Artefact Relation Graphs for Unit Test Reuse Recommendation

Robert White*, Jens Krinke*, Earl T. Barr*, Federica Sarro*, Chaiyong Ragkhitwetsagul†
*University College London, UK
†Mahidol University, Thailand

***Abstract*—The reuse of artefacts is fundamental to software development and can reduce development cost and time as well as improve the quality of the output. For example, developers often create new tests from existing tests by copying and adapting them. However, reuse opportunities are often missed due to the cost of discovering suitable artefacts to reuse.**

Development artefacts form groups that have both internal connections between artefacts of the same type, and cross-group connections between artefacts of different types. When a pair of artefact groups are considered, the cross-group connections form a bipartite graph. This paper presents RASHID, an abstract framework to assist artefact reuse by predicting edges in these bipartite graphs. We instantiate RASHID with RELATEST, an approach to assist developers to reuse tests. RELATEST recommends existing tests that are closely related to a new function and can, therefore, be easily adapted to test the new function. Our evaluation finds that RELATEST's recommendations result in an average 58% reduction in developer effort (measured in tokens), for 75% of functions, resulting in an overall saving of 43% of the effort required to create tests. A user study revealed that, on average, developers needed 10 minutes less to develop a test when given RELATEST recommendations and all developers reported that the recommendations were useful.

I. INTRODUCTION

Reuse is a core pillar of software development that delivers multiple benefits including an increase in development productivity and a reduction in the cost and development time of software projects. Reuse can also improve the quality of the resultant software as reused artefacts tend to be more mature and well tested than newly created artefacts.

Discovering relationships between artefacts facilitates reuse as software artefacts are linked to each other by their relationships, for example, a test for a function, or a design artefact for a requirement. Revealing new connections between existing artefacts can, therefore, be used to discover situations where artefacts may be reused. For example, an existing test may be discovered and adapted to test a new function, or an existing function may be discovered that can be adapted, or even used directly, to fulfil a new requirement. The discovered relationships may also be useful for traceability establishment, such as discovering which regulatory codes are relevant to a particular requirement.

This paper presents RASHID[1], an abstract framework for facilitating the reuse of software artefacts by modelling the relations between artefacts of two different types using what we define as an artefact relation graph: a graph with a bipartite subgraph connecting the artefacts of the two types. We also present a tool, RELATEST, that instantiates RASHID to partially automate the reuse of existing unit tests.

Test reuse was selected to serve as the practical example for this approach as many modern software systems struggle to maintain a high level of test coverage, especially as projects grow in size and complexity. This is in part due to the heavy burden that writing many unit tests places on the developers and the de-prioritisation of this work in the presence of tight release deadlines. The failure to maintain an adequate level of test coverage can result in large numbers of bugs going undetected for long periods of time, jeopardising the correctness and reliability of the system.

RELATEST assists in test reuse by using RASHID to discover existing tests that are closely related to a new function and can therefore be recommended and adapted to test the new function. When the recommended test is close to the needed test, RELATEST saves time and avoids introducing new faults that might have been introduced if the developer had written a new test from scratch.

RELATEST also provides a unique benefit over test generation tools in that the recommended tests contain human written elements that are difficult or impossible for test generation tools to produce, such as oracles and specific test inputs.

An investigation of the quality of RELATEST's recommendations showed that developers consider 65% of its recommendations to be useful and, when using the token-based edit distance to known tests as a proxy for effort, represent a 58% reduction in developer effort versus writing tests from scratch. When considering only top-ranked recommendations, the chance of being considered useful by a developer rises to 91% and the reduction in developer effort rises to 66%. When considering the rate of functions that receive recommendations, this results in a 43% reduction in the total effort required to create tests. We demonstrate this in a real-world application by using RELATEST to create twelve tests for a large open source project, yielding an average 45% saving of effort. This supports our evaluation results.

The main contributions of this paper are:

- RASHID, an abstract framework for the reuse of artefacts using artefact relation graphs.
- RELATEST, a realisation of RASHID to recommend existing tests to be reused for new functions.

[1]Rashid (meaning "guide"): the Arabic name of Rosetta, the town where the Rosetta Stone was discovered.

- An evaluation of the effectiveness of RELATEST which shows a 43% overall reduction in effort to create tests across a project.

Our evaluation artefacts are available at https://figshare.com/s/2e54394880818d6d32dc.

II. APPROACH

This section presents the approach of RASHID and its realisation in RELATEST. The overall idea of RASHID is to use existing relationships between the artefacts of two domains and the domain-intrinsic relationships between the artefacts to recommend new relationships across the two domains. For example, if two artefacts are related across domains, then a relationship to similar artefacts is recommended.

A. Example

Figure 1 shows how test reuse through RELATEST can be applied to a real-world example from the JFreeChart system. In this example, the developer has just implemented a new function (f_2) implementing `next` for a new class and requires a new unit test for it. Without RELATEST, the developer would need to either write the new test from scratch or manually search through a potentially very large codebase to find an appropriate test to reuse: both time consuming and repetitive tasks. RELATEST streamlines this process by finding f_1, a function similar to f_2 (actually implementing `next` in a different class) in the corpus and exploiting the traceability link between f_1 and t_1 (the unit test `testNext`) to recommend t_1 to the developer as the starting point for a test for f_2 (the new unit test). In Figure 1 the developer needs only to make a few edits to transform t_1 (the old unit test) into the new test t_2 to test f_2 and all but one of the changes (marked in bold) are simple type replacements or variable renames to match the new type.

B. Artefact Reuse Framework

Figure 2 illustrates RASHID, which utilises an artefact relation graph to model the intra- and inter-domain relationships for a set of artefacts over two distinct domains. The artefact relation graph is defined as $G = (V_1, V_2, E_1, E_2, E_B, E_P)$ where V_1 and V_2 are the sets of vertices representing the artefacts in domain 1 and domain 2 respectively; E_1 is the edge set that contains the edges between the vertices in V_1, and E_2 is the edge set that contains the edges between the vertices in V_2. E_1 and E_2 model the intra-domain relationships between the artefacts. E_B is the edge set that contains the edges of the bipartite subgraph, that is the edges between the vertices in V_1 and the vertices in V_2. E_B models the inter-domain connections between the artefacts. E_P is a set of predicted bipartite edges that we will construct using E_1, E_2, and E_B.

To construct E_1, E_2, and E_B, we define three binary relations (R_1, R_2, R_B), one for each of the three edge sets, such that two vertices will be connected by an edge if they satisfy the relation. For E_1: $(v, v') \in E_1$ if vR_1v' $(v, v' \in V_1)$. For E_2: $(v, v') \in E_2$ if vR_2v' $(v, v' \in V_2)$, and for the bipartite edges E_B: $(v, v') \in E_B$ if vR_Bv' $(v \in V_1, v' \in V_2)$. In the case of R_1 and R_2, the relation is defined over pairs of the same specific type of artefact. For example, for code artefacts, the relation may be similarity, where the vertices representing two functions satisfy the relation if the code similarity between the artefacts is above a certain threshold.

