Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability

András Kicsi
Department of Software Engineering
University of Szeged
Szeged, Hungary
akicsi@inf.u-szeged.hu

László Tóth
Department of Software Engineering
University of Szeged
Szeged, Hungary
premissa@inf.u-szeged.hu

László Vidács
MTA-SZTE Research Group on Artificial Intelligence
University of Szeged
Szeged, Hungary
lac@inf.u-szeged.hu

ABSTRACT

Striving for reliability of software systems often results in immense numbers of tests. Due to the lack of a generally used annotation, finding the parts of code these tests were meant to assess can be a demanding task. This is a valid problem of software engineering called test-to-code traceability. Recent research on the subject has attempted to cope with this problem applying various approaches and their combinations, achieving profound results. These approaches have involved the use of naming conventions during development processes and also have utilized various information retrieval (IR) methods often referred to as conceptual information. In this work we investigate the benefits of textual information located in software code and its value for aiding traceability. We evaluated the capabilities of the natural language processing technique called Latent Semantic Indexing (LSI) in the view of the results of the naming conventions technique on five real, medium sized software systems. Although LSI is already used for this purpose, we extend the viewpoint of one-to-one traceability approach to the more versatile view of LSI as a recommendation system. We found that considering the top 5 elements in the ranked list increases the results by 30% on average and makes LSI a viable alternative in projects where naming conventions are not followed systematically.

CCS CONCEPTS

• **Computing methodologies** → **Natural language processing**; • **Software and its engineering** → **Traceability**;

KEYWORDS

testing, traceability, natural language processing (nlp), LSI

ACM Reference Format:
András Kicsi, László Tóth, and László Vidács. 2018. Exploring the Benefits of Utilizing Conceptual Information in Test-to-Code Traceability. In *RAISE'18 : IEEE/ACM 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering, May 27, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/3194104.3194106

RAISE'18 , May 27, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5723-4/18/05…$15.00
https://doi.org/10.1145/3194104.3194106

1 INTRODUCTION

The strive for higher software quality produces an increasingly large amount of test code. Nowadays larger systems include tens of thousands of test cases. This vast amount of test cases can be genuinely hard to manage properly. In these cases identifying which test case is meant to test which parts of the system becomes a major issue. This is a well known problem of software engineering called test-to-code traceability [21, 29].

Having a large number of tests, it is crucial to be able to find out in a relatively simple way which parts of the production code are tested by a single test case. If we encounter a failed test case, the code under test usually needs to be modified. Test-to-code traceability is still an open problem in software engineering, but several results have been achieved in the recent years. The most simple solution would be to regulate the process of testing so that traceability information could be close at hand at any time. Regrettably, these regulations are rather hard to maintain and without the proper awareness of their importance are seldom used. This would also be tedious to implement on already existing software systems with no previous traceability information.

There are several techniques however for extracting test-to-code traceability information from an existing codebase. Rompaey and Demeyer [29] inspect some of these techniques and provide a comparison between them. These techniques included naming conventions and information retrieval (IR) which are based on conceptual information. As reported by the paper, the naming convention technique achieved perfect precision on this evaluation, but its applicability proved rather low generally. Naming conventions are very precise and provide an easy way to extract traceability links, but they are in most cases loosely defined and hard to enforce. They mainly depend on the conscience and the discipline of the developers. Even assuming the best possible attitude, naming conventions are still unable to cope with every situation.

The information retrieval approach relies mainly on textual information extracted from the source code of the system. Variables and comments usually contain meaningful text aimed to be understood by human readers. Uncovering the conceptual context of the test cases can be used for extracting traceability information. The state of the art techniques in test-to-code traceability use a combination of different approaches, including structural dependencies and conceptual information, which latter relies on the Latent Semantic Indexing (LSI) technique [23].

In this paper we extend the viewpoint of one-to-one traceability approach of naming conventions to the more flexible view of LSI as

a traceability recommendation system [27, 28]. This means that instead of the most similar class we consider the 5 most similar classes as subjects to a given test case. This introduces fault tolerance and greater versatility when traceability links are not obvious at a cost of a small amount of additional human intervention. Although LSI is not the leading standalone technique, we argue that it provides a viable alternative in projects where testing naming conventions are not followed. To support this view, we experiment with naming conventions and textual similarities provided by the LSI technique as traceability methods.

