# An Assessment of Test-Driven Reuse: Promises and Pitfalls

Mehrdad Nurolahzade, Robert J. Walker, Frank Maurer

Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{mnurolah, walker, frank.maurer}@ucalgary.ca

**Abstract.** Test-driven reuse (TDR) proposes to find reusable source code through the provision of test cases describing the functionality of interest to a developer. Proponents claim that their TDR approaches work well. This paper presents the results of an experiment to evaluate the ability of state-of-the-art TDR tools to locate reusable source code for realistic tasks. We find that non-trivial functionality, like that needed in the daily tasks of developers, can largely *not* be retrieved by these approaches. We provide an analysis of the shortcomings and underlying problems in the existing approaches, and a discussion of potential solutions.

## 1 Introduction

Recent research considers locating and reusing source code by leveraging test cases [5, 13, 8]. Such *test-driven reuse* (TDR) is a reasonable proposition in development practices where test cases are written prior to implementing the functionality under test—called *test-driven development* (TDD)—such as in several agile development methodologies [e.g., 1]. While the TDR literature reports good results, it is not clear how the approaches compare, nor whether the results generalize to realistic TDD scenarios.

In TDR, the developer starts from test cases to indicate what they seek. The test cases are typically interpreted by automated TDR approaches as precise specifications to be satisfied by the located source code. And therein lies the crux of the potential problem: in test-driven development, the developer may have at best a fuzzy notion (initially) of the functionality she wants—TDD is inherently iterative. If a TDR approach places too much weight on the details of the test cases written (such as the particular names of types and methods), it is unlikely to find appropriate source code nor source code that can be trivially transformed to become a perfect match for those test cases.

The evaluation of TDR tools to-date [5, 13, 8] has been performed using a collection of classic tasks commonly used in the software reuse literature. Most of these tasks involve common data structures and functions, for which the developer can be expected to use the standard domain-specific vocabulary. We claim that these tasks are not representative of the daily requirements of developers performing TDD, where a developer cannot be expected to know of a standard domain-specific vocabulary.

To evaluate the existing test-driven reuse tools, we conducted an experiment with a set of trial tasks discovered from the daily development activities of developers. We found that known solutions for such tasks are largely not retrieved by the existing TDR

approaches, but instead, these approaches tend to recommend irrelevant source code. We analyze the published descriptions of the approaches and details of the returned solutions to identify the problems underlying this failure.

The rest of this paper is organized as follows. An overview of related work in TDR is provided in Section 2. We describe our experiment in Section 3. Results from the experiment and additional discussion are presented in Section 4.

## 2 Related Work

The reuse of source code through copying and modifying has a long history. While this approach of *pragmatic* software reuse can be abused, a growing body of evidence exists that it is a standard industrial practice, that it can be performed in a disciplined and principled manner, and that the use of pragmatic reuse and pre-planned reuse techniques could be coordinated to complement each other [for an overview, see 2].

A number of techniques have been used for indicating desired functionality for reuse, including signature matching [10], and specification matching [6]. Signature matching and specification matching both place a high burden on the developer to provide precise input; both approaches either will fail to retrieve pertinent source code if exact but inappropriate details are provided, or will retrieve an excessive number of results, demanding laborious examination by the developer. We note that both approaches mention the possibility of more approximate matching strategies.

Podgurski and Pierce [12] proposed behaviour sampling, an approach to retrieve reusable components using searcher-supplied samples of input/output for desired operations. As a pre-filtering step, behaviour sampling uses signature matching to limit the components to be tested, resulting in the same potential drawbacks. Podgurski and Pierce proposed two extensions to the classic form of behaviour sampling to overcome its limitations. Test-driven reuse realizes one of these extensions that permits the searcher to provide the retrieval criteria through a programmed acceptance test.

Three approaches to test-driven reuse have appeared in the literature: Code Conjurer [4, 5], CodeGenie [8], and S6 [13]. The prototype tool for each approach operates on source code written in the Java programming language.

Code Conjurer and CodeGenie are JUnit[1]-based implementations of test-driven reuse. The plug-in to the integrated development environment (IDE) provided by each of the tools automatically extracts operation signatures from the searcher-supplied JUnit test code. Search is then performed via a source code search engine and results are presented to the searcher for inspection in the IDE. The Merobase and Sourcerer [9] code search engines power Code Conjurer and CodeGenie searches respectively. CodeGenie further assists the searcher in slicing the source code to be reused; however, unlike Code Conjurer, the current implementation of CodeGenie can only be used to search for a single missing method, and not an entire class [8].

S6 complements the use of behaviour sampling with other forms of semantic specifications. It implements the abstract data type extension proposed by Podgurski and Pierce to handle object-oriented programming. Unlike Code Conjurer and CodeGenie, S6 does not utilize JUnit test code as a query language, but requires that the searcher provide the interactions of a class's methods through its web user interface. S6 attempts

[1] JUnit is an automated testing framework for Java code.

a small set of limited transformations on candidate source code, in an attempt at relaxing the constraints imposed by literal interpretation of the input query. S6 can use a local repository or remote code search engine like Google Code Search as the codebase searched through for candidate source code. Similar to Code Conjurer and CodeGenie, S6 depends on being able to find an appropriate initial result set.

