Automatically Documenting Software Artifacts

Boyang Li
Advisor: Denys Poshyvanyk
Department of Computer Science, College of William Mary
Williamsburg, VA 23185
{boyang,denys}@cs.wm.edu

***Abstract*—Software artifacts constantly change during evolution and maintenance of software systems. One critical artifact that developers need to be able to maintain during evolution and maintenance of software systems is up-to-date and complete documentation. However, recent studies on the co-evolution of comments and code showed that the comments are rarely maintained or updated when the respective source code is changed.**

In order to understand developer practices regarding documenting two kinds of software artifacts, unit test cases and database-related operations, we designed two empirical studies both composed of (i) an online survey with contributors of open source projects and (ii) a mining-based analysis of method comments in these projects. Later, motivated by the findings of the studies, we proposed two novel approaches. `UnitTestScribe` is an approach for automatically documenting test cases, while `DBScribe` is an approach for automatically documenting test cases. We evaluated our tools by means of an online survey with industrial developers and graduate students. In general, participants indicated that descriptions generated by our tools are complete, concise, and easy to read.

I. Introduction

Software artifacts, like unit test cases and database schema, constantly change during evolution and maintenance of software systems. For example, the number of unit test cases often grows as new functionality is introduced into the system. Maintaining these unit tests is important to reduce the introduction of regression bugs due to outdated unit tests (i.e., unit test cases that were not updated simultaneously with the update of the particular functionality that it intends to test). For instance, Test Driven Development (TDD) [4] has been employed by a myriad of developers and organizations to create and expand software systems [5, 6]. TDD requires unit test cases to be written prior to development after which developers write code to build the particular functionality that is required to pass those existing test cases.

On the other hand, previous work extensively studied the co-evolution of source code and DB schemas demonstrating that: (i) schemas evolve frequently, (ii) the co-evolution oftentimes happens asynchronously (i.e., code and schema evolve collaterally) [9, 36], and (iii) schema changes have significant impact on DCAs' code [9]. Therefore, co-evolution of code and DB schemas in DCAs often leads to two types of challenging scenarios for developers, where (i) changes to the DB schema need to be incorporated in the the source code, and (ii) maintenance of a DCA's code requires understanding of how the features are implemented by relying on DB operations and corresponding schema constraints. Both scenarios demand detailed and up-to-date knowledge of the DB schema.

Therefore, one critical artifact that developers need to be able to maintain during evolution and maintenance of software systems is up-to-date and complete documentation. Source code comments are another source of documentation that could help developers understand nuances of the test cases and database usages. However, recent studies on the co-evolution of comments and code showed that the comments are rarely maintained or updated when the respective source code is changed [14, 15]. In order to support developers in maintaining unit test cases and database schema usage, we propose novel approaches,`UnitTestScribe` and `DBScribe`. We evaluated our tools by means of an online survey with industrial developers and graduate students. In general, participants indicated that descriptions generated by our tools are complete, concise, and easy to read.

In summary, the paper makes the following contributions:

- Two empirical studies to understand whether developers comment and update comments of unit test cases and database related methods;
- Two surveies of both open-source and industrial developers to understand their perspective and practices with respect to documenting unit test cases and database related methods;
- An approach for automatically documenting test cases;
- An approach for automatically documenting database schema usage;

II. Related works

A. Studies on automatically summarizing software artifacts

There are several related techniques for automatically summarizing and documenting different software artifacts. Sridhara et al. [41] presented an approach for automatically generating summary comments for Java methods. They demonstrated how to identify important lines of code, depending on various characteristics of methods, and convert them into NL phrases. McBurney and McMillan [31] presented a novel approach for method summarization by considering contextual information. As far as class level granularity is concerned, Moreno et al. [32, 33] focused on documenting content and responsibilities of the Java classes. Their descriptions are based on superclass, stereotypes of the class, and behavior of the blocks. Automatic summarization techniques have also been applied to exceptions [7], bug reports [37], loops [42], changes summary [8, 10, 27], and code examples [44].