We provide the following contributions by applying natural language processing to test-to-code traceability:

- We introduce flexibility to test-to-code traceability by applying the LSI technique as a recommendation system. Considering top 5 classes instead of only 1 increases recall rate by 30% on average.
- We show that a customized LSI recommender system approximates the naming convention technique by 97% on average when top 5 recommended classes are considered.
- Manual inspection shows that LSI can produce meaningful results even when naming conventions are not followed. In some cases the tested class is ranked within the top 5 results by the LSI.

2 GOALS AND METHOD

2.1 Test-to-Code Traceability Approach

Our current goals were to recover test-to-code traceability information relying on conceptual information extracted through natural language processing. To achieve this, we used the LSI technique widely used throughout software engineering. LSI has been successfully applied in various traceability problems in the recent years [2]. Test-to-code traceability differs from these tasks in many aspects [21]. One of the biggest differences is that there is no completely natural language based text to rely on as a textbase. This means that we have to work with only identifier names and comments found throughout the code. This complicates the work, opposed to different traceability problems, where the algorithm can rely on larger textual information like requirements or bug reports. Information retrieval methods also depend on the habits of the developers, proper commenting and descriptive naming factors greatly. The LSI technique builds a corpus from a set of documents and computes conceptual similarity of these documents with each query presented to it. In our current experiments the production code classes of a system were considered the documents forming the corpus, while the test cases were used as queries. The textual information was prepared with the habitual preprocessing methods. Figure 1 provides a high level glance at our process (see Section 2.3 for details).

```
public void testInitMakesManagerAvailableInFacade() {
  try {
    ProfileFacade.getManager();
    fail("ProfileFacade shouldn't be initialized!");
  } catch (RuntimeException e) { }
  initSubsystem.init();
  assertNotNull(ProfileFacade.getManager());
}
```

Listing 1: A simple example of a JUnit test case

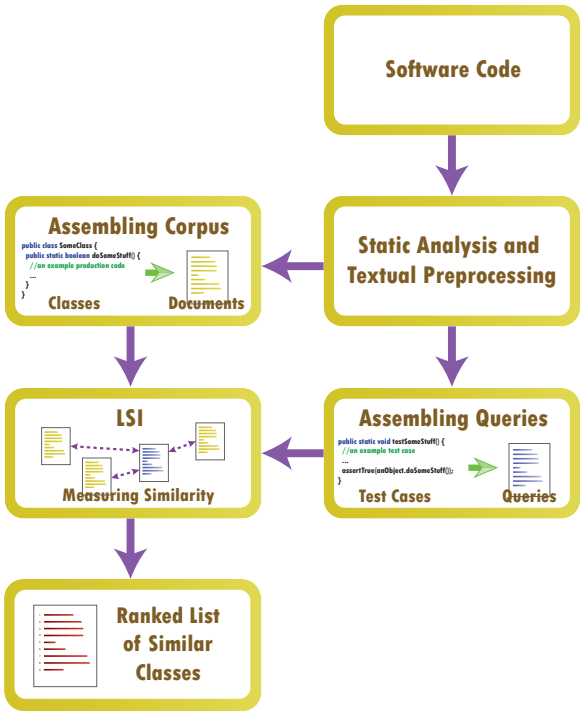

Figure 1: An illustration of our process

Let us consider a real test case taken from ArgoUML, featured in Listing 1. This test case ensures that *ProfileFacade*'s manager is not initialized before we call *initSubsystem*'s *init* method, and that it is properly initialized afterwards. It is not visible from the example, but *initSubsystem* is an instance of *InitProfileSubsystem*. The test case is located in the *org.argouml.profile.TestSubsystemInit* class.

Recovering test-to-code traceability links in a recommendation system manner holds a number of benefits. Though unit tests usually aim to test one part of the code, this is often less clear in practice. In our current example the success of the test depends both on *InitProfileSubsystem*'s *init* method and also *ProfileFacade*'s state. While naming conventions could only approach this as a simple one-to-one relationship, this is not necessarily the best way. A recommendation system providing for instance five possible matches for the test case could highlight this relationship more thoroughly. As a matter of fact, the top two classes found most similar to the test case by LSI are exactly *ProfileFacade* and *InitProfileSubsystem*.

Of course, recommending a number of matches for each test case results in having to filter out the possible bad matches. It is a small amount of manual work, but it is still vastly less effort than searching the whole projects for the tested code would mean. With only one-to-one matching used by naming conventions, we would not get any clues on what other parts of the code might be influencing the failure of the test case, while with similarity, this information is readily available. In case of a faulty match, this also results in a number of other possibilities to choose from. Thus, naming conventions, though highly useful, can still have drawbacks even if they have been properly used. It is also important to note that naming conventions in many cases are not really defined formally

and can vary by each system, or even within one system itself, which means that uncovering traceability links may need to be tuned to a specific system before functioning properly.