All three approaches initially filter the repository on the basis of lexical or syntactic similarity with the supplied test case/query specification. All three claim to then execute the test case on each of the filtered results to assess semantic similarity as well; this acts solely to further constrain the set of potential matches.

## 3 Experiment

Our research question is "Do existing TDR tools recommend relevant source code for which minimal effort should be needed to integrate with an existing project?" In particular, we consider whether each tool is able to recommend known source code that solves a task, given modified forms of the existing test cases—taken from the same codebase—that exercise that source code.

To this end, we located a set of tasks that were discussed by developers on the web, suggesting that these were problematic for them. For each task, we also located an implementation that would serve as a solution; each implementation needed to exist in the TDR tools' repositories and have automated test cases associated with them. We detail our tasks and our selection methodology in Section 3.1.

The test cases associated with the known solutions would not be appropriate to give to the TDR tools without modification, because (1) they would immediately identify the project and classes that would serve as the solution, and (2) they would be far too detailed to represent realistic test cases drawn from test-driven development. In TDD, the developer can at best be expected to have an approximate idea of the interfaces to be exercised. Thus, the test cases needed to be altered in as unbiased manner as possible. Furthermore, because of implementation choices and shortcomings of the prototype tools, additional translations were required in order to be fair to the tools. We detail our experimental design including the test case modification methodology in Section 3.2.

After obtaining recommendations from each tool for each task, we needed to assess their suitability as solutions for the task. We utilized two subjective, qualitative measures for this purpose: relevance and effort. To improve the construct validity of our measurements, we had other developers assess a sample of our suitability assessments, and compared them with our own. We detail our and the external assessments of suitability in Section 3.3.

### 3.1 Task Selection

We located the 10 programming tasks in Table 1 for this experiment. We explain below our methodology for obtaining these.

*Sources of tasks.* As the existing TDR tools all operate on Java source code, we focused on Java-oriented sources of information. We first examined material designed for developer education, including Oracle's Java Tutorial (http://docs.oracle.com/javase/tutorial). Such tutorials are usually developed by experienced developers to teach other developers what they should know about a language or technology because they are likely to

Table 1: Brief task descriptions.

| Task | Description |
|---|---|
| 1 | A Base64 coder/decoder that can Base64 encode/decode simple text (Stringtype) and binary data (byte array); should ignore invalid characters in the input; should return NULL when it is given invalid binary data. |
| 2 | A date utility method that computes the number of days between two dates. |
| 3 | A HTML to text converter that receives the HTML code as a String object and returns the extracted text in another String object. |
| 4 | A credit card number validator that can handle major credit card types (e.g., Visa and Amex); determines if the given credit card number and type is a valid combination. |
| 5 | A bag data structure for storing items of type String; it should provide the five major operations: add, remove, size, count, and toString. |
| 6 | An XML comparison class that compares two XML strings and verifies if they are similar (contain the same elements and attributes) or identical (contain the same elements and attributes in the same order). |
| 7 | An IP address filter that verifies if an IP address is allowed by a set of inclusion and exclusion filters; subnet masks (like 127.0.1.0/24, 127.0.1/24, 172.16.25.* or 127.*.*.*) can be used to define ranges; it determines if an IP is allowed by the filters or not. |
| 8 | A SQL injection filter that identifies and removes possible malicious injections to simple SELECT statements; it returns the sanitized version of the supplied SQL statement; removes database comments (e.g., −-, #, and $\star$) and patterns like INSERT, DROP, and ALTER. |
| 9 | A text analyzer that generates a map of unique words in a piece of text along with their frequency of appearance; allows for case-sensitive processing; it returns a Map object that maps type String to Integer where the key is the unique word and the value is the frequency of the word appearance. |
| 10 | A command line parser with short (e.g., −v) and long (e.g. −-verbose) options support; it allows for options with values (e.g. −d 2, −-debug 2, −-debug=2); data types (e.g. Boolean, Integer, String, etc.) can be explicitly defined for options; it allows for locale-specific commands. |

come across tasks that would require that knowledge. In a similar fashion, we looked in source code example catalogues including java2s (http://www.java2s.com), which features thousands of code snippets demonstrating common programming tasks and usage of popular application programming interfaces (APIs). These two sources represent material designed for consumption by developers. To find what kinds of problems developers seek help to solve, we also looked at popular developer forums: Oracle Discussion Forums (https://forums.oracle.com/forums/category.jspa?categoryID=285) and JavaRanch (http://www.javaranch.com).