The relation R_B that constructs the bipartite edge set E_B is over pairs of artefacts of different types and will therefore be defined in terms of a traceability technique that establishes links between artefacts of the two types, for example naming conventions [1] for test-to-function links, or a tracing network for requirement-to-design-element links [2]. The relation would then be satisfied if the artefacts represented by the two vertices were identified as linked by the traceability technique.

After E_1, E_2, and E_B have been constructed, RASHID predicts a set of new edges E_P. The edges in E_P can reveal inter-domain artefact connections that could not be discovered otherwise. The newly revealed connections present opportunities to reuse artefacts but can also be considered as predicted traceability links and therefore used as a method for combating the missing link problem in traceability.

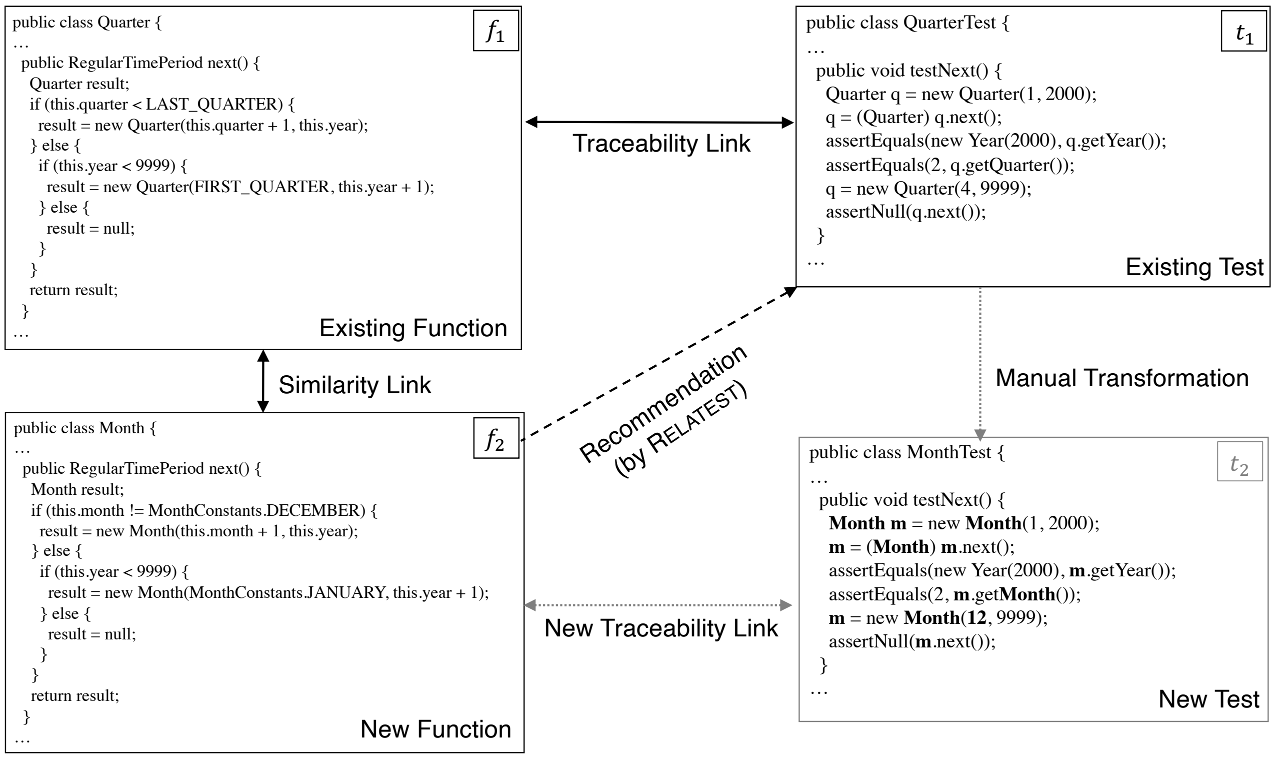
Since the edge prediction is being performed over a bipartite subgraph, the edge prediction techniques that can be applied are not limited to only those applicable in strictly bipartite graphs, as the non-bipartite edges provide extra information. This means that almost any edge prediction method can be used, including neighbourhood methods, such as Common Neighbours and Jaccard's coefficient [3], path methods, such as Page Rank [4], or supervised machine learning [5]. The edge prediction method used in Figure 2 is a neighbourhood method that predicts a bipartite edge in E_P where the addition of that edge creates a semi-bipartite 3-vertex clique, consisting of two vertices from the same domain and one vertex from the other domain. This clique forms an inter-domain triangle and this method will be referred to as the triangle method hereafter. Simplified, an edge (v, v') is added to E_P if $vR_1v''R_Bv'$.

C. RELATEST

Figure 3 shows how the RELATEST approach uses RASHID to make test recommendations for functions. RELATEST utilises a code corpus consisting of a set of unit tests $T = \{t_1, t_2, ...t_n\}$, where n is the total number of test functions, and a set of functions $F = \{f_1, f_2, ...f_m\}$, where m is the total number of functions. This corpus is used to build a set of traceability links L that maps tests to tested functions:

$$L = \{(t, f) \in T \times F \mid t \text{ tests } f\} \qquad (1)$$

When we want to make recommendations for a new function, hereafter referred to as the query function f_q, we instantiate RASHID so that V_1 contains a vertex for each function in $F \cup \{f_q\}$, and V_2 contains a vertex for each test in T. The relation R_1 for building the edge set E_1 is a similarity relation, where R_1 is satisfied by a pair of vertices $(v \in V_1, v' \in V_1)$ if the function represented by v and the function represented by v' are similar to each other above a certain threshold, formally vR_1v' if $\text{sim}(A(v), A(v')) \geq \tau$, where $A(v)$ returns the artefact represented by vertex v, $\text{sim}(a_1, a_2)$ returns the code



```
public class Quarter {
…
 public RegularTimePeriod next() {
  Quarter result;
  if (this.quarter < LAST_QUARTER) {
   result = new Quarter(this.quarter + 1, this.year);
  } else {
   if (this.year < 9999) {
    result = new Quarter(FIRST_QUARTER, this.year + 1);
   } else {
    result = null;
   }
  }
  return result;
 }
…
```

```
public class QuarterTest {
…
 public void testNext() {
  Quarter q = new Quarter(1, 2000);
  q = (Quarter) q.next();
  assertEquals(new Year(2000), q.getYear());
  assertEquals(2, q.getQuarter());
  q = new Quarter(4, 9999);
  assertNull(q.next());
 }
…
```

```
public class Month {
…
 public RegularTimePeriod next() {
  Month result;
  if (this.month != MonthConstants.DECEMBER) {
   result = new Month(this.month + 1, this.year);
  } else {
   if (this.year < 9999) {
    result = new Month(MonthConstants.JANUARY, this.year + 1);
   } else {
    result = null;
   }
  }
  return result;
 }
…
```

```
public class MonthTest {
…
 public void testNext() {
  Month m = new Month(1, 2000);
  m = (Month) m.next();
  assertEquals(new Year(2000), m.getYear());
  assertEquals(2, m.getMonth());
  m = new Month(12, 9999);
  assertNull(m.next());
 }
…
```

Fig. 1. Unit Test Recommendation using RELATEST.