As the naming conventions technique utilizes textual matching, its results could theoretically also be reproduced by textual similarity. These would provide less certain matches, but leave room for the small violations of the naming conventions often present. We can also see this in the example, where *InitProfileSubsystem* and *TestSubsystemInit* would match only by a very loose definition of naming conventions, while their similarity is still very high. Thus, relying on similarity of the names could result in a more versatile approach than simply defining rules.

In determining the recommendation factor, we rely on empirical results borrowed from fault localization research. In a fault localization scenario, developers examine methods based on suspiciousness rank to find the cause of test failures. User experiments report, that developers tend to examine only the first 5, or at least the first 10 elements in the ranked list [11, 34]. Other places are neglected by developers regardless of their content. Thus, we experiment with 1 to 10 long recommendation lists, where 1 long list means the one-to-one matching of LSI. In our experiments we put emphasis on 5 long lists, since this was supported by most developers, and an acceptable degree of freedom can be achieved by the LSI algorithm.

Considering the arguments presented in this section, we set up three research questions we aim to answer in this paper:

RQ1: How does the IR method applied as a recommendation system perform compared to the naming conventions method?

RQ2: Can we customize the IR method to achieve similar results as the naming conventions technique?

RQ3: Are there useful results produced by LSI when naming conventions are barely followed (manual investigation)?

2.2 Evaluation Procedure

According to previous research [29], proper naming conventions can produce 100% precision in finding the tested class where the conventions were systematically followed. As the systems under test contain naming conventions to at least some extent, we based our evaluation method on the test cases properly covered by naming conventions. To produce a sufficiently precise set of correct test-code pairs, we made an algorithm with rather simple, yet sufficiently strict rules. We require the test class to have the same name as the production code class it tests, having the word 'Test' before or after the name. Pairs should also have the same package hierarchy, starting from the test package in case of the test classes, meaning that their qualified names are also the same. If these rules apply to two classes, we deemed it sufficient to be covered by naming conventions, and be used for evaluation purposes.

In order to quantify our results, we introduce the recall rate metric, which is frequently used in case of recommendation systems [33, 36]. The number of results provided is represented by k. Since the recommendations are aimed for human users, we do not evaluate outside this k recommendation factor. In our experiments we mainly considered the top 5 results, because this is the quantity the developers tend to still accept, as stated in the previous subsection.

$$recall\text{-}rate@k = \frac{N_{detected}}{N_{total}}$$

In the current case $N_{detected}$ signifies the total number of correctly detected units under test, while N_{total} represents the total number of units under test we are looking for. Because of the limitations of our chosen evaluation technique based on naming conventions we are only capable of considering one-to-one traceability links. This means that N_{total} is always the same as the number of test cases in a system. Thus the results of the recall rate and an average of the frequently used recall measure will always coincide in our case.

2.3 Applying LSI on Real Projects

The experiments featured in this paper were done on systems written in the Java programming language. Since we work with an IR-based technique, using mainly the natural language part of the code, the programming language of the source code should be of no real significance, however we cannot guarantee this. Our experiments feature the extraction of program code from the systems under test using static analysis, distinguishing tests from production code, textual preprocessing and determining the conceptual connections between tests and production code using Latent Semantic Indexing. During the experiments we used the Gensim [26] toolkit's implementation of LSI. The initial static analysis that provides the text of each method and class of a system in a structured manner is performed with the Source Meter [32] static source code analysis tool.

Table 1: Size and versions of the programs used, Methods = the number of all production and test methods, NC = the percentage of test cases which follow naming conventions

Program	Version	Classes	Methods	Tests	NC
Comm. Lang	3.4	596	6 523	2 473	87.38%
Comm. Math	3.4.1	2 033	14 837	3 493	77.61%
ArgoUML	0.35.1	2 404	17 948	554	75.63%
Mondrian	3.0.4	1 626	12 186	1 546	19.73%
JFreeChart	1.0.19	953	11 594	2 239	37.42%

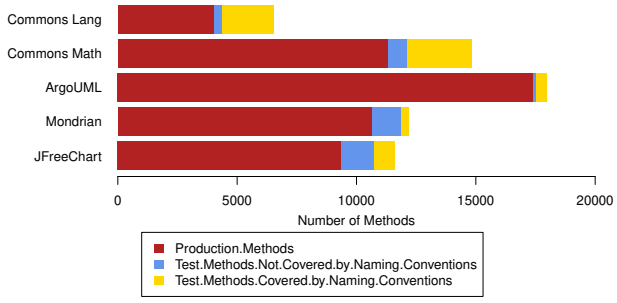

Figure 2: Properties of the sample projects used

The preprocessing phase involves the commonly used preprocessing methods of NLP. It features splitting [6][4] up camel case names, which can transform variable names to a more useful, meaningful form, lower casing, bringing the terms to a more similar and

more easily manageable form with stemming [7], filtering out some, mainly Java-specific stopwords, and weighting some other terms positively, like the terms in the names of methods.