*Locating known solutions and their test cases.* After locating descriptions of pertinent tasks, we sought existing implementations relevant to those tasks on the internet through code search engines, discarding search results that did not also come with JUnit test

cases. After locating pertinent candidates we checked that both the solution and the test cases that exercised the solution existed in the repositories of the tools. Dissimilarity of the tools' underlying repositories made it difficult to select targets that simultaneously existed in all three repositories. Therefore, we settled for targets that exist in at least two of the three investigated repositories. Task selection and experimentation was performed incrementally over a period of three months; we found this process to be slow and laborious and thus we limited the investigated tasks to ten.

*Coverage of multiple units.* JUnit tests can cover more than one unit (despite the apparent connection between its name and "unit testing"). For example, an integration test can cover multiple classes collaborating in a single test case. Ideally, a test-driven reuse tool should be able to recommend suitable candidates for each missing piece referred to in a test scenario. Instead, current TDR prototypes have been designed around the idea that tests drive a single unit of functionality (i.e., a single method or class) at a time. We aimed our trial test cases to those targeting a single unit of functionality.[2]

### 3.2 Experimental Method

The experiment involved, for each identified task, (a) modifying the associated test case to anonymize the location of the known solution, (b) feeding the modified test case to the interface of each TDR tool, and (c) examining the resulting recommendations from each tool in order to assess their suitability to address the task (discussed in Section 3.3).

For simplicity of the study design, we assume that iterative assessment of recommendations and revision of the input test cases can be ignored. We further assume that a searcher would scan the ranked list of recommendations in order from the first to the last; however, we only consider the first 10 results, as there is evidence that developers do not look beyond this point in search results [7].

*Anonymization.* We wished to minimize experimenter bias by using (modified versions of) the existing test cases of the solutions. The query for each task then consisted of the modified JUnit test cases—thereby defining the desired interfaces, vocabulary, and testing scenarios of the trial tasks.

The test code used in the experiment was anonymized by a four step process: (1) any **package** statement is removed; (2) any **import** statements for types in the same project are removed (by commenting them out); (3) the set of test methods is minimized, by ensuring that all required methods for the sought functionality are exercised no less than once, while removing the rest; and (4) the statements within each test method are minimized, for cases where multiple conditions are tried, by retaining the first condition and removing the rest. This process was intended to reduce the test cases to the point that would resemble the minimal test scenarios developed in a TDD setting.[3]

*Tool-specific adjustments.* Minor adjustments were made to some test cases in the end to make them compatible with individual tools. For instance, Code Conjurer does not

[2] In one case, this constraint was not strictly met: the known solution for Task 4 relies on a set of **static** properties defined in a helper class CreditCardType.

[3] The complete set of known solutions and test cases, and their transformed versions used as inputs, can be retrieved from: http://tinyurl.com/icsr13.

fire a search when no object instantiation takes place in the test code, preventing Code Conjurer from triggering a search when the target feature is implemented through **static** methods. To get around this problem, we revised test cases for Tasks 1, 2, and 4 and replaced **static** method calls with instance method calls preceded by instantiation of the unit under test. The example in Figure 1 demonstrates some of the changes made to the query test class Base64UtilTest used in Task 1 for replacing **static** method calls with instance method calls.

```
@Test public void testEncodeString() {
  final String text = "This is a test";
// String base64 = Base64Util.encodeString(text);
// String restore = Base64Util.decodeString(base64);
  Base64Util util = new Base64Util();
  String base64 = util.encodeString(text);
  String restore = util.decodeString(base64);
  assertEquals(text, restore);
}
```

Fig. 1: A sample test query illustrating the replacement of **static** calls.

Unlike Code Conjurer and Code Genie, S6 comes with a web-based search interface; a class-level search through the Google or Sourcerer search provider was selected for all the searches performed through the S6 web interface. As S6 cannot process test code directly, a conversion process was followed to transform individual test cases into a form acceptable by the S6 interface (as a "call set"). Minor changes were made in some transformations due to the limitations imposed by the interface. In the case of Task 1, we could not provide the three test cases to S6 all at the same time; tests were therefore provided to the S6 search interface one at a time but S6 did not return results for any of them. For Task 9, we were not able to exactly reproduce the test case using an S6 call set. More specifically, we were not able to manipulate the returned java.util.Map object from the getWordFrequency() call; neither removing the assertions on the returned java.util.Map object nor using the "user code" feature produced any results. Task 10 involves the inner class CmdLineParser.Option that made S6 complain about an unknown type; we replaced the inner class with the type Object in order for the search to be launched.

### 3.3 Suitability Assessment

Many factors can make unplanned reuse difficult [2]. Two pieces of code of similar quality might satisfy the same feature set; however, developers are inclined to reuse the one that requires less adaptation and accommodation—after all, major motivations for reuse are to save time and effort and to reduce the likelihood of bugs. Therefore, it is important to present the developer with choices that they are likely to consider as relevant to their task, suitable for integration in their code, and whose adaptation would result in a net cost savings. To assess the quality of retrieved results, we thus recorded two subjective, qualitative measurements: relevance and effort.

*Relevance and effort.* Relevance is a measure traditionally used in evaluation of information retrieval (IR) systems, to indicate how well a retrieved resource meets the information needs of the user. In the context of this study, relevance measures how many of the features expected by a trial task are covered by a search result. For TDR, good relevance is necessary but not sufficient.