B. Approaches and studies on unit test cases

Kamimura and Murphy [21] presented an approach for automatically summarizing JUnit test cases. The approach identified the focal method based on how many times the test method invokes the function. The least occurring invocations are the most unique function calls for the test case. Xuan and Monperrus [43] split existing test cases into multiple fractions for improving fault localization. Their test case slicing approach has also influence on code readability. Recently, Pham et al. [35] presented an approach for automatically recommending test code examples when programmers make changes

in the code. Panichella et al. [34] presented an approach for automatically generating test case summaries for JUnit test cases. Runeson [39] conducted a survey to understand how unit testing is perceived in companies. Some researchers focused on other aspects of testing, which include unit test case minimization [22, 23], prioritization [12, 38], automatic test case generation [11, 16], test templates [45], data generation [25, 29]. However, none of the existing approaches focuses on generating unit test case documentation as NL summaries.

C. Approaches and studies on DCAs

Recent studies showed a presence of strong evolutionary coupling between database schemas and source code [18, 30, 36, 40]. Maule *et al.* used program slicing and dataflow-based analysis to identify the impact of database schema changes [30]. Qiu *et al.* conducted an empirical study into co-evolution between DB schemas and source code demonstrating that DB schemas frequently evolve with many different types of changes at play [36]. Sjøberg mainly focused on database schema changes and presented a technique for measuring the changes of database schemas; a study on health management systems over several years showed additions and deletions to be the most frequent operations [40]. Cleve *et al.* presented a method to analyze the change history of DBs [9]. In addition, a group of studies focuses on inferring database relations[3, 25].

III. State-Of-The-Practice

In order to understand developer practices regarding documenting two kinds of software artifacts, unit test cases and database-related operations, we designed two empirical studies [24, 28] both composed of (i) an online survey with contributors of open source projects at GitHub and (ii) a mining-based analysis of method comments in these projects.

The context of **study 1** was 1,414 open source C# projects hosted on GitHub and the complete revision history of 246 of these projects. The survey was completed by 212 developers that either contributed to these projects or worked in industry. The perspective is that of researchers interested in identifying developers practices for documenting unit tests.

As for the context of **study 2**, we analyzed 3,113 open source Java projects at GitHub (with JDBC API calls executing SQL queries/statements) and the complete change history of 264 of those projects; we also surveyed 147 developers contributing to these projects. We were interested in identifying developers practices for documenting database-related operations.

We investigated the following research questions (RQs) based on the two studies:

RQ$_1$ *To what extent do unit test cases (resp. database-related operations) contain comments?* This RQ aims to address the prevalence of both a preceding comment and inner comments for the unit test cases (resp. database-related operations).

RQ$_2$ *To what extent do developers update unit test case comments (resp. database-related comments)?* This RQ investigates how often developers modify and update the unit test case comments (resp. database-related comments) during software evolution.

RQ$_3$ *To what extent, do developers have difficulty understanding unit test cases (resp. database-related operations)?* This RQ investigates whether there are obstacles in understanding unit tests cases (resp. database-related operations) and the need by developers for support in this task.

A. Result summary for RQ1

Study 1: Although 47.17% of the developers indicated that they document unit test cases in comments, we observed that 96.44% of the projects lacked preceding comments and 85.98% lacked inner comments to document the unit tests. We also observed that 27.93% projects contained test cases despite 53.77% of developers indicating that they "fairly often" or "always" write test cases (83.01% if we consider the response "sometimes").

Study 2: While developers indicated that they documented methods, we found 77% of methods with database access were completely undocumented. In fact, 115 out of 147 (78.23%) surveyed developers consider that documentation of schema constraints should not be included in the source code and it is a responsibility of the schema or external documentation.

B. Result summary for RQ2

Study 1: Despite 44.81% of developers indicated that they "rarely" or "never" update unit test comments, we found that 1.54% of the preceding comments and 15.23% of the inner comments in 101 projects were changed at least once between releases when the unit test method was also modified.

Study 2: While approximately half of the developers indicated that they "rarely" or "never" update method comments for database-related methods, we empirically observed that only 17.15% of methods that were changed in 3,113 open source projects also had their comments updated at least once between releases. Thus, we empirically found database-related methods are far less frequently commented during evolution.

