOFT symposium on The foundations of software engineering*, pages 35–44. ACM, 2007.

[61] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)*, pages 318–343. Springer-Verlag, 2009.