Effort is a measure of the work involved to adapt the retrieved source code. Effort is a compound measurement of size and complexity of the source code to be integrated, external objects it refers to, and the amount of mismatch it has with the existing development context; for our study, the "existing development context" comprised the test suite used as input to the search.

Two recommendations with the same relevance level may have completely different effort levels. For example, one recommendation can be considerably larger, more complex, or have more external dependencies than an equally relevant one. However, relevance and effort are not orthogonal. As the relevance starts to decline, the effort tends to increase; for example, additional effort might be required to add missing features. Ultimately the developer will avoid reusing located source code if the adaptation effort is (apparently) comparable to that of reimplementation.

To account for partial suitability, we adopted 5-point scales of relevance and of effort, as shown in Tables 2 and 3 respectively. For rating relevance, 1 stands for no/min-

Table 2: Guidelines for classifying relevance.

| Level | Description |
|---|---|
| 1 | There is no meaningful connection between the given task and the recommended code. I would not reuse this code to finish this task. |
| 2–4 | There is a noticeable overlap between the given task and the recommended code. However, some of the required features of the described functionality are missing or implemented in a different way (the smaller the mismatch, the higher the relevance). |
| 5 | The recommended code exactly or closely matches the functionality described in the task. |

Table 3: Guidelines for classifying effort.

| Level | Description |
|---|---|
| 1 | This code can be reused to develop the entire functionality described in the task. I may only need to do one or two very simple adjustments. |
| 2–4 | I may or may not choose to reuse the recommended code or its design ideas. However, to reuse it I would have to refactor it, make modifications to its design, or write new code for missing features (the fewer the adjustments, the lower the effort). |
| 5 | I would not reuse this code to develop the functionality described in the task. It would require too much effort to build upon this code. |

imal relevance and 5 stands for complete relevance. For rating efforts, 1 stands for little/no effort while 5 stands for excessive effort.

*Assessments of suitability.* After running the 10 trial tasks against the three test-driven reuse tools we collected 109 individual recommendation results (out of 300 possible results), as the number of results was less than our cut-off of 10 for some task/tool combinations.

There are 25 potential combinations of relevance and effort scores; however, only some of those combinations are expected to be observed, due to the relationship between relevance and effort discussed above. Table 4 indicates our classification of each possible combination, as good (a solution), ok (a near-solution), bad (a non-viable recommendation), or impossible (left as blank). We deem combinations of fairly low relevance and low effort to be contradictory and hence impossible, since lack of relevance implies high effort would be needed. In the absence of a good solution, a developer might consider a near-solution, which is a relevant result that still requires non-trivial effort to use for the task.

Relevance and effort ratings were assigned to the results following a manual inspection of the retrieved source code. An attempt was made to integrate the query test cases with the retrieved code, if possible. To make the tests run, external dependencies of the source code were resolved and refactorings were performed, if necessary. Ratings were given based on the effort spent to make the tests run and an estimate of the extent of the missing features in each case.

*External validation.* As relevance and effort are subjective measures, different developers can disagree on the reuse suitability of a piece of code in a certain context. To improve construct validity of the experiment, we compared our relevance and effort ratings with those from five experienced Java developers. Participants consisted of two graduate students and three industrial developers, all with 3–5 years of industrial experience in developing Java software. All participants reported that being familiar with the JUnit framework, having developed unit tests, and having conducted code reviews in the past. A short training example was used at the start of the session to familiarize participants with the procedures. A random sample of results (12 out of 109 recommendations) were selected, and provided to each participant for evaluation; each participant was asked to evaluate the same sample. In a short post-study questionnaire, all participants indicated that they have developed code for tasks similar to the ones they were given in the experiment, confirming that these are realistic tasks.

Table 4: Quality classes.

| | | Relevance | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 |
| Effort | 1 | | | | | good |
| | 2 | | | | ok | good |
| | 3 | | | bad | ok | ok |
| | 4 | | bad | bad | bad | bad |
| | 5 | bad | bad | bad | bad | bad |

Table 5: Inter-rater reliability scores.

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| Relevance | 0.86 | 0.89 | 0.93 | 0.84 | 0.81 |
| Effort | 0.72 | 0.75 | 0.82 | 0.74 | 0.72 |

Participants were given a guide that described the purpose of this experiment and the rationale behind the relevance and effort scores, along with—for the results in the sample to be validated—a short description of each task, the test cases, and the source code retrieved by a tool. Participants were asked to rate each result's relevance and effort according to our 5-point scales. Participants were asked to justify their choice through additional comments, which we used to check that their reasoning conformed to their numerical ratings. Spearman's rank correlation coefficient ($\rho$) was used to measure the inter-rater reliability of the rankings made by the first author and each participant; Spearman's $\rho$ can measure pairwise correlation among raters who use a scale that is ordered. Table 5 displays the $\rho$ values computed for participants P1 to P5. In all five cases, there is strong positive correlation between relevance and effort ratings of the first author and those of the external validators.