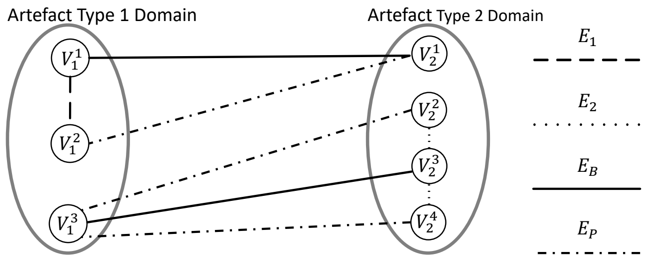


Fig. 2. Artefact Reuse Framework.

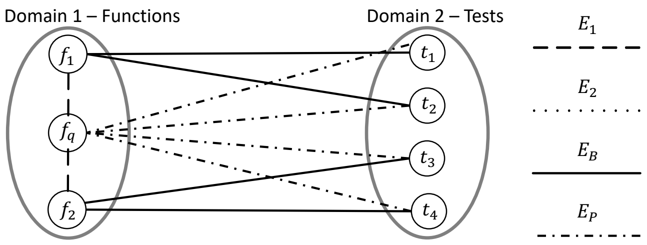


Fig. 3. Example framework instantiation for RELATEST.

similarity between artefacts a_1 and a_2, and τ is a similarity threshold. The technique used for implementing the similarity function is dependent on the implementation and is discussed in Section III-B. E_2 is assigned the empty set as in this instance we do not use intra-domain edges in the test domain.

The relation R_B used to construct the inter-domain edges E_B is defined using the traceability link set L so that a pair of vertices (v, v') where $v, v' \in V_1 \cup V_2$ satisfy the relation if the artefacts represented by those vertices are linked in L, formally, $v R_B v'$ if $(A(v), A(v')) \in L$.

Now that we have E_1, E_2, and E_B, we apply the triangle method for bipartite edge prediction, as described in Section II-B, to construct our set of predicted edges E_P. The set E_P links all functions to the tests of similar functions.

We use the completed artefact relation graph to discover recommendations by constructing a ranked list of recommendations $R(f_q)$ for our query function f_q. First, we build a list $S(f_q)$ which ranks the functions that are neighbours of f_q in the graph in descending order of similarity to f_q so that:

$$\underset{1 \le i < |S(f_q)|}{\forall} \operatorname{sim}(f_q, S(f_q)_i) \ge \operatorname{sim}(f_q, S(f_q)_{i+1}) \tag{2}$$

Where $\operatorname{sim}(f_1, f_2)$ is the similarity between f_1 and f_2.

Given the list of similar functions $S(f_q)$ and the inter-domain edges, the recommendation list is built by iterating through the elements of $S(f_q)$ and for each member $f \in S(f_q)$ constructing the set of tests T_f that are neighbours with f:

$$T(f) = \{t \in T \mid \exists (V(f), V(t)) \in (E_B \cup E_P)\} \tag{3}$$

Where $V(a)$ returns the vertex representing artefact a.

The elements in $T(f)$ are added to $R(f_q)$ until $T(f)$ has no more elements or because we also want to place an upper bound k on the size of the recommendation list, if $|R(f_q)| = k$. Otherwise, the next element in $S(f_q)$ is selected and the process continues. Once this process terminates, the recommendation list $R(f_q)$ is presented to the user.

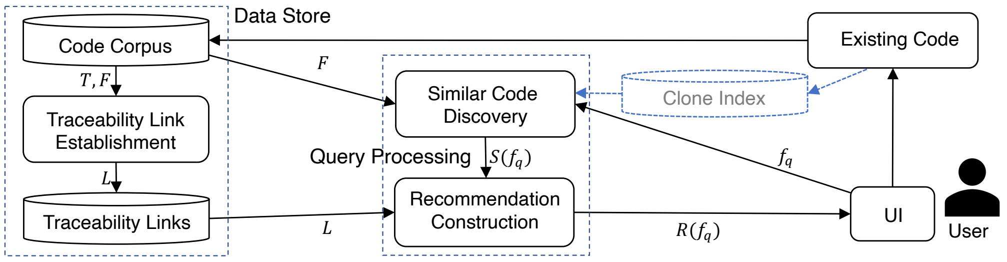


Fig. 4. System diagram. T is all tests from the corpus, F is all functions from the corpus, L is all traceability links, f_q is the query function, $S(f_q)$ is the list of functions similar to the query function, and $R(f_q)$ is the list of test recommendations for f_q.

III. Implementation

Figure 4 shows the overall architecture of RELATEST. The core of RELATEST is comprised of two component groups; the data store and the query processing. The data store builds and maintains the corpus of existing functions and tests, establishes test-to-function traceability links from the code corpus, and stores those links in the traceability link database. Query processing takes the query function, discovers similar functions, and builds the test recommendations. The UI allows the user to specify the locations of the existing code to add to the corpus, input the query function, and view the recommendations. To discover similar functions, one may rely on clone detectors that build a clone index over existing code.

One key distinction between the data store and query processing is that the data store is constructed once and only updated when the user changes the existing code. The query processing components are invoked for every individual query.

A. Data Store: Traceability Link Establishment

The correct selection and configuration of the traceability link establishment method is of high importance as it has a large impact on the performance of the system. Given this, we selected a state-of-the-art test-to-code traceability tool, TCTRACER [6], to generate the traceability links. TCTRACER implements a set of traceability techniques that use dynamic information from both the method and class levels gathered during the execution of test suites to create links between tests and tested functions along with test classes and tested classes. In RELATEST, we utilise the method level links produced using the LCS-B technique which utilises the distances between the names of tests and the names of executed functions to make predictions as to which functions are tested functions. The distance is measured based on the longest common subsequence (LCS) in both names. We chose to use LCS-B as it has high recall with good precision and high recall is important for RELATEST because if we do not have any traceability links for a test we cannot use it as a recommendation.

The first time RELATEST is run on a new project, the output from TCTRACER is parsed and then stored in the traceability link database. The traceability database is refreshed only when a project is added or removed from the corpus, ensuring that the expensive operations involved in building the traceability links are only executed when necessary.