C. Result summary for RQ3

Study 1: More than half of the developers indicated a difficulty of "moderate" to "very hard" in terms of understanding unit tests. Emphasizing this importance, we observed that 89.15% of developers "agree" or strongly agree" that maintaining test cases impacts the quality of the system. This suggests that developers could benefit from tools that support them in maintaining unit test cases during software evolution and maintenance.

Study 2: Surveyed developers prefer to rely on external database documentation and two-thirds of developers indicated tracing constraints along the call-chain was a "moderate" challenge or a "very hard" challenge. This opens the discussion about whether external database documentation is enough for supporting source code evolution and maintenance tasks.

D. Discussion

These studies suggest that (i) documenting unit test cases and database usages is not a common practice in source code methods, (ii) developers do not update comments when changes are done to unit test cases or database-related methods, and (iii) understanding unit test cases and database-related usage (e.g., tracing schema constraints through call-chains in the call graph) are not an easy task in most of the cases. While results for **RQ**$_1$ and **RQ**$_2$ describe developers rationale for not documenting some software artifacts, the findings in **RQ**$_3$ present a different perspective in terms of whether current practices for documenting test cases and databases are enough or useful for supporting developers.

IV. Proposed Approaches

Based on the findings from the study (Section III), it is clearly important to have approaches to support developers in maintaining different software artifacts (i.e., unit test cases and database-related methods). Therefore, we designed and implemented i) an approach, called `UnitTestScribe` [24], to support unit test cases documentation and ii) a novel approach, namely `DBScribe` [26], aimed at automatically generating always up-to-date natural language descriptions of database operations and schema constraints.

A. UnitTestScribe: Documenting unit test cases

`UnitTestScribe` is a novel approach that combines static analysis, natural language processing, backward slicing, and code summarization techniques in order to automatically generate expressive NL descriptions concisely documenting the purpose of unit test methods (i.e., methods in unit tests). The main conjecture of `UnitTestScribe`'s approach is that the purpose of a unit test method can be described by identifying (i) general descriptions of the test case method, (ii) focal methods, (iii) assertions in the test case method, and (iv) internal data dependencies for the variables in assertions. A focal method is a method from the system under test, which is invoked in a unit test case, and is responsible for system state changes that are examined through assertions in unit tests [17]. We recognized focal methods as an important piece of information to be included in the resulting summary.

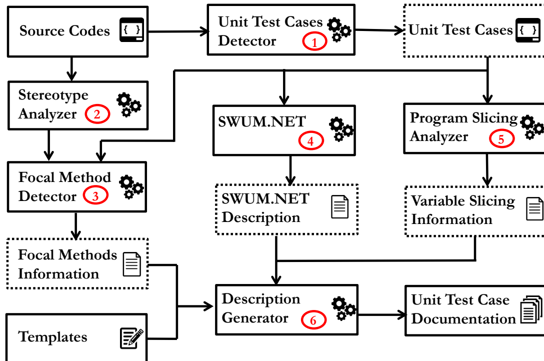

Fig. 1: `UnitTestScribe` components and workflow.

The architecture of `UnitTestScribe` is depicted in Fig. 1. The starting point of `UnitTestScribe` is the source code of the system, including source code of the unit tests. `UnitTestScribe` analyzes the source code to identify all the unit test cases ①. Then, `UnitTestScribe` performs data-flow analysis to identify stereotypes at method level [13] in the source code; the stereotypes detection is necessary to identify the focal methods in the unit test methods ②. After having identified all the test cases and stereotypes, `UnitTestScribe` detects focal methods for each unit test case ③. `UnitTestScribe` also uses `SWUM.NET` to generate a general NL description for each unit test case method. `SWUM.NET` [2, 19] captures both linguistic and structural information about a program, and then generates a sentence describing the purpose of a source code method ④. The data dependencies between focal methods, assertions, and variables in the test method are detected by performing static backward slicing [20] ⑤. Finally, the extracted information (focal methods, assertions, slices, and SWUM sentence) are structured in NL description by using predefined templates

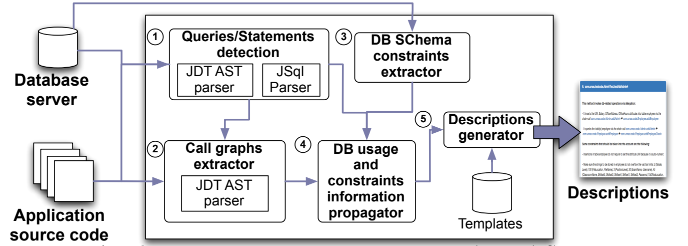

Fig. 2: `DBScribe` components and workflow.