## 4 Results and Discussion

Table 6 summarizes the results of the experiment for each tool/task pairing, indicating: the number of recommendations returned; how many of these were duplicates; the ranking by the respective tool of the recommendation that we deemed the best, within the first 10 results (or fewer if fewer were recommended); and the quality of the best solution. To be clear, in some cases, the recommendation by a tool that we deemed best amongst its results, we still assessed as badly suited; the tools' rankings and our assessment of quality often did not correlate.

We can see that each of the tools did a poor job at recommending solutions. Code Conjurer provided a good solution for only one task, and near-solutions for two others;

Table 6: Results of the experiment for each tool/task combination. The columns for each tool indicate: the number of recommendations produced (rec); the number of these that are duplicates of other recommendations (dup); the ranking by the tool of the recommendation that we deemed the best within the results (best); and the quality of that best recommendation (qual). In cases marked with an asterisk, we were not able to verify the presence or absence of the known solution within the tool's repository.

| Task | Code Conjurer | | | | CodeGenie | | | | S6 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | rec (#) | dup (#) | best (rank) | qual | rec (#) | dup (#) | best (rank) | qual | rec (#) | dup (#) | best (rank) | qual |
| 1 | 8 | (1) | 8 | ok | *8 | (3) | 3 | good | 0 | | | |
| 2 | 9 | (1) | 1 | bad | *10 | | 2 | ok | 10 | (6) | 1 | good |
| 3 | 10 | (1) | 3 | ok | *0 | | | | 0 | | | |
| 4 | 0 | | | | *0 | | | | 10 | (6) | 1 | bad |
| 5 | 2 | (1) | 1 | bad | *0 | | | | 0 | | | |
| 6 | 10 | (2) | 1 | bad | 2 | | 1 | bad | 0 | | | |
| 7 | 10 | (5) | 2 | bad | 0 | | | | 0 | | | |
| 8 | 0 | | | | *0 | | | | 0 | | | |
| 9 | *10 | (4) | 1 | bad | 0 | | | | 0 | | | |
| 10 | 10 | (3) | 3 | good | 0 | | | | 0 | | | |

Table 7: Summary of quality classifications for all recommendations. The number of duplicate recommendations included is shown in parentheses.

| Quality | Code Conjurer | CodeGenie | S6 |
|---|---|---|---|
| good | 4 (3) | 3 (2) | 10 (6) |
| ok | 3 | 11 (2) | — |
| bad | 62 (15) | 6 | 10 (6) |
| Total | 69 (18) | 20 (3) | 20 (12) |

for five of the remaining tasks only bad recommendations were provided. CodeGenie provided a good solution for only one task, and a near-solution for one other; no recommendations were provided for seven out of eight of the remaining tasks, so false positives were relatively low. S6 only provided a good solution for one task, and no near-solutions; again, no recommendations were provided for seven out of eight of the remaining tasks, so false positives were relatively low. For Task 8, none of the tools provided a recommendation. For Tasks 4–7 & 9, each tool either provided no recommendations or only bad recommendations. In fairness, for task/tool combinations in which we were unable to verify the presence of the known solution (marked with asterisks), the associated repository may not have contained a viable alternative but this only affects four of the tasks for CodeGenie and none for the other two tools.

Table 7 summarizes our classifications of all recommendations produced by the tools for the 10 tasks. Code Conjurer has a much larger number of false positive (i.e., bad) recommendations than the other two, but all three tools produce many bad recommendations, when they produce any recommendations at all.

The results of our experiment indicate that there is a serious problem at work with the existing TDR approaches. Could the problem simply be due to implementation weaknesses, or is there a more fundamental shortcoming with the underlying ideas? To address this question, we first examined similarities between the input test cases and the results, described below.

### 4.1 Lexical and syntactic matching

From the published literature on the TDR approaches, we recognized the importance that each places on lexical and (to a lesser extent) syntactic similarity between potential hits in the repository and the input test case. Specifically, Code Conjurer and CodeGenie both utilize similarity of type and method names plus similarity of method signatures; S6 utilizes similarity between user-supplied keywords and potential hits plus similarity of method signatures. We manually examined each recommendation returned by the tools to determine if lexical or syntactic similarities existed with the input test case for each task. We empirically discerned four kinds of matching criteria: type name, method name, signature, and other keywords.

Table 8 presents the results of our similarity examination. We can see that Code Conjurer places great emphasis on type name similarity while S6 ignores it. But ultimately, every recommendation could be traced to a mostly lexical similarity.