B. Query Processing: Similar Code Discovery

The measure that we have used to implement the similarity function $\mathrm{sim}(f_1, f_2)$ is based on the Jaccard index [7]. This measure was selected as the Jaccard index is a widely accepted language-agnostic measure of similarity and a recent investigation [8] showed that textual similarity measurements can perform well on source code with modifications. Using Ragkhitwetsagul et al.'s framework [8], [9], we have confirmed that using the Jaccard index over 3-grams as implemented in the Java String Similarity library [10] performs better than most of the 30 algorithms as given in the paper (the framework reports a precision of 0.891 and an accuracy of 0.884).

The pair-wise Jaccard index method finds every existing function in the corpus that is a member of at least one traceability link and adds it to $S(f_q)$ if the similarity to the query function is greater than or equal to a certain threshold τ (function $f \in L$ is added to $S(f_q)$ if $\mathrm{sim}(f, f_q) \geq \tau$). $S(f_q)$ is then ranked by these similarity scores.

Pair-wise Jaccard index may become too expensive for a large corpus of many projects so one may substitute the pair-wise comparison with a scalable clone detector to create a clone index from which similar functions can be retrieved instantly. The corpus used in the evaluation is sufficiently small that the substitution with a clone detector is not necessary. Moreover, using Jaccard we establish a baseline which is not affected by implementation details of a clone detector as different clone detector may produce very different results.

C. Query Processing: Recommendation List Construction

Given the ranked list of similar functions $S(f_q)$ that has been constructed by the similar code discovery, the recommendation list $R(f_q)$ is constructed using the algorithm in Figure 5. The function *append*$(R(f_q), t)$ appends test t to the end of the list $R(f_q)$ and *truncate*$(R(f_q), k)$ truncates the list $R(f_q)$ to length k. This algorithm uses the ranked list of similar functions to the target function and the set of traceability links to build a recommendation list by iterating through the similar functions and, for each function, adding the tests linked to that function to the recommendation list. If necessary, the recommendation list is truncated to the given maximum length at the end of this process.

```
**Input:** The set of traceability links $L$, The ranked list
        of existing similar functions $S(f_q)$, The max
        length of recommendation list $k$
**Result:** A list of recommendations $R(f_q)$ for the
        query function
$R(f_q) \leftarrow \emptyset$;
$i \leftarrow 0$;
$(s_1, ..., s_n) \leftarrow S(f_q)$;
while $|R(f_q)| < k$ do
    foreach $t \in \{t \in T \mid (s_i, t) \in L\}$ do
        append($R(f_q), t$);
    end
    $i \leftarrow i + 1$;
end
if $|R(f_q)| > k$ then
    truncate($R(f_q), k$);
end
```

Fig. 5. Recommendation List Construction.

IV. Evaluation

We now present our research questions, the design of the experiments carried out to answer them, and the results.

a) Experimental Setup: To carry out the experiments, we first establish the sets of test-to-function traceability links for all projects using TCTRACER, as described in Section III-A. We then take the functions that are a member of at least one traceability link as the query functions – the functions we are making recommendations for. The maximum recommendation list size was set to five as it has been shown that the average person is only able to reason about five to nine different items at a given time [11]. We chose to stay at the bottom of that range as test recommendations can be relatively complex and may take some time for a developer to assess.

b) Corpus: For our corpus, we selected four well known open source projects that are written in Java and utilise the JUnit testing framework as subject projects: Commons Collections [12], Commons IO [13], Commons Lang [14], and JFreeChart [15]. These subjects were selected as they are well known, widely used, and sufficiently large to demonstrate the applicability of RELATEST to real-world systems. Three of the subject systems have been used in the TCTRACER evaluation [6] and we added Commons Collections to increase the size and diversity of the subject sample. To filter out empty tests, we define a minimum test length θ in terms of lines of source code and set it to three.

Table I gives the following information about the subjects:

- Version (Ver.): The version that was used.
- Number of Functions (F): The total number of functions.
- Number of Tests (T): The total number of JUnit tests.
- Instruction Coverage (IC): The total instruction coverage provided by the JUnit tests – measured with Jacoco [16].
- Branch Coverage (BC): The total branch coverage provided by the JUnit tests – measured with Jacoco.

TABLE I
SUBJECT STATISTICS.

Project	Ver.	F	T	IC	BC	$\|F_Q\|$
Commons Collections	4.1	4132	2819	84%	78%	1063
Commons IO	2.4	1187	1430	89%	87%	690
Commons Lang	3.7	3169	3556	95%	90%	2033
JFreeChart	1.0.19	9061	2502	52%	45%	2179

- Number of Queries ($|F_Q|$): Number of query functions for each project.

A. RQ1 (Intra-project Recommendations):

What performance is achieved by the tool when recommending tests from the same project as the query function?

We perform an investigation to evaluate the usefulness of the recommendations in an intra-project scenario, where the recommendations come from the same project as the query function. The evaluation uses the token-based Levenshtein edit distance between the recommendations and the tests linked to the query function to determine the usefulness of the recommendations. To compute this, we tokenise the code using JavaParser [17] and calculate the Levenshtein edit distance between the two tests being compared.

To perform the evaluation we use the following process for each project: First we construct the set of test-to-function traceability links L as shown in Equation (1). We then use L to construct the set of functions F_Q that are a member of at least one traceability link $F_Q = \{f \mid \exists(f, t) \in L\}$, these functions are used as the query functions. Then, for each query function $f_i \in F_Q$, we construct the set of test recommendations $T_R(f_i)$ for that function and the set of linked tests $T_L(f_i)$ for that function $T_L(f_i) = \{t \mid (f_i, t) \in L\}$. Then for each member of $T_R(f_i)$ the average edit distance to the members of $T_L(f_i)$ is computed. This provides a measure of how close to the actual tests each test recommendation is, and therefore also provides an approximation of the amount of manual effort that would be required for a developer to turn the recommendation into a working, useful test. To avoid biasing the results, we ensure that $T_R(f_i)$ does not contain any elements of $T_L(f_i)$ by skipping any tests in $T_L(f_i)$ when we build the recommendation list. Table I shows the number of query functions ($|F_Q|$) for each project.

Table II presents the results for the following measures:

- Rank (Rank): The rank in the recommendation lists that the recommendations come from.
- The Number of Recommendations (NR): The number of test recommendations made.
- Average Recommended Test Length (ARTL): The average length in tokens of the recommended tests.
- Average Known Test Length (AKTL): The average length in tokens of the known tests for the query functions.
- Median Average Edit Distance (MAED): The median of the average token-based edit distances between each recommendation and the known tests for the query function.

TABLE II
RQ1 – INTRA-PROJECT RECOMMENDATIONS RESULTS.