⑥. The final descriptions for all the methods are organized in `UnitTestScribe` documentation in HTML format. In the following subsections, we describe the details behind each of the steps and components in `UnitTestScribe`.

B. DBScribe: Documenting database usages

`DBScribe` provides developers with updated documentation describing database-related operations and the schema constraints imposed on those operations. The documentation is contextualized for specific source code methods; in other words, the documentation is generated considering the local context of the methods and the operations delegated through inter-procedural calls and the subsequent call-chains that involve at least one SQL-statement. Therefore, our method-level documentation can provide developers with descriptions that work at different layers for a given DCA. Concerning the usefulness, we designed `DBScribe` to help developers when (i) understanding how features are implemented using SQL operations, and (ii) understanding schema constraints that need to be satisfied in both specific methods of the source code and all the operations involved. Also, `DBScribe` is suitable for on-demand execution by developers that require up-to-date documentation.

The architecture of `DBScribe` is depicted in Figure 2. `DBScribe`'s workflow is composed of five phases: ① SQL-statements and the methods executing them are detected in the source code statically; ② a partial call graph with the call-chains including the methods executing SQL-statements (locally and by delegation) are extracted from the source code statically; ③ database schema constraints are extracted by querying the master schema of the database engine that has an instance of the database supporting the DCA under analysis; ④ the constraints and SQL-statements are propagated through the partial call graph from the bottom of the paths to the root; and ⑤ the local and propagated constraints and SQL-statements (at method-level) are used to generate natural language based descriptions. Current `DBScribe`'s implementation covers SQL-statements invoked by means of JDBC and Hibernate API calls.

V. Empirical Study

A. Research Questions

We conducted two user studies in which the descriptions generated by the tools were evaluated by developers, computer science students, and researchers from different universities. The *goal* of the studies was to measure the quality of `UnitTestScribe` and `DBScribe` descriptions as perceived by users according to a well-established framework for evaluating automatically generated documentation [10, 32, 41]. The RQs aimed at evaluating the three quality attributes in the evaluation framework (i.e., $completeness$, $conciseness$, and $expressiveness$).

TABLE I: Study questions and answers.

RQ 4: Do you think the message is complete?	UnitTestScribe	DBScribe
• Does not miss any important info.	165(63.5%)	205(65.7%)
• Misses some important info.	78(30.0%)	91(29.2%)
• Misses most important info.	17(6.5%)	16(5.1%)
RQ 5: Do you think the message is concise?	UnitTestScribe	DBScribe
• Contains no redundant info.	136(52.3%)	221(70.8%)
• Contains some redundant info.	102(39.2%)	77(24.7%)
• Contains a lot of redundant info.	22(8.5%)	14(4.5%)
RQ 6: Do you think the description is expressive?	UnitTestScribe	DBScribe
• Is easy to read	157(60.4%)	241(77.3%)
• Is somewhat readable	69(26.5%)	60(19.2%)
• Is hard to read and understand	34(13.1%)	11(3.5%)
RQ 7: Are our generated description useful for understanding the artifacts?	UnitTestScribe	DBScribe
• Yes	21(80.8%)	48(92.3%)
• No	5(19.2%)	4(7.7%)

RQ$_4$ *How **complete** are the descriptions generated by the tools (`UnitTestScribe` and `DBScribe`)?*

RQ$_5$ *How **concise** are the descriptions generated by the tools (`UnitTestScribe` and `DBScribe`)?*

RQ$_6$ *How **expressive** are the descriptions generated by the tools (`UnitTestScribe` and `DBScribe`)?*

RQ$_7$ *How well can the tools help developers understand the related software maintenance tasks?*

B. Data Collection

To answer the RQs related to `UnitTestScribe`, we asked the participants to evaluate the descriptions generated by `UnitTestScribe` for four open source systems `SrcML.NET`, `Sando`, `Glimpse`, and `Google-api-dotnet`. For each group, we created an on-line survey using the Qualtrics tool [1]. The survey included (i) demographic background questions, and (ii) questions aimed at answering the RQs (See Table I). We also checked the free-text responses in depth to understand the rationale behind the choices.

