
Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

Summer 2017

Automatically Documenting Software Artifacts

Boyang Li

College of William and Mary - Arts & Sciences, bli01@email.wm.edu

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Li, Boyang, "Automatically Documenting Software Artifacts" (2017). *Dissertations, Theses, and Masters Projects*. William & Mary. Paper 1530192342.

<http://dx.doi.org/10.21220/s2-ss5q-gj87>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Automatically Documenting Software Artifacts

Boyang Li

Jinan, Shandong, China

Master of Science, Miami University, 2011
Bachelor of Science, Shandong Normal University, 2007

A Dissertation presented to the Graduate Faculty
of The College of William & Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

College of William & Mary
January 2018

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



Boyang Li

Approved by the Committee, September 2017



Committee Chair

Associate Professor Denys Poshyvanyk, Computer Science
College of William & Mary



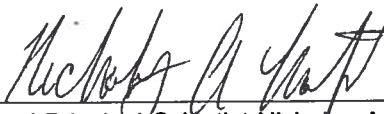
Professor Weizhen Mao, Computer Science
College of William & Mary



Assistant Professor Xu Liu, Computer Science
College of William & Mary



Professor Peter Kemper, Computer Science
College of William & Mary



Lead Principal Scientist Nicholas A. Kraft,
ABB Corporate Research

COMPLIANCE PAGE

Research approved by

Protection of Human Subject Committee

Protocol number(s): PHSC-2014-12-04-9995-dposhyvanyk

Date(s) of approval: 12/15/2014

ABSTRACT

Software artifacts, such as database schema and unit test cases, constantly change during evolution and maintenance of software systems. Co-evolution of code and DB schemas in Database-Centric Applications (DCAs) often leads to two types of challenging scenarios for developers, where (i) changes to the DB schema need to be incorporated in the source code, and (ii) maintenance of a DCAs code requires understanding of how the features are implemented by relying on DB operations and corresponding schema constraints. On the other hand, the number of unit test cases often grows as new functionality is introduced into the system, and maintaining these unit tests is important to reduce the introduction of regression bugs due to outdated unit tests. 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.

In order to understand developer practices regarding documenting and maintaining these software artifacts, we designed two empirical studies both composed of (i) an online survey of contributors of open source projects and (ii) a mining-based analysis of method comments in these projects. We observed that documenting methods with database accesses and unit test cases is not a common practice. Further, motivated by the findings of the studies, we proposed three novel approaches: (i) DBScribe is an approach for automatically documenting database usages and schema constraints, (ii) UnitTestScribe is an approach for automatically documenting test cases, and (iii) TeStereo tags stereotypes for unit tests and generates html reports to improve the comprehension and browsing of unit tests in a large test suite. We evaluated our tools in the case studies with industrial developers and graduate students. In general, developers indicated that descriptions generated by the tools are complete, concise, and easy to read. The reports are useful for source code comprehension tasks as well as other tasks, such as code smell detection and source code navigation.

TABLE OF CONTENTS

Acknowledgments	vi
Dedication	vii
List of Tables	viii
List of Figures	x
1 Introduction	2
1.1 Contribution	2
1.2 Dissertation Overview	3
2 Documenting Database-Centric Applications	7
2.1 An Empirical Study on Documenting DCAs	10
2.1.1 Research Questions	10
2.1.2 Data Collection	12
2.1.3 Results	14
2.1.4 Discussion	20
2.1.5 Threats to Validity	22

2.2 DBScribe: Documenting Database Usages and	
Schema Constraints	22
2.2.1 Detecting SQL-Statements	26
2.2.2 Propagating Constraints and SQL-Statements	
through the Call Graph	29
2.2.3 Generating Contextualized Natural Language	
Descriptions	31
2.3 DBScribe: Empirical Study Design	32
2.3.1 Research Questions:	34
2.3.2 Data Collection	34
2.3.3 Threats to Validity	37
2.4 DBScribe: Empirical Study Results	38
2.5 Related Works on Documenting DCAs	46
2.5.1 Studies on Co-evolution of Schema and Code	46
2.5.2 Extracting Database Information	47
2.5.3 On Documenting Software Artifacts	47
2.6 Conclusion	49
2.7 Bibliographical Notes	50
3 Documenting Unit Test Cases	51
3.1 An Empirical Study on Documenting Unit Tests	54

3.1.1	Research Questions	54
3.1.2	Data Collection	55
3.1.3	Results	58
3.1.4	Threats to Validity	64
3.2	UnitTestScribe: Documenting Unit Tests	65
3.2.1	UnitTestScribe Architecture	68
3.2.2	Unit Test Detector	70
3.2.3	Method Stereotype Analyzer	70
3.2.4	Focal Method Detector	72
3.2.5	General Description Extractor	73
3.2.6	Slicing Path Analyzer	74
3.2.7	Description Generator	74
3.3	UnitTestScribe: Empirical Study Design	76
3.3.1	Data Collection	76
3.3.2	Research Questions	78
3.3.3	Analysis Method	79
3.3.4	Threats to Validity	80
3.4	UnitTestScribe: Empirical Study Results	80
3.4.1	Demographic Background	82
3.4.2	Completeness (RQ4)	83
3.4.3	Conciseness (RQ5)	84