	Rank	NR	ARTL	AKTL	MAED	ARD	AEDSD
Commons Collections	1	752	326	326	150	46%	267
	2	622	302	319	167	52%	257
	3	514	294	314	178	57%	280
	4	420	303	300	172	58%	272
	5	344	265	296	157	53%	255
	All	2652	302	314	164	52%	256
Commons IO	1	487	266	231	81	35%	245
	2	403	243	219	114	52%	261
	3	343	240	210	110	52%	348
	4	294	220	209	109	52%	306
	5	253	196	200	113	56%	170
	All	1780	238	216	105	49%	347
Commons Lang	1	1592	264	245	59	24%	313
	2	1442	251	240	67	28%	265
	3	1340	247	235	70	30%	265
	4	1208	249	226	71	31%	272
	5	1133	246	223	76	34%	309
	All	6715	252	235	68	29%	293
JFreeChart	1	1669	250	261	85	33%	344
	2	1537	241	262	92	35%	447
	3	1417	216	263	98	37%	301
	4	1285	212	263	101	38%	355
	5	1196	238	268	125	47%	327
	All	7104	232	263	99	38%	325

TABLE III
RQ2 – INTER-PROJECT RECOMMENDATIONS RESULTS.

	Rank	NR	ARTL	AKTL	MAED	ARD	AEDSD
Commons Collections	1	754	327	326	151	46%	268
	2	625	301	319	168	52%	257
	3	519	293	314	178	57%	279
	4	425	305	300	174	58%	275
	5	350	263	296	158	53%	254
	All	2673	302	313	164	52%	256
Commons IO	1	488	266	230	81	35%	246
	2	405	242	218	114	52%	260
	3	344	240	209	111	53%	348
	4	294	220	209	109	52%	306
	5	253	195	200	113	56%	170
	All	1784	238	215	105	49%	347
Commons Lang	1	1598	245	245	60	25%	313
	2	1452	250	239	68	28%	264
	3	1347	246	235	71	30%	266
	4	1213	249	226	72	32%	272
	5	1139	246	223	77	34%	309
	All	6749	252	235	68	29%	293
JFreeChart	1	1669	261	261	85	33%	344
	2	1537	241	262	92	35%	447
	3	1417	216	263	98	37%	301
	4	1285	212	263	101	38%	355
	5	1196	238	268	125	47%	327
	All	7104	232	263	99	38%	325

- Average Relative Distance (ARD): The median average token-based edit distance (MAED) divided by the average length of the known tests (AKTL) for the query functions.
- Average Edit Distance Standard Deviation (AEDSD): Standard dev. of the average token-based edit distances.

The Median Average Edit Distance (MAED) and the Average Relative Distance (ARD) can be seen as an approximation for the effort required by the developer to adapt the recommendations to the desired tests versus writing the tests from scratch. This is discussed further in Section IV-G.

Findings: The results show that, firstly, the recommendations at rank 1 in the lists were consistently the best performing in terms of average relative distance, with a maximum of a 17% improvement over the rank 2 recommendations, and that the quality (average relative distance) of the recommendations consistently decreases as the rank increases. As every list has a rank 1 recommendation, we can say that, in the average case, if recommendations are found, the developer will only have to expend 34%[2] of the effort required to adapt the rank 1 recommendation for use, as opposed to writing a test from scratch. From looking at the number of rank 1 recommendations generated and the total number of query functions, we can see that 75% of the query functions received recommendations. Further discussion of the results and their implications is presented in Section IV-G.

B. RQ2 (Inter-project Recommendations):

What is the effect on the performance of the tool when incorporating other projects into the corpus?

[2]the average ARD using rank 1 recommendations

For RQ2 we perform the evaluation in the same fashion as for RQ1 except that, as we are now testing inter-project recommendations as well, the corpus used by RELATEST for making recommendations contains all subject projects. Table III presents the results for the same measures as in RQ1.

Findings: The results show very little difference from the intra-project recommendations in RQ1 and where differences do occur the results tend to be slightly worse more often than they are slightly better. The reasons for this are discussed in Section IV-G.

C. RQ3 (Recommendation Evaluation):

To what extent does edit distance to known tests predict the usefulness of a recommendation to a developer?

We perform an investigation to determine the relationship between edit distances and the perceived usefulness of the recommendations to a developer. We first establish a range of edit distances from 0 to 1000 and split this range into ten bands. We then organised all recommendations into their respective bands and uniformly sampled 50 recommendations from each band for manual evaluation. The recommendations were classified by a judge as either true positive (a useful recommendation), or false positive (not a useful recommendation). The classified recommendations were then used to compute the precision (true positives / number recommendation samples) for each band. As usefulness is inherently subjective, the judge had to establish some criteria by which they would determine if a recommendation would reduce the time/effort of writing a test or improve the quality of the resulting test. The criteria included looking for elements in the recommendations that exactly matched or were close to the elements that would

TABLE IV
RQ3 – MANUAL RECOMMENDATION VALIDATION.

Avg. Edit Distance	Samples	TP	FP	Precision
1–99	50	45	5	90%
100–199	50	41	9	82%
200–299	50	32	18	64%
300–399	50	38	12	76%
400–499	50	36	14	72%
500–599	50	37	13	74%
600–699	50	39	11	78%
700–799	50	33	17	66%
800–899	50	33	17	66%
900–1000	50	28	22	56%

be required in a final test. These elements included asserts, function calls, control flow statements, and object declarations/initialisations. Where an element of the recommendation did not match exactly a required element, if only a minor change was required, e.g., simply changing an identifier, type name, or value, the element was still considered useful. To cross-validate the judgements, a second judge evaluated a sample of 100 of the same recommendations and the inter-rater agreement between the judges was computed using Fleiss' kappa [18].

Findings: The results, as reported in Table IV, show firstly that the large majority of recommendations which achieve an average edit distance of less than 200 are judged to be useful, with an average precision of 86%. In contrast, the recommendations which have an average edit distance of over 900 are less likely to be judged as useful, with only 56% precision. Between these two extremes, the precision of the recommendations hovers between the mid-sixties and mid-seventies. The consequences of these results in the context of the RQ1 and RQ2 results is discussed in Section IV-G. The inter-rater agreement between the judges was $\kappa = 0.43$, which is interpreted as "moderate agreement".

D. RQ4 (Benefit of Using RELATEST):

What benefit does using RELATEST *provide in real-world test creation tasks?*

For RQ4 we conducted a user study in which we presented four unit test creation tasks to a set of developers. For each task, we asked the developers to create a JUnit test for a target function that was selected randomly from our subject projects (trivial functions such as getters and setters were ignored). The participants were provided with a dedicated interface in which, for each task, the participants were given the fully qualified name and source of the target function and anywhere between zero and five RELATEST recommendations for the target function in the ranked order produced by RELATEST. The source code for the project of the target function was also given to the participants to simulate a real development scenario and the existing tests for the target functions of all tasks were removed to stop the participants simply copying the existing tests. The participants were asked to write a test for the target function and rate any recommendations that had been provided as either useful or not useful. The ability to run the written tests and receive the output was also provided and the participants were encouraged to not submit tests that did not pass. A link to the JUnit documentation was also provided to the participants and no time limit was imposed on the tasks.

TABLE V
EXPERIENCE LEVELS OF USER STUDY PARTICIPANTS.