We conducted another user study to evaluate the usefulness of `DBScribe`. As for the *context*, we used five open-source DCAs hosted at GitHub and SourceForge. We randomly selected six methods from each system (30 descriptions in total from five open-source DCAs); in particular, we selected two methods from the GUI layer that are at the root of method call-chains invoking SQL-statements, two methods that are leaves of the call-chains (i.e., declare SQL-statements, but do not delegate declaration/execution to other methods), and two methods in the middle of the call-chains. This selection was aimed at evaluating `DBScribe`'s descriptions at different layers of DCAs' architectures. Also, we limited the survey to six descriptions per system to make sure our survey could be completed in one hour to avoid an early survey drop-out.

Due to the space limitation, we did not show the detailed information about the subject applications and experimental designs. More information can be found in our previous publications and their online appendix [24, 26].

C. Results

The results of the studies are shown in Table I.

For completeness (**RQ 4**), 63.5% `UnitTestScribe` descriptions do not miss any important information, while only 6.5% of the answers indicate that the descriptions miss majority important information to understand the unit test case. We also observed that `UnitTestScribe` was evaluated more positively on complex methods rather than simple methods. In addition, the results show that 65.71% answers agreed that `DBScribe`'s descriptions do not miss any important information, while only 5.13% answers indicated the documents missed the most important information.

For conciseness (**RQ 5**), 52.3% of the answers indicate that `UnitTestScribe` descriptions contain no redundant/useless information, while only 8.5% of the answers indicate the description contain significant amount of redundant/useless information. Most of the responses with lower scores were from test case methods with the number of assertions greater than four. On the other hand, 70.83% of the answers asserted that `DBScribe`'s descriptions do not contain redundant information and only 4.49% answers indicated that the descriptions contain a lot of redundant information.

For expressiveness (**RQ 6**), 60.4% of the answers indicate that `UnitTestScribe` descriptions were easy to read and understand, while only 13.1% of the answers indicated the descriptions were hard to read and understand. The distribution of ratings with the lowest rank is similar to the conciseness question where descriptions for simple test case methods were evaluated more positively than the complex test case methods. Similar to conciseness, the reason is that `UnitTestScribe` are attempting to cover all important information for expressiveness. On the other hand, in 77.3% of the answers, `DBScribe`'s descriptions were evaluated as easy to read, while only 3.5% answers indicated that the descriptions were hard to read. The participants who thought some descriptions were hard to read claimed that the descriptions had a lot of information. Similar to Conciseness, our descriptions are attempting to capture more important information, which may come at the expense of expressiveness.

For preferences (**RQ 7**), we also asked whether the generated `UnitTestScribe` descriptions are useful for understanding the unit test cases. 21 out of 26 (80.8%) participants answered "Yes". For `DBScribe`, 48 participants (92.3%) claimed that the generated descriptions would be useful for understanding the database usages in source code methods.

VI. Conclusion

This work is motivated by studies in which we surveyed open source/professional developers to understand their perspective towards unit test cases and database operations. We found that developers believe that maintaining good documentations is important for the quality of a software system. We also mined changes of large amount of open source projects to show that the projects lack related comments in practice.

We presented a novel approach `UnitTestScribe` that combines static analysis, natural language processing, and code summarization techniques in order to automatically generate expressive NL descriptions concisely documenting the purpose of unit test methods. Future work will support other unit test frameworks and other programming languages.

In addition, we presented `DBScribe`, a novel approach for automatically generating natural language documentation at source code method level that describe database usages and constraints for a given DCA. `DBScribe` currently supports systems using the JDBC APIs; therefore, future work will support SQL-statements executed with other ORM frameworks and database engines. We will improve the SQL-statement detection by resolving SQL literals in source code that are declared with values returned by interprocedural calls or passed as arguments to methods.

References

[1] "qualtrics. http://www.qualtrics.com."