LSI's corpus can basically be considered as a multidimensional semantic field in which the conceptually more similar documents are located closer to each other. If a query is inserted into this field, it also signifies a point in the field, which results in a measurable distance, thus similarity to the documents of the corpus. This means that for every document-query pair we can easily get the magnitude of semantic similarity. If we order these results by similarity then we can recover any desired number of most similar documents for each query, or using a well chosen similarity threshold value, we can create a versatile algorithm, producing only the pairs with significant similarity.

We evaluated our technique on the following programs listed in Table 1. Commons Lang is a module of the Apache Commons project. It aims to broaden the functionality provided by Java regarding the manipulation of Java classes. Commons Math is also a module of Apache Commons, aiming to provide mathematical and statistical functions missing from the Java language. ArgoUML is a tool for creating and editing UML diagrams, offering a graphic interface and relatively easy usage. The Mondrian Online Analytical Processing (OLAP) server improves the handling of SQL databases of large applications. JFreeChart enables Java programs to display various diagrams, supporting several diagram types and output formats.

The evaluated versions of programs, their total number of classes and methods and the quantity of their test cases are shown in Table 1 with the NC column featuring how many of the test cases followed the naming conventions based on qualified names. Figure 2 reflects these numbers in a visual manner.

3 RESULTS AND DISCUSSION

3.1 LSI as a standalone technique

In this section we evaluate LSI as a recommendation system for traceability, compared to the naming convention technique. Results were evaluated on five open source programs and are shown in Table 2. We experimented with ranked lists of 1, 2, 5 and 10 most similar classes according to LSI. We used method bodies, Javadoc comments and qualified names in the corpus. In order to achieve an accurate evaluation, we compare LSI results to traceability links identified by naming conventions, which means that the numbers on the table represent the results achieved with only these test cases. We have to remark that though most cases this means a large part of the system under test, there are some systems, especially Mondrian with only 19.73% of its test cases covered by naming conventions, that provide less data of the whole system. Furthermore, considering the limitations of naming conventions, it is not possible to assess the possibility of more than one-to-one real traceability links.

Our results show that LSI as a standalone technique can usually successfully produce 30-60% of the valid traceability links also recovered by naming conventions. From the results it is also visible that the high or low use of naming conventions does not necessarily mean that the LSI result will be the same, its success does not seem to depend on naming conventions.

Table 2: Results featuring the corpus built from code, Javadoc comments and qualified names of the methods

Program	LSI	LSI_{top2}	LSI_{top5}	LSI_{top10}
Commons Lang	**61.7%**	73.6%	**88.5%**	96.3%
Commons Math	**29.7%**	42.3%	**56.9%**	67.0%
ArgoUML	**37.2%**	49.9%	**60.9%**	68.1%
Mondrian	**45.2%**	58.4%	**73.1%**	80.3%
JFreeChart	**33.5%**	45.7%	**63.0%**	75.2%

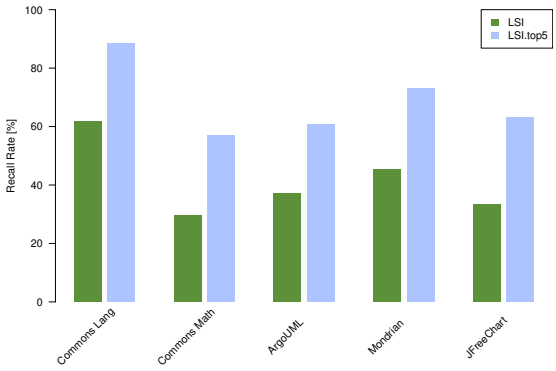

Figure 3: Our method to class traceability results

Figure 4 presents the values of Table 2 in a more easily comparable manner, while Figure 3 highlights the improvement achieved by using the top 5 most similar classes rather than only one. As these figures also testify, providing a number of results can increase the quality of the output significantly, achieving an improvement of on average 30% at the top 5 scenario, raising the number of correctly found links to a number between 57 and 88 percent. While LSI as a standalone technique may perform worse than relying on naming conventions where these are followed systematically, a recommendation system based on LSI can successfully identify a considerable amount of the correct traceability links. It is also apparent that there are great differences between projects. LSI is not bound by differences and provides results for each case. For example in case of Mondrian less than a fifth of its code follows naming conventions and the top1 45% and top5 73% LSI results are still relatively high. Judging by the numbers, we believe that in case of a weak naming convention result, LSI especially as a recommendation system, can bring significant benefits and projected to the whole system it may even outperform naming conventions.