Programming Experience (years)	None	< 1	1–3	> 3
Total Programming Exp.	0%	7%	40%	53%
Java Programming Exp.	0%	27%	47%	27%
Industrial Programming Exp.	20%	47%	27%	7%

To evaluate the effect of using RELATEST, we split the participants into two groups and gave group one RELATEST recommendations for the first and third tasks, while group two was given RELATEST recommendations for tasks two and four. The other two tasks had to be done without a provided recommendation (tasks two and four for the first group and tasks one and three for the second group). This allowed us to compare the tests created by the participants when they had recommendations versus when they had no recommendations.

The participants consisted of 17 computer science students, 16 at masters level and one at undergraduate level, with 9 participants assigned to group one and 8 assigned to group two. We collected information on the level of experience of the participants (Table V). For the evaluation, we measured the median time taken to complete each task both with and without RELATEST recommendations. We also asked the participants to rate each recommendation as either useful or not useful.

We conducted the study over multiple sessions, using the first session as a pilot to determine if we needed to change the design. We did not deem it necessary to change the design as the results on four tasks revealed a noticeable difference. The desired outcomes and the practicalities of conducting a user study in a lab setting with supervision informed the study design, e.g., we limited the number of tasks to four to keep the participant's time commitment to reasonable 2–3 hours.

TABLE VI
RQ4 – MEDIAN TIME TAKEN IN SECONDS FOR THE TASK COMPLETION.

Task	1	2	3	4
With Recommendations	1766	549	795	708
Without Recommendations	2769	808	1324	1082

Findings: The results for the task timings in Table VI show that there is a clear difference in the median time taken to complete the task when recommendations are provided. On average, there is a 541 second difference in the median time taken to complete a task, representing a saving of almost 10 minutes to create a test when recommendations are provided. We performed a Wilcoxon signed-rank test ($\alpha < 0.05$) to assess if the time-to-complete the user study tasks with RELATEST recommendations is significantly lower than the time-to-complete them without recommendations; the difference is significant with a p-value of 0.034. The results

TABLE VII
RQ4 – DEVELOPER RATINGS OF RELATEST RECOMMENDATIONS.

Rank	1	2	3	4	5
Total Rec.	34	34	26	26	18
Useful Rec.	31	21	13	16	11
Percent Useful	91%	62%	50%	62%	61%

for the participants ratings of recommendations, as reported in Table VII, show that at rank 1, the recommendation that RELATEST judges to be best, we achieve a 91% useful rating. At lower ranks we see that percentage drop to between 50% and 62%, demonstrating that the majority of recommendations are still seen as useful at lower ranks.

E. RQ5 (Developer Opinions of RELATEST):

What are the opinions of developers that use RELATEST*?*

For RQ5, we conducted a questionnaire with fifteen of the seventeen developers that participated in the RQ4 user study. As part of this questionnaire, we asked the participants to state if they believed that the recommendations helped them complete the task and if they would use RELATEST in their typical development workflow were it to be freely available (Table IX). We also asked if the developers were already using existing tests as templates when creating a new test to determine how easily RELATEST could be adopted by developers (Table VIII). The intuition for this question is that if developers are already searching for existing tests to use as templates, it would not be a big change for them to use RELATEST and they would be more likely to adopt it. The questionnaire was administered immediately after the participants completed the user study tasks so that they were fresh in their mind. Two of the participants did not complete the questionnaire and thus were not included in the results.

Findings: The results for the questionnaire (shown in Tables VIII and IX) show that all participants believed that the recommendations were useful and that 74% would use RELATEST in normal development at least some of the time, with 47% saying they would use it most or all of the time. As for their current development practices, 74% used existing tests in some capacity when creating new tests.

F. Real-world Applicability Example

To help demonstrate applicability, we used RELATEST to create a set of new tests for a large open source system. We measured the reduction in effort achieved by using RELATEST versus writing the tests from scratch using the token-based edit distance, in the same way as the research questions. To select a suitable system to generate tests for, we used SonarCloud to create a set of projects which fulfilled the criteria of being currently active (last analysed within the previous week), having the appropriate level of test coverage (30% to 70%), and being of adequate size (over 100k lines of Java code). We then randomly selected and inspected the projects from this set to find an appropriate untested class as a target to generate new tests for. Our criteria for target class selection included looking for a class that contained approximately ten uncovered methods that were neither trivial, such as getters/setters, nor overly complex, such as graphics rendering methods. Using this process we selected the *SmsUtils* class from the District Health Information Software 2 (DHIS2) [19] project, which contains 12 such methods. RELATEST was used to generate the recommendation lists for each of these methods and the best recommendation was selected and transformed into a test for that method. These new tests were submitted to the project in a pull request, which has been merged. We computed the absolute and relative token-based edit distance, in the same manner as the research questions, between the recommendations and the final tests. This showed an average absolute edit-distance of 12 tokens and an average relative edit-distance of 0.55.

G. Discussion

We use the average relative distance (ARD) as a proxy for relative effort as there is no established method for accurately measuring required developer effort [20]. While there are instances where token-based edit distance may not translate into effort, we believe that token-based edit distance is an adequate proxy (not requiring correlation). Our user study showed no results that would invalidate our assumption.

When comparing the results from RQ1 and RQ2 to examine the impact of inter-project recommendations, the minimal difference and slight worsening of results from RQ1 to RQ2 can be accredited to the fact that the shared lexicon between functions from the same project is much larger than between functions from different projects, where the purpose and formatting of the code often differ greatly. This means that firstly, RELATEST is far more likely to make recommendations from the same system as the query function since it is using the similarity between functions to find recommendations. Secondly, this factor means that in the instances where RELATEST does make inter-project recommendations, the average edit distances are likely to be larger than for intra-project recommendations.

Another useful observation provided by the RQ1 and RQ2 results is that the performance of the rank 1 recommendations is always the best rank and the performance almost always decreases as the rank increases. This is a positive result as rank 1 always has the most recommendations and demonstrates that our approach is correct in assuming that the more similar two functions are, the better the recommendations that they generate for each other will be.

A further notable observation emerges from the average relative distance in RQ1 and RQ2:q Commons Lang has only a small reduction in performance as the rank of the recommendations increases, compared to the other projects. This indicates the number of high-quality recommendations is higher in Commons Lang, which may be indicative of a high number of similar functions in the project.

The RQ3 results give important context to the RQ1 and RQ2 results as they provide an indication as to how average edit distances translate into the usefulness of the recommendations

TABLE VIII
QUESTIONNAIRE – PREVIOUS TEST DEVELOPMENT STRATEGY QUESTION.

While developing without a test recommendation tool, do you typically write tests from scratch or use existing tests as a template?	I usually write tests from scratch	27%
	I usually use an existing test as a template	47%
	I usually write tests from scratch, but I look at other tests before	27%

TABLE IX
QUESTIONNAIRE – RELATEST USAGE QUESTION.