Table 8: Classification of matches between recommendations and input test cases.

| Match kind | Code Conjurer | | CodeGenie | | S6 | |
|---|---|---|---|---|---|---|
| | High similarity | Partial similarity | High similarity | Partial similarity | High similarity | Partial similarity |
| Type name | 62 | | 15 | | | |
| Method name | 5 | 8 | 1 | 8 | | 6 |
| Signature | 15 | 8 | 10 | 8 | 20 | |
| Other keyword | 21 | | 2 | | 20 | |

This heavy reliance on lexical/syntactic similarity to the supplied test case, in making recommendations, yields many false positive results—especially when simple functionality is sought. For example, the utility program sought in Task 2 consists of a single function with two parameters of type java.util.Date and a return value of type **int**. Code Conjurer retrieved 9 results all of which match this signature but none of which match the desired functionality.

Each approach had the greatest success when multiple kinds of similarity occurred simultaneously; again, this is not surprising since the likelihood that similarities in multiple dimensions are spurious seems much lower than in few dimensions. Unfortunately, it appears from the results that simply demanding multiple kinds of lexical/syntactic similarity simultaneously would lead to limited applicability of these tools. Others have noted the tradeoff limitations to lexical/syntactic similarity in code search [3].

### 4.2 Issues with the Approaches

We see several issues that arise not from weaknesses in tool implementation, but more fundamentally from the ideas behind the TDR approaches.

*Signature matching.* The existing test-driven reuse approaches make signature matching a necessary condition to the relevance and matching criteria: a component is considered only if it offers operations with sufficiently similar signatures to the test conditions specified in the original test case. However, semantic similarity neither implies nor is implied by structural similarity. This limits the applicability of the test-driven reuse to situations in which the design of the feature sought is very simple or known in advance.

A more flexible approach to signature matching could improve recall. For example, operation argument and return value types, order, or count could be ignored. Unfortunately, this would in turn make the execution of test cases for validating candidate results difficult. Automated tests can only be run if a match can be established between missing elements in the tests and those in the retrieved source code. Code Conjurer retrieved testable results (source code that has sufficiently close signatures for at least some of the operations) for Tasks 1, 3, and 10, while CodeGenie could achieve the same goal only for Tasks 1 and 2. S6 tries to take advantage of simple transformations to generate possible candidates that can pass the tests; however, a candidate is considered for applying the transformations only if structural dissimilarities are minor. Consequently, S6 ended up retrieving results for only two of the tasks (Tasks 2 and 4) because candidates retrieved for other tasks did not meet the input criteria of the transformations.

*Filtering by lexical relevance.* Filtering candidate results based on their lexical relevance before other relevance criteria are considered leaves out all potential solutions that do not match the searcher's choice of program vocabulary. For example, the date utility function in Task 2 is named getNumberOfDaysBetweenTwoDates() and is defined in a class named DateUtils. Tokenizing these two names, as is performed by Code Conjurer and CodeGenie, would give a list of generic words that can be matched with almost any date utility class. All the 9 classes retrieved by Code Conjurer are named DateUtils and each has at least one method matching the signature of the method sought after; however, none of them is a method that computes the distance between two dates. CodeGenie manages to find an instance of the function for Task 2 that is given the same name and is defined in a class with the same name. Other results are false positives arising from lexical and signature similarities.

Code Conjurer and CodeGenie use terms in the signature of methods as a key component of relevance. While S6 uses a supplied keyword list to shrink the candidate space in which signature matching and transformations are to be performed, the use of keywords in the search has been limited to lexical matching that in turn results in tool performance being limited by the searcher's choice of vocabulary [13, 8]. We designed our experiment to favour the evaluated tools: we used test code taken from the same project in order to retrieve the feature under test. Changing the original program vocabulary instead—which would be reasonable in modelling situations where the developer does not know the needed vocabulary—would have limited lexical relevance, resulting in even worse performance of the tools.

*Automatically compiling and running source code.* The existing test-driven approaches all attempt to execute the supplied test case on potential matches in the repository. This has the advantage that additional semantic constraints implied by the supplied test case can be checked, eliminating false positive matches. Given the large number of false positives that we obtained from the approaches, it is clear that this idea is not working as intended; in fact, for Code Conjurer in particular, we believe that test case execution remains an unimplemented idea, judging from the very large number of false positives.

The retrieved source code has to be runnable in order to execute test cases, but automatically compiling and running arbitrary source code accumulated in a repository is no easy task, often because of dependencies on external source code. A number of heuristics have been proposed by the research community to resolve external dependencies without developer intervention [11]. In the context of our trial, only Task 4 has an external dependency on the org.apache.commons.lang project, while Tasks 6, 7, and 8 have dependencies to other source code in the same project, and the rest of the tasks can be compiled using a standard JDK by itself.

Even if source code can be compiled, there is still no guarantee that it can also be run, as programs can rely on specific runtime environments or resources. For example a web or mobile component relies on a specific container to run. A database or network application requires those external resources to offer its services. To the benefit of the evaluated tools, none of our tasks required an external environment or resources to run.