Answer to RQ1: Considering LSI as a recommendation system (top5 case) increases its overall benefits by **30%** on average. Where naming conventions are not often followed (Mondrian and JFreeChart projects), **top1** LSI results may be comparable to naming conventions, but the **top5** recommendation version may outperform naming conventions projected to the whole project and can be a valuable alternative.

LSI also produces results where naming conventions were not applied, while a technique relying solely on naming conventions could produce none. The number of correct results in these cases

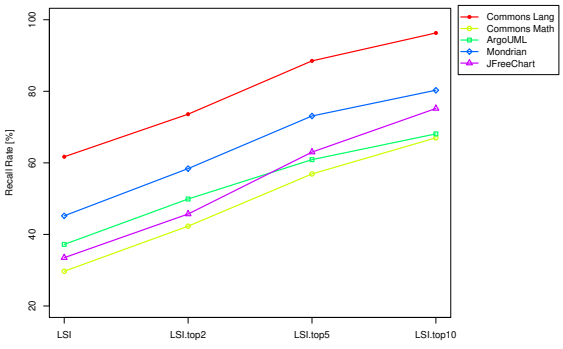

Figure 4: An overview of our results on different programs

unfortunately cannot be measured with our current technique, thus these are not reflected in our numbers. Not following naming conventions may worsen LSI results too, but it may still provide a considerable amount of correct traceability links. This is an important point and we are taking a glance at this scenario in a later subsection by manual evaluation.

3.2 Approximating the Naming Conventions Technique

In the second experiment the LSI corpus is built only from the qualified names of the methods. This approach is in some respects close to the naming conventions technique, which uses only names to detect connections. The results of this approach are shown on Table 3. As it can bee seen from the table, LSI can successfully approximate the results of naming conventions. Since our evaluation method is based on the detected naming conventions, the precision of the naming conventions method is considered 100% in each presented case. This does not mean that our technique performs necessarily poorer on the whole system than the one using naming conventions only, nor does it mean that this name-based LSI performs better than the one using code also. These results are only given on the code covered by naming conventions, saying nothing of the remaining classes, so these numbers merely show that the results of the method using naming conventions can be successfully approximated by a name-based LSI method. While the naming conventions can identify only one-to-one connections, and only in the cases they where correctly used, the LSI technique can be effective in theses cases too, given the appropriate settings. Thus, LSI can produce similarly correct results, while maintaining a limited amount of versatility at the same time.

Table 3: Results featuring the corpus built from qualified names of the methods

Program	*LSI*	LSI_{top2}	LSI_{top5}	LSI_{top10}
Commons Lang	**55.3%**	82.2%	**96.4%**	100.0%
Commons Math	**73.9%**	89.6%	**97.2%**	99.3%
ArgoUML	**92.8%**	98.8%	**99.8%**	100.0%
Mondrian	**75.1%**	95.7%	**99.7%**	99.7%
JFreeChart	**63.4%**	77.7%	**91.7%**	97.8%

Answer to RQ2: A customized LSI recommender system approximates the naming convention technique to **97%** on average when the **top 5** recommended classes are considered.

3.3 Manual Evaluation

As already established, the biggest disadvantage of the naming conventions technique is that conventions are not followed in every single case and the rate of their usage differs largely depending on which system we are talking about. We see IR-based techniques as a possible circumvention of this obstacle, since the usage of natural language during development is much less optional than conventions. Due to the lack of the true traceability links, the real performance of LSI on the parts of systems not following naming conventions is unknown. That is why we deemed it necessary to do a manual evaluation of our results too, thus reducing their dependence on the naming conventions technique. For this purpose we investigated several different, randomly chosen cases for which the naming conventions were not applicable to. Our findings show that LSI can indeed produce veritable traceability links even in these circumstances. We have already seen a simple example on this in Section 2, but we would like to present some further points on this matter.