Overall, did the recommendations help you complete the tasks?	Yes	100%
	No	0%
If freely available, how often would you use RELATEST or a similar tool in normal development?	Always	13%
	Most of the time	34%
	Some of the time	27%
	Rarely	13%
	Never	13%

to the developer and can, therefore, give us a more complete picture of the value of using RELATEST as opposed to writing tests from scratch. For example, when considering the rank 1 results from RQ1 we can see that all of the median average edit distances for recommendations at rank 1 fall into the first and second bands (0–200), which have an average precision of 86% in the RQ3 results. Given the 75% rate of at least one recommendation being made, and the 34% average relative distance for rank 1, we can state that in 86% of the 75% of cases where at least one recommendation was generated, the developer saved 66% of development effort on average. This gives us a final figure of a 43% reduction in effort to create tests overall, even when only considering rank 1 recommendations. This represents a large amount of effort, especially over the full development cycle of a large project.

The RQ4 results reveal that the average task completion time was 36% less when recommendations were given, indicating a reduced amount of effort required to create the tests. This result is complemented by the developer ratings of the recommendations which show that 91% of the rank 1 recommendations were seen as useful and, on average, 59% of the lower-ranked recommendations were also seen as useful. The RQ5 results further demonstrate that the developers believed the recommendations, and RELATEST in general, to be useful.

The real-world applicability example demonstrated that it was possible to use the recommendations to create useful tests even when the test author has no previous expertise in the system. This is not the usual scenario in which the recommendations would be used, however, the recommendations will still be useful when used by developers for their own systems.

One important consideration to bear in mind when considering the results is the that the traceability technique used to build the test-to-function links and the used similarity measure have a direct impact on the results and if the same experiments were to be conducted with a better traceability technique or a better similarity measure, we would expect the results to improve as there will be less noise in the recommendations.

H. Threats To Validity

As the evaluation was done on a small set of projects, the results may not generalise to other projects. Moreover, only Java projects have been analysed. As the implementation relies on a third-party traceability technique, the evaluation results depend on the accuracy of the links generated by TCTRACER because developers rarely annotate their tests with the methods it is supposed to test [21].

The main threat comes from the reliance on a manual investigation for the RQ3 results. Firstly, the manual evaluation is a very time-consuming process which limited the size of the sample that we could evaluate. This is an external threat to validity as we have no clear evidence as to the representativeness of this sample. However, the fact that the subjects are large and diverse projects helps to ameliorate this threat. The use of manual investigation for evaluation also poses an internal threat to validity as there is some amount of subjectivity for what constitutes a useful recommendation. However, this risk is mitigated by our approach of firstly, establishing criteria for how recommendations should be judged and secondly, computing an inter-rater agreement between two judges using Fleiss' kappa, which shows statistically significant agreement.

There is also an external threat to validity created by the sample of participants for the user study who are all students and may not be representative of the wider developer community. However, given that all but one of the participants were graduate students at the masters level, there is a reasonably diverse range of experience levels and the majority (80%) had at least one year of industry experience, as shown in Table V.

The user study underwent Ethics Review and was approved by the UCL Research Ethics Committee. The participation was anonymous.

V. RELATED WORK

The related work breaks down into two distinct sections: recommendation systems and similarity of functions.

A. Code Recommender Systems

Code reuse is a common practice in software development as developers look to solve problems and save time by using pre-existing code from other projects and the web. Most of this reuse comes from libraries, APIs, and code snippets [22]. Due to the prevalence of reuse, a number of tools have been developed to discover and reuse existing code. Early examples of these tools include SCRUPLE [23], which locates code features using a simple pattern-based query processor.

One recent approach is Test-Driven Code Search (TDCS) which uses test suites to define the desired behaviour and ensure that the code fragments returned by a code search engine can be tested in the context of the target system. CodeGenie [24] utilises this approach by taking the designated test as input and performing a Sourcerer [25] code search using keywords, identifiers, and interface definitions. Discovered candidate functionality is extracted via slicing and presented in the plugin for testing. However, CodeGenie and TDCS, in general, cannot be used directly with RELATEST as they are geared towards a Test Driven Development (TDD) scenario where a new test is used to find an existing function.

Other approaches use textual IR methods to search code and make suggestions. One example of this is Prompter, an IDE plug-in that searches Stack Overflow discussions and recommends code fragments for developers [26]. Prompter uses the IDE context, such as the currently displayed code, to formulate a query for a Stack Overflow search with are presented to the developer. The weakness of this system in comparison to RELATEST is that the user must manually extract the code snippets and there is a high probability the snippets will not be executable without extensive modification. In comparison, RELATEST returns whole functions from existing projects.

Erfani et al.'s work [27] is related to RELATEST as their system also recommends unit tests based on the similarity between tested code. This can be shown to be an instantiation of RASHID as it is also based on the principle of utilising the relationships between artefacts in one domain (functions) to make recommendations of artefacts from another domain (tests). However, the results presented by Erfani et al. show that they were only able to make a recommendation for 8.6% of untested functions and the quality of these recommendations was not evaluated. In contrast, RELATEST makes at least one recommendation 75% of the time, saving at least 43% of developer effort in total.

Landhäußer and Tichy [28] utilises the idea of using existing tests for similar functions, with the difference that they attempt to programmatically transform the tests instead of providing them as recommendations. However, given the limited evaluation and the acknowledged issues with the approach to test transformation, we believe that RELATEST has greater applicability.

Makady and Walker [29] explore the concept of aiding the reuse of existing tests when the corresponding tested code is reused. This provides an alternative scenario in which RELATEST could be used where the reused tested function is provided as the query function instead of a new function.

Recently, machine learning techniques have been utilised to generate code recommendations by learning from existing code. TESTNMT [30] and REASSERT [31] use recurrent and transformer neural networks respectively to generate code recommendations, specifically unit tests, by learning from existing tests.

B. Similarity Measurements

Like traceability, code similarity measures have received a great deal of interest from the community. The tasks for which code similarity is useful include refactoring, fixing a bug, or performing plagiarism detection [8].

Natural language processing (NLP) can be used for measuring the similarity between code, such as in recent work by Zilberstein and Yahav [32]. Their tool, SIMON, uses NLP to establish similarity between snippets of code. Finding code snippets that are similar to a query snippet involves searching a database of code snippet to natural language description pairs to find a snippet that is semantically similar to the query snippet. The similarity of the natural language descriptions is then used to establish similarity between their respective code snippets.