In a similar fashion, the TDR query test cases may also rely on resources external to the JUnit program. For example, the original version of the XML utility for Task 6 relied on XML strings loaded from the file system. For the sake of our experiment, we

```
import static org.junit.Assert.*;
import org.junit.Test;
import org.xml.sax.SAXException;

public class TestXmlDiff {
    String xml1 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><a><b>
        text1</b><c>text2</c></a>";
    String xml2 = "<?xml version=\"1.0\" encoding=\"ISO-8859-1\"?><!-- copy
        --><a><c>text2</c><b>text1</b></a>";

    /* @Test public void testDiff() throws XmlException, IOException, SAXException {
        DomainsDocument dd1 = DomainsDocument.Factory.parse(new File("src/test/
            resources/instances/test1.xml"));
        DomainsDocument dd2 = DomainsDocument.Factory.parse(new File("src/test/
            resources/instances/test2.xml"));
        Diff myDiff = new Diff(dd1.toString(), dd2.toString());
    } */

    @Test public void testDiff() throws IOException, SAXException {
        Diff myDiff = new Diff(xml1, xml2);
        assertTrue(myDiff.similar());
        assertFalse(myDiff.identical());
    }
}
```

Fig. 2: Test case for validating equality and similarity of XML strings (Task 6).

modified the test[4] (Figure 2) to utilize XML strings embedded in the source code, but there is no a priori reason to expect that the developer will not wish to rely on external resources in this fashion.

*Contextual facts.* Test cases are in fact simple examples demonstrating the use of the system-under-test. They show helper types that may interact with the system-under-test in typical scenarios. The existing TDR approaches disregard the elements of this interaction like participating types and the data/control flow between them. They merely extract lexical and syntactic features of missing elements from the tests while the context in which those elements appear might also help to understand their semantics.

For example, by solely relying on names and signatures, Code Conjurer did not retrieve anything related to XML processing for Task 6. Only one of the results out of the 10 retrieved happened to contain the keyword "xml" that was a statement importing the class org.allcolor.xml.parser.CStringTokenizer. The exception type org.xml.sax.SAXException thrown by the test method testDiff is part of a well-known API for processing XML documents. SAXException is not thrown by any of the resolvable methods in the test scenario; therefore, the functionality being sought should throw that excep-

[4] We repeated this experiment with the original version of the test case. Code Conjurer and CodeGenie produced the same result. S6 does not allow using external resources.

tion. Incorporating this additional semantic fact could have helped to improve the relevance of retrieved results.

*Searching for a specific implementation.* Code Conjurer and CodeGenie managed to retrieve relevant results for Task 1 in which a Base64 encoder/decoder is sought. The class name, base64 is the name of a well-known algorithm. The method names encode and decode are common choices for a utility class offering such services. However, the Base64 encoder/decoder described in the tests extends the common variation of this algorithm and adds a few constraints. When decoding Base64 character sequences, it should detect and ignore invalid sequences and simply return NULL. Therefore, our Task 1 is slightly different from most of the Base64 encoder/decoders available on the internet. None of the recommendations by Code Conjurer and CodeGenie offers the special behaviour expected. S6 fails to retrieve any results for the very same task. We speculate that it has retrieved various implementations of the Base64 algorithm through its initial keyword search, but not exactly the variation described in the tests; as none of them could pass the tests, they were all discarded in the end.

Most of the tasks reported by proponents of test-driven reuse approaches seek common variations of well-known algorithms and data structures. Using lexical and signature relevance criteria would yield multiple instances of such programs that can possibly pass the tests. However, similar to the example given above, if a variant were sought, relying on common terms and operation signatures would not suffice.

### 4.3 Threats to Validity

The primary question regarding generalizability of our study is the representativeness of the tasks. We took task ideas from the discussions in the Java developer community websites, and from code example catalogues commonly used as a reference by Java developers. Developers evidently find these features worthwhile to discuss and learn from, and not so easy to develop or to find. In addition, our five external evaluators consisting of three industrial developers and two graduate students found the tasks familiar in the sense that they had previously developed similar functionality. The number of trial tasks is another limiting factor of our study. However, it is comparable to the average number of tasks used in the evaluations in the TDR literature [5, 8, 13].

The modifications we made to the trial test cases in our study also threatens the validity of our study. Anonymization was performed to ensure that the facts in the test cases, other than the identity of the target project, could still contribute to identification of the solution. Test case refinement was done to remove test cases that exercised features beyond the scope of the study tasks. Neither anonymization nor refinement should negatively affect the retrieval capacity of the tools. To find the best strategy to overcome the tools' limitations, we experimented with different alternatives and compared results in each case. The alternative that yielded better results was chosen over others.

Test-driven reuse is a repetitive process. Searchers might reformulate their queries based on the current results in a way that may result in finding better results. This brings up the question of whether having a static query set is the right way to evaluate a code search tool. We deliberately ignored this issue by giving the tools the best possible queries by providing them with the test cases developed for the same code. We consid-

ered different existing variations of the features, and chose the ones that came with a reasonable test suite. This should have biased the results in favour of the tools.

Code relevance and effort categorization are subjective, and thus may differ from one developer to the next. It is often easy to say that one source code recommendation is more suitable than another, but the quantification of this difference is somewhat arbitrary. While our categorization of the relevance and effort of each recommendation represents our best judgment, a random subset of our categorizations was independently evaluated by experienced Java developers. The strong positive correlation between raters suggests our categorization of the results is reasonable.