Let us look at a real example shown on Listing 2. It is a test case of the *org.jfree.chart.StackedBarChartTest* class of JFreeChart. This is one case chosen randomly of the several test cases located in the same package not covered by naming conventions.

```
/**
 * Replaces the dataset and checks that it has changed
     as expected.
 */
@Test
public void testReplaceDataset() {

  // create a dataset...
  Number[][] data = new Integer[][]
    {{new Integer(-30), new Integer(-20)},
     {new Integer(-10), new Integer(10)},
     {new Integer(20), new Integer(30)}};

  CategoryDataset newData =
       DatasetUtilities.createCategoryDataset(
       "S","C",data);

  LocalListener l = new LocalListener();
  this.chart.addChangeListener(l);
  CategoryPlot plot = (CategoryPlot)
       this.chart.getPlot();
  plot.setDataset(newData);
  assertEquals(true, l.flag);
  ValueAxis axis = plot.getRangeAxis();
  Range range = axis.getRange();
  assertTrue("Expecting the lower bound of the range to
       be around -30: " + range.getLowerBound(),
       range.getLowerBound() <= -30);
  assertTrue("Expecting the upper bound of the range to
       be around 30: " + range.getUpperBound(),
       range.getUpperBound() >= 30);

}
```

Listing 2: A test method from JFreeChart

As it can be observed after manual inspection, the focal method should be the *setDataset(int index, CategoryDataset dataset)* method presented in Listing 3, located in the *org.jfree.chart.plot.CategoryPlot* class. It is called by the *setDataset(CategoryDataset dataset)* method,

with an index value of 0, which is the sole purpose of the method. This setter changes the old dataset to a new one obtained through parameter by changing the dataset's *ChangeListener* and triggering a new change event. The test case basically tests this process. It firstly tests whether the change event occurred through checking whether the *LocalListener* defined in the same class for the purpose of this test has changed its flag to true, which signifies that a *ChartChangeEvent* occurred. The test case also tests whether the range of the plot with new data is of appropriate lower and upper bound, using previously defined values.

```
/**
 * Sets a dataset for the plot and sends a change
     notification to all
 * registered listeners.
 *
 * @param index  the dataset index (must be &gt;= 0).
 * @param dataset  the dataset ({@code null} permitted).
 *
 * @see #getDataset(int)
 */
public void setDataset(int index, CategoryDataset
    dataset) {
  CategoryDataset existing = (CategoryDataset)
      this.datasets.get(index);
  if (existing != null) {
    existing.removeChangeListener(this);
  }
  this.datasets.put(index, dataset);
  if (dataset != null) {
    dataset.addChangeListener(this);
  }
  // send a dataset change event to self...
  DatasetChangeEvent event = new
      DatasetChangeEvent(this, dataset);
  datasetChanged(event);
}
```

Listing 3: The code Listing 2 was meant to test

As it is evident from the names, this traceability connection is not recoverable through naming conventions, since all class and method names differ greatly. Our LSI-based method however marks *CategoryPlot* as the most similar production code class, leading us correctly to the relevant part of the code. It is also worth mentioning that although *setDataset* is the method the test was meant to evaluate, the success of the test also depends on the correctness of several other methods of JFreeChart, hence a failure of the test does not necessarily stem from the fault of *setDataset*. The test case also depends on *ChartChangeEvent*, *CategoryPlot*'s *getRangeAxis* method, and even on *DatasetUtilities* for creating new dataset value and *ValueAxis* for the range it is meant to test. In a fault localization scenario these can mean crucial information, which IR-based methods can also highlight.

Generally speaking of JFreeChart, it can be observed that the two main packages of production code, *chart* and *data* show significant difference in the appliance of conventions. In the *data* package naming conventions are used with great thoroughness, the name of the test class always includes the name of the class tested. In the *chart* package on the other hand, there are many cases that do not reflect any other class in name, only the functionality they are meant to test, like generating a specific type of plot. Testing higher level functions can involve the use of several different parts of the production code, and this may present significant barriers to naming conventions, though the code may still provide clues to the appropriate traceability links. Considering the above circumstances, we believe that leaning more on textual similarity can produce good results even when conventions can not. A more thorough evaluation of this case is needed in future research.

Answer to RQ3: First manual evaluation shows that LSI can produce meaningful results even when naming conventions are neglected. In some cases the tested class is ranked within the top 5 results by LSI.

4 RELATED WORK

Conceptual analysis was successfully applied in various software engineering topics in recent years [19]. LSI [3] is often used throughout software engineering, for example in fault localization [18], in detection of bug report duplicates [12], test-prioritization [30], feature analysis [9, 10] and in the field of traceability, for example between tests and requirements [15]. Several efforts have been made to improve the application of the LSI technique itself, for example Query-based reconfiguration approach [17] and using part of speech information [1]. Summarization techniques aim to generate descriptive names or human-oriented summaries of program elements in general [5, 16], and for tests as well [8, 13, 20, 35].