The approach utilised by SIMON can also be shown to be an instantiation of RASHID defined in Section II-B as it utilises inter-domain relationships between artefacts to discover other relationships between artefacts, specifically using the two domains of code snippets and natural language descriptions to find similar code snippets. The way that SIMON instantiates RASHID is slightly different from RELATEST and [27] in that the relationships that are being searched for are intra-domain (code-snippet-to-code-snippet). In this instance E_B is the first set of edges constructed, by using a relation defined by a database of code snippet to natural language description mappings. This database is used for the R_B relation such that vR_Bv' $(v, v' \in (V_1 \cup V_2))$ if the code snippet represented by v and the natural language description represented v' are linked in the database. The intra-domain edges in the natural language description domain E_2 are constructed using textual similarity as the relation, such that vR_2v' $(v, v' \in V_2)$ if the descriptions represented by v and v' have a textual similarity above a certain threshold. The predicted edges are not the new inter-domain edges E_P, as is the case for RELATEST, but are the intra-domain edges in the code snippet domain E_1. The prediction method uses squares, specifically that an edge (v, v'), $v, v' \in V_1$, is added to E_1 if there exists a path (v, n_1, n_2, v') of length three from v to v', where $n_1 \in E_2$ and $n_2 \in E_2$.

The advantage that RELATEST has over this system for building recommendations is that RELATEST does not require a database of code snippet to natural language description pairs, which is difficult to build and maintain.

VI. CONCLUSION

We have presented RASHID, an abstract framework for assisting the reuse of existing artefacts, and RELATEST, an instantiation of RASHID that discovers existing tests which can be recommended to developers to test new functions. We have also presented an empirical evaluation of RELATEST with four real-world open source projects in two possible usage scenarios (intra and inter-projects) and through a user study. The results show that overall RELATEST can make reductions of 43% of the total amount of developer effort required to test new functions. The user study shows that, on average, developers needed 10 minutes less to develop a test when given RELATEST recommendations and all developers reported that the recommendations were useful. The results demonstrate the potential power of discovering and exploiting connections between artefacts to improve the software development process.

References

[1] B. Van Rompaey and S. Demeyer, “Establishing traceability links between unit test cases and units under test,” in *13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 209–218.

[2] J. Guo, J. Cheng, and J. Cleland-Huang, “Semantically enhanced software traceability using deep learning techniques,” in *39th International Conference on Software Engineering*, 2017, pp. 3–14.

[3] A. Nigam and N. V. Chawla, “Link prediction in a semi-bipartite network for recommendation,” *Intelligent Information and Database Systems*, vol. 9622, pp. 127–135, 2016.

[4] D. Liben-Nowell and J. Kleinberg, “The link-prediction problem for social networks,” *Journal of the American Society for Information Science and Technology*, vol. 58, no. 7, pp. 1019–1031, May 2007.

[5] N. Benchettara, R. Kanawati, and C. Rouveirol, “Supervised machine learning applied to link prediction in bipartite social networks,” in *International Conference on Advances in Social Network Analysis and Mining, ASONAM 2010*, 2010, pp. 326–330.

[6] R. White, J. Krinke, and R. Tan, “Establishing multilevel test-to-code traceability links,” in *42nd International Conference on Software Engineering (ICSE’20)*, 2020.

[7] P. Jaccard, “Étude comparative de la distribution florale dans une portion des alpes et des jura,” *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[8] C. Ragkhitwetsagul, J. Krinke, and D. Clark, “A comparison of code similarity analysers,” *Empirical Software Engineering*, vol. 23, no. 4, pp. 2464–2519, 2018.

[9] C. Ragkhitwetsagul, J. Krinke, and D. Clark, “Similarity of source code in the presence of pervasive modifications,” in *International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 117–126.

[10] “Java String Similarity – Jaccard Index,” https://github.com/tdebatty/java-string-similarity#jaccard-index, accessed: 2018-08-23.

[11] G. A. Miller, “The magical number seven, plus or minus two: Some limits on our capacity for processing information.” *Psychological review*, vol. 63, no. 2, p. 81, 1956.

[12] “Apache Commons Collections,” https://commons.apache.org/proper/commons-collections/, accessed: 2018-07-30.

[13] “Commons IO,” https://commons.apache.org/proper/commons-io/, accessed: 2018-07-30.

[14] “Apache Commons Lang,” https://commons.apache.org/proper/commons-lang/, accessed: 2018-07-30.

[15] “JFreeChart,” http://www.jfree.org/jfreechart/, accessed: 2018-07-30.

[16] “Java Code Coverage Library,” https://www.jacoco.org/, accessed: 2018-08-16.

[17] “Java Parser,” https://javaparser.org/, accessed: 2020-03-02.

[18] J. L. Fleiss, “Measuring nominal scale agreement among many raters.” *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[19] “Dhis2.” [Online]. Available: https://github.com/dhis2

[20] E. Shihab, Y. Kamei, B. Adams, and A. E. Hassan, “Is lines of code a good measure of effort in effort-aware models?” *Information and Software Technology*, vol. 55, no. 11, pp. 1981–1993, Nov. 2013.

[21] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, “EZUNIT: A framework for associating failed unit tests with potential programming errors,” in *Agile Processes in Software Engineering and Extreme Programming*. Springer Berlin / Heidelberg, 2007.

[22] R. E. Gallardo-Valencia and S. E. Sim, *Source Code Seeking on the Web: A Survey of Empirical Studies and Tools*. Lulu.com, 2014.

[23] S. Paul and A. Prakash, “A framework for source code search using program patterns,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[24] O. A. L. Lemos, S. Bajracharya, J. Ossher, P. C. Masiero, and C. Lopes, “A test-driven approach to code search and its application to the reuse of auxiliary functionality,” *Information and Software Technology*, vol. 53, no. 4, pp. 294–306, 2011.

[25] S. K. Bajracharya, J. Ossher, and C. V. Lopes, “Sourcerer: An infrastructure for large-scale collection and analysis of open-source code,” *Sci. Comput. Program.*, vol. 79, no. 79, pp. 241–259, 2014.

[26] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, “Mining stackoverflow to turn the ide into a self-confident programming prompter,” in *11th Working Conference on Mining Software Repositories*, 2014, pp. 102–111.

[27] M. Erfani, I. Keivanloo, and J. Rilling, “Opportunities for clone detection in test case recommendation,” in *37th Annual Computer Software and Applications Conference Workshops*, Jul. 2013, pp. 65–70.

[28] M. Landhäußer and W. F. Tichy, “Automated test-case generation by cloning,” in *7th International Workshop on Automation of Software Test (AST)*, Jun. 2012, pp. 83–88.

[29] S. Makady and R. J. Walker, “Validating pragmatic reuse tasks by leveraging existing test suites,” *Software: Practice and Experience*, vol. 43, no. 9, pp. 1039–1070, 2013.

[30] R. White and J. Krinke, “Testnmt: Function-to-test neural machine translation,” in *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*, ser. NL4SE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 30–33. [Online]. Available: https://doi.org/10.1145/3283812.3283823

[31] ——, “Reassert: Deep learning for assert generation,” *arXiv preprint arXiv:2011.09784*, 2020.

[32] M. Zilberstein and E. Yahav, “Leveraging a corpus of natural language descriptions for program similarity,” in *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward!*, 2016, pp. 197–211.