Considering only the top 10 results for evaluating a retrieval algorithm might be overly restrictive, despite evidence that developers generally do not investigate more than the first 10 results [7]. However, in our experiment, the tools provided fewer than 10 recommendations in 22 out of the 30 cases. Therefore, we have considered all the results collected by the tools in more than 70% of the cases.

### 4.4 Precision versus accuracy

The measures we used in our evaluation are qualitative and imprecise. Nevertheless, the results suffice to demonstrate that the approaches work poorly for these examples, and point to the need to address their underlying designs. Thus the results do provide *accuracy*: our criteria for rating the results are sufficiently well defined that our participants' ratings agreed to a degree that is quantitatively demonstrable.

The greater precision that would be obtained by using quantitative measures is not warranted—measuring degrees of "poor performance" would not provide us with a deeper understanding of the cause of the failure of these approaches. Only with an acceptable level of performance is it worthwhile to invest in precise measurements.

## 5 Conclusion

As test-driven development has gained in industrial popularity, the prospect of utilizing test cases as the basis for software reuse has become tantalizing: test cases can express a rich variety of structure and semantics that automated reuse tools can potentially utilize. However, the practice of test-driven development implies that the test cases that are written cannot be too heavily depended on as the absolute truth regarding the functionality that is sought.

We have performed an experiment on the three state-of-the-art tools for test-driven reuse, in which we found realistic, non-trivial tasks in developer forums, and for which a known solution existed in the tools' repositories. We used existing test cases that exercised the known solution as the basis of the input to the tools. All the tools failed in most cases to locate relevant source code that would be simple to reuse, and often recommended irrelevant source code.

One may posit that it is unrealistic to expect any TDR approach to *not* depend on the presence of specific names. If one works in a context where domain vocabulary is well-established (within a specific organization, or while utilizing a specific application programming interface [3]), this dependency could even be reasonable. However, we have illustrated that these approaches still do not suffice to find uncommon variations of functionality in the presence of a common vocabulary. Furthermore, we believe

that demanding this limitation is defeatist: the desire to find useful functionality in the absence of known vocabulary is industrially reasonable and should not be dismissed.

We remain convinced of the value of the idea of TDR. To overcome the problems we have identified, alternative approaches need to be more flexible in recommending solutions, recognizing the inability of the developer to know exact vocabulary and that such vocabulary will often fail to suffice in locating a desired variation on common functionality. Other aspects of the rich information available in test cases could be leveraged to reduce the dependency on specific names, which would allow TDR to become a more general purpose and hence more generally useful approach. We are currently investigating alternative solutions to achieve this.

## Acknowledgments

We thank Steven Reiss, Oliver Hummel, Werner Janjic, and Sushil Barjracharya for assisting us with their tools, and Brad Cossette, Rylan Cottrell, Soha Makady, and Valeh Hosseinzadeh Nasser for helpful suggestions in editing this paper. This work was supported by scholarships and grants from NSERC and IBM.

## References

[1] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition, 2004.

[2] R. Holmes and R. Walker. Systematizing pragmatic software reuse. *ACM Trans Softw Eng Methodol* **21**(4):20/1–20/44, 2012.

[3] R. Holmes et al. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans Softw Eng* **32**(12):952–970, 2006.

[4] O. Hummel and C. Atkinson. Supporting agile reuse through extreme harvesting. In *Proc Int Conf Extreme Progr*, LNCS 4536, 28–37, 2007.

[5] O. Hummel et al. Code Conjurer: Pulling reusable software out of thin air. *IEEE Softw* **25**(5):45–52, 2008.

[6] J.-J. Jeng and B. Cheng. Specification matching for software reuse: A foundation. In *Proc ACM Symp Softw Reusabil*, 97–105, 1995.

[7] T. Joachims et al. Accurately interpreting clickthrough data as implicit feedback. In *Proc ACM SIGIR Int Conf Info Retrieval*, 154–161, 2005.

[8] O. Lemos et al. A test-driven approach to code search and its application to the reuse of auxiliary functionality. *Info Softw Technol* **53**(4):294–306, 2011.

[9] E. Linstead et al. Sourcerer: Mining and searching internet-scale software repositories. *Data Min Knowl Discov* **18**(2):300–336, 2009.

[10] A. Zaremski and J. Wing. Signature matching: A key to reuse. In *Proc ACM Int Symp Foundations Softw Eng*, 182–190, 1993.

[11] J. Ossher et al. Automated dependency resolution for open source software. In *Proc Working Conf Min Softw Repos*, 130–140, 2010.

[12] A. Podgurski and L. Pierce. Behavior sampling: A technique for automated retrieval of reusable components. In *Proc Int Conf Softw Eng*, 349–361, 1992.

[13] S. Reiss. Semantics-based code search. In *Proc Int Conf Softw Eng*, 243–253, 2009.