Test-to-code traceability is an intensively studied topic [14], however no known perfect solution exists to the problem. Several individual solutions were proposed, like plugins integrated in development environments [22], and also methods relying on static or dynamic analysis [31], highlighting their usage in this current problem. Call graphs, information in names and timestamps were utilized in the process of finding traceability links. Rompaey et al. [29] use three open source programs to evaluate the effectiveness of techniques that rely on naming conventions, fixture elements, the program's call graph, the last call before each assert command (LCBA), their code as a text, and data provided by a version control system. For their information retrieval based technique the authors chose LSI, which proved outstanding in its applicability, but performed the worst at its precision. Qusef et al. [25] improved the LCBA technique with data flow analysis, relying highly on data dependencies. In a follow-up work [23] dynamic slicing is used to increase the number of identified connections, and precision is maintained using conceptual coupling, incorporating the LSI technique. The authors named their method SCOTCH and have proposed several improvements to it [24].

In this paper we provided a deeper analysis of the LSI technique and compared it to the naming convention-based method. Although LSI is not appropriate as a standalone technique, we carried out using extensive measurements and pointed out its advantages, especially when naming conventions are not properly followed. The current state of the art solutions also use conceptual information. We introduced a recommendation system approach to its use and presented measurable benefits, with the tradeoff of having top 5 results.

5 CONCLUSION

In this paper we experimented with how natural language processing can aid test-to-code traceability. We used the Latent Semantic Indexing technique to identify the tested class for each test case of a program. We analyzed the LSI method in the light of the naming

convention-based approach and performed experiments on five open source systems. Our contribution lies in the idea to apply LSI as a recommendation system, thus to consider not only the topmost element of its ranked list. This allows much greater versatility and fault tolerance in traceability. The LSI is always applicable, and can also detect one-to-many links, which is a deficiency of the naming conventions technique.

We found that considering the top 5 elements in the ranked list increases the results by 30% on average and makes LSI as viable alternative in projects where naming conventions are not strictly followed. With special corpus selection we approximated the naming convention technique to 97% in the top 5 scenario, however in this case the generality of the LSI is questionable. We also manually looked at cases where naming conventions are not followed and found examples for LSI being capable. In the future we plan to extend the manual analysis and collect data for a combined traceability method incorporating LSI.

REFERENCES

[1] Reem S. Alsuhaibani, Christian D. Newman, Michael L. Collard, and Jonathan I. Maletic. 2015. Heuristic-based part-of-speech tagging of source code identifiers and comments. In *2015 IEEE 5th Workshop on Mining Unstructured Data (MUD)*. IEEE, 1–6.

[2] Markus Borg, Per Runeson, and Anders Ardö. 2014. Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability. *Empirical Software Engineering* 19, 6 (dec 2014), 1565–1616.

[3] SC Deerwester, ST Dumais, and TK Landauer. 1990. Indexing by latent semantic analysis. *Journal of the American Society for Information Science and Technology* 41, 6 (1990), 391–407.

[4] Bogdan Dit, Latifa Guerrouj, Denys Poshyvanyk, and Giuliano Antoniol. 2011. Can Better Identifier Splitting Techniques Help Feature Location?. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on (ICPC '11)*. IEEE, Washington, DC, USA, 11–20.

[5] Brian P. Eddy, Jeffrey A. Robinson, Nicholas A. Kraft, and Jeffrey C. Carver. 2013. Evaluating source code summarization techniques: Replication and expansion. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 13–22.

[6] Emily Hill, David Binkley, Dawn Lawrie, Lori Pollock, and K. Vijay-Shanker. 2014. An empirical study of identifier splitting techniques. *Empirical Software Engineering* 19, 6 (dec 2014), 1754–1780.

[7] Emily Hill, Shivani Rao, and Avinash Kak. 2012. On the Use of Stemming for Concern Location and Bug Localization in Java. In *Working Conference on Source Code Analysis and Manipulation*. IEEE, 184–193.

[8] Manabu Kamimura and Gail C. Murphy. 2013. Towards generating human-oriented summaries of unit test cases. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 215–218.

[9] András Kicsi, László Vidács, Arpád Beszédes, Ferenc Kocsis, and István Kovács. 2017. Information Retrieval Based Feature Analysis for Product Line Adoption in 4GL Systems. In *Proceedins of the 17th International Conference on Computational Science and Its Applications – ICCSA 2017*. IEEE, 1–6.