[2] "Swum.net. https://github.com/abb-iss/Swum.NET/."

[3] R. Alhajj, "Extracting the extended entity-relationship model from a legacy relational database," *Information Systems*, vol. 28, no. 6, pp. 597–618, 2003.

[4] K. Beck, *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

[5] ——, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[6] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, 1998.

[7] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *2008 ISSTA*.

[8] ——, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM ASE*, 2010, pp. 33–42.

[9] A. Cleve, M. Gobert, L. Meurice, J. Maes, and J. Weber, "Understanding database schema evolution: A case study," *Science of Computer Programming*, vol. 97.

[10] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *SCAM'14*. IEEE, 2014, pp. 275–284.

[11] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on FSE*, 2015.

[12] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *ICST'13*. IEEE, 2013, pp. 302–311.

[13] N. Dragan, M. L. Collard, J. Maletic *et al.*, "Reverse engineering method stereotypes," in *ICSM'06*.

[14] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *WCRE 2007.*, Oct, pp. 70–79.

[15] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, 2009.

[16] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *FSE'11*.

[17] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *SCAM'15*, p. 10 pages.

[18] M. Goeminne, A. Decan, and T. Mens, "Co-evolving code-related and database-related changes in a data-intensive software system," in *CSMR-WCRE 2014*.

[19] E. Hill, *Integrating natural language and program structure information to improve software search and exploration*. University of Delaware, 2010.

[20] R. Jhala and R. Majumdar, "Path slicing," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005.

[21] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *ICPC'13*. IEEE, 2013, pp. 215–218.

[22] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *Software Reliability Engineering, 2005.* IEEE, 2005.

[23] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *ASE 2007*.

[24] B. Li, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and N. Kraft, "Automatically documenting unit test cases," in *Proceedings of ICST'16*, Apr. 2016.

[25] B. Li, M. Grechanik, and D. Poshyvanyk, "Sanitizing and minimizing databases for software application test outsourcing," in *ICST 2014*. IEEE, 2014, pp. 233–242.

[26] M. Linares-Vasquez, B. Li, C. Vendome, and D. Poshyvanyk, "Documenting database usages and schema constraints in database-centric applications," in *ISSTA'16*.

[27] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "Changescribe: A tool for automatically generating commit messages," in *37th IEEE/ACM ICSE'15, Formal Research Tool Demonstration*, 2015.

[28] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *ASE 2015*. IEEE, pp. 36–41.

[29] R. Malhotra and M. Garg, "An adequacy based test data generation technique using genetic algorithms," *Journal of information processing systems*, vol. 7, no. 2, 2011.

[30] A. Maule, W. Emmerich, and D. S. Rosenblum, "Impact analysis of database schema changes," in *ICPC 2008*.

[31] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *ICPC 2014*. ACM, pp. 279–290.

[32] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *ICPC 2013*. IEEE, pp. 23–32.

[33] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *ICPC 2013*.

[34] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," in *Proceedings of the 38th ICSE*, 2016.

[35] R. Pham, Y. Stoliar, and K. Schneider, "Automatically recommending test code examples to inexperienced developers," in *FSE'15*. ACM, pp. 890–893.

[36] D. Qiu, B. Li, and Z. Su, "An empirical analysis of the co-evolution of schema and code in database applications," in *Proceedings of the FSE 2013*.

[37] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *Software Engineering, IEEE Transactions on*, vol. 40, no. 4, pp. 366–380, 2014.

[38] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, 2001.

[39] P. Runeson, "A survey of unit testing practices," *Software, IEEE*, vol. 23, no. 4, pp. 22–29, 2006.

[40] D. Sjøberg, "Quantifying schema evolution," *Information and Software Technology*, vol. 35, no. 1, pp. 35–44, 1993.

[41] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *ASE*, 2010.

[42] X. Wang, L. Pollock, and K. Vijay-Shanker, "Developing a model of loop actions by mining loop characteristics from a large code corpus," in *ICSME*. IEEE, Sep 2015.

[43] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *FSE 2014*. ACM.

[44] A. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," in *FSE*. ACM, 2014, pp. 460–471.

[45] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from test names (n)," in *ASE 2015*.