[10] András Kicsi, László Vidács, Viktor Csuvik, Ferenc Horváth, Arpád Beszédes, and Ferenc Kocsis. 2018. Supporting Product Line Adoption by Combining Syntactic and Textual Feature Extraction. In *International Conference on Software Reuse, ICSR 2018*. Springer International Publishing.

[11] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. ACM Press, New York, New York, USA, 165–176.

[12] Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Improving the accuracy of duplicate bug report detection using textual similarity measures. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, New York, New York, USA, 308–311.

[13] Boyang Li, Christopher Vendome, Mario Linares-Vasquez, Denys Poshyvanyk, and Nicholas A. Kraft. 2016. Automatically Documenting Unit Test Cases. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Coll. of William & Mary, Williamsburg, VA, USA, IEEE, 341–352.

[14] Patrick Mader and Alexander Egyed. 2012. Assessing the effect of requirements traceability for software maintenance. *IEEE International Conference on Software Maintenance, ICSM* (2012), 171–180.

[15] Andrian Marcus and Jonathan I. Maletic. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. *25th International Conference on Software Engineering, 2003* (2003), 125–135.

[16] Paul W. McBurney. 2015. Automatic Documentation Generation via Source Code Summarization. In *Proceedings - International Conference on Software Engineering (ICSE 2015)*, Vol. 2. 903–906.

[17] Laura Moreno, Gabriele Bavota, Sonia Haiduc, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, and Andrian Marcus. 2015. Query-based configuration of text retrieval solutions for software engineering tasks. In *ESEC/FSE 2015*. ACM Press, 567–578.

[18] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. 2014. On the use of stack traces to improve text retrieval-based bug localization. In *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*. IEEE, 151–160.

[19] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. 2013. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 199–208.

[20] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. 2016. The impact of test case summaries on bug fixing performance. In *38th International Conference on Software Engineering - ICSE '16*. ACM Press, New York, New York, USA, 547–558.

[21] Reza Meimandi Parizi, Sai Peck Lee, and Mohammad Dabbagh. 2014. Achievements and Challenges in State-of-the-Art Software Traceability Between Test and Code Artifacts. *IEEE Transactions on Reliability* 63 (2014), 913–926.

[22] Friedrich Steimann Philipp Bouillon, Jens Krinke, Nils Meyer. 2007. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. In *Agile Processes in Software Engineering and Extreme Programming*. Vol. 4536. Springer Berlin Heidelberg, 101–104.

[23] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2011. SCOTCH: Test-to-code traceability using slicing and conceptual coupling. In *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 63–72.

[24] Abdallah Qusef, Gabriele Bavota, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2014. Recovering test-to-code traceability using slicing and textual analysis. *Journal of Systems and Software* 88 (2014), 147–168.

[25] Abdallah Qusef, Rocco Oliveto, and Andrea De Lucia. 2010. Recovering traceability links between unit tests and classes under test: An improved method. In *IEEE International Conference on Software Maintenance, ICSM*. IEEE, 1–10.

[26] Radim Rehurek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks* (2010), 45–50.

[27] Martin Robillard, Robert Walker, and Thomas Zimmermann. 2010. Recommendation Systems for Software Engineering. *IEEE Software* 27, 4 (jul 2010), 80–86.

[28] Martin P. Robillard, Walid Maalej, Robert J. Walker, and Thomas Zimmermann. 2014. *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated.

[29] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. In *European Conference on Software Maintenance and Reengineering, CSMR*. IEEE, 209–218.

[30] Ripon K. Saha, Lingming Zhang, Sarfraz Khurshid, and Dewayne E. Perry. 2015. An Information Retrieval Approach for Regression Test Prioritization Based on Program Changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 268–279.

[31] H.M. Sneed. 2004. Reverse engineering of test cases for selective regression testing. In *European Conference on Software Maintenance and Reengineering, CSMR 2004*. IEEE, 69–74.

[32] SourceMeter 2018. SourceMeter Webpage. https://www.sourcemeter.com/. (2018).

[33] Chengnian Sun, David Lo, Siau Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings* (2011), 253–262.

[34] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 267–278.

[35] Benwen Zhang, Emily Hill, and James Clause. 2016. Towards automatically generating descriptive names for unit tests. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. ACM Press, New York, New York, USA, 625–636.

[36] Yun Zhang, David Lo, Xin Xia, and Jian Ling Sun. 2015. Multi-Factor Duplicate Question Detection in Stack Overflow. *Journal of Computer Science and Technology* 30, 5 (2015), 981–997.