3.4.4 Expressiveness (RQ6)	85
3.4.5 User Preferences (RQ7 - RQ8)	85
3.4.6 Participants' Feedback	87
3.5 Related Works on Documenting Unit Tests	88
3.5.1 Approaches and studies on unit test cases	88
3.5.2 Studies on classifying stereotypes	89
3.6 Conclusion	89
3.7 Bibliographical Notes	90
4 Stereotype-based Tagging of Unit Test Cases	92
4.1 Unit Test Case Stereotypes	95
4.1.1 JUnit API-based Stereotypes	99
4.1.2 Data-/Control-flow Based Stereotypes	101
4.2 Documenting Unit Test Cases with TeStereo	102
4.3 Empirical Study	104
4.3.1 Research Questions	105
4.3.2 Context Selection	107
4.3.3 Experimental Design	112
4.4 Empirical Results	114
4.4.1 What is accuracy for identifying stereotypes?	115

4.4.2 Do the proposed stereotypes improve comprehension of tests cases (i.e., methods in test units)?	117
4.4.3 What are the developers perspectives of the TeStereo-based reports for systems in which they contributed?	121
4.4.4 Threats to Validity	127
4.5 Related Work	128
4.5.1 Stereotypes Definition and Detection	128
4.5.2 Utilizing Stereotypes for Automatic Documentation	129
4.5.3 Automatic Documentation of Unit Test Cases	130
4.6 Conclusion	130
4.7 Bibliographical Notes	131
5 Conclusion	132

ACKNOWLEDGEMENTS

First of all, I would like to express my deepest gratitude to my advisor, Denys Poshyvanyk. Thank you, Denys, for your excellent guidance, patience, and trust. You not only teach me how to do good research but also give me unconditional help when it is needed. I truly feel lucky to be your student.

In addition, I would also like to thank my committee members, Weizhen Mao, Xu Liu, Peter Kemper, and Nicholas Kraft. Thank you for your input, valuable discussions and accessibility.

I was staying at ABB USCRC for about one-third of my Ph.D. period. I would like to thank my colleagues. Thank you, David Shepherd, Nicholas Kraft, Patrick Francis, Andrew Cordes, and James Russell, for your support and collaboration. I learned a lot from you that I would never learn in school.

I want to thank two mentors, Isil Dillig and Mark Grechanik, in the early stage of my Ph.D. period. Thank you, Isil and Mark. I really appreciate your support and guidance.

I also want to thank all SEMERU members that I collaborated with or I had great interactions. Thank you, Bogdan Dit, Mario Linares Vasquez, Qi Luo, Christopher Vendome, Kevin Moran, Michele Tufano, and Carlos Eduardo Bernal Cardenas. We have a lot of memories as friends or collaborators.

Finally, I would like to thank my parents for their continuous encouragement. Thank you for allowing me to realize my own potential. I would also like to thank my wife, Yingzi Pan, for her unconditional love and support. Thank you for going through the hard time with me these years. I would not be where I am today without you.

To my family

LIST OF TABLES

2.1	Developer survey questions and results	13
2.2	Subset of Templates used by DBSCRIBE to generate the database-related descriptions at method level.	28
2.3	Systems' statistics: Lines Of Code, TaBles in the DB schema, # of JDBC API calls involving SQL-statements, # of SQL statements that DBScribe was Not able to Parse, # of Methods declaring SQL statements Locally (ML), via Delegation (MD), Locally + Delegation (MLD), execution Time in sec	35
2.4	Study questions and answers	39
2.5	Answers to " <i>What software engineering tasks will you use this type of summary for?</i> "	43
3.1	Developer Survey Questions and Results	57
3.2	Taxonomy of method stereotypes proposed by Dragan et al.[63] with our proposed modifications	67
3.3	A subset of placeholder templates with examples	71
3.4	Leaf level placeholders	74
3.5	Subject systems: number of Files (NF), number of methods (MD), number of classes (CLS), number of namespaces (NS), number of test cases (TS), Running Time (RT)	79
3.6	Study questions and answers	81

3.7	“What SE tasks would you use <i>UnitTestScribe</i> descriptions for?”	87
4.1	JUnit API-Based Stereotypes for Methods in Unit Test Cases.	96
4.2	C/D-Flow Based Stereotypes for Methods in Unit Test Cases	97
4.3	Accuracy Metrics for Stereotype Detection. The table lists the results for the first round of manual annotation, and second round (in bold) after solving inconsistencies	115
4.4	Questions used for RQ2 and the # of answers provided by the participants for the summaries written without (SW–T eStereo) and with (SW+T eStereo) access to stereotypes	119

LIST OF FIGURES

2.1	Frequency of methods grouped by the ratio between the number of changes to the header comment and number of method changes in methods invoking SQL queries/statements	18
2.2	DBScribe components and workflow	24
2.3	Sets of methods in a DCA. M is the set of all the methods in the DCA, SQL_L is the set of methods executing at least one SQL-statement locally, and SQL_D is the set of methods executing at least one SQL-statement by means of delegation	24
2.4	Iterative propagation of database usage information over the ordered sets defined by the paths in the partial call graph	27
2.5	Iterative propagation of database usage information over the ordered sets defined by the paths in the partial call graph	33
3.1	Developer programming experience	58
3.2	Highest level of education achieved by the developers	58
3.3	Developer industry or open source experience	59
3.4	CoOccurrenceMatrixTests.AddWordsSeveralTimes unit test method of the Sando system	66
3.5	AcronymExpanderTests.ExpandMoreLetters unit test method of the Sando system	68

3.6	UnitTestScribe Architecture. The solid arrows denote the flow of data. Numbers denote the sequence of operations	69
3.7	An example of UnitTestScribe Description for Sando's method CoOccurrenceMatrixTests.AddWordsSeveralTimes	75
4.1	<i>Test Cleaner</i> and <i>Empty Tester</i> method from SessionTrackerCheckTest unit test in Zookeeper	99
4.2	<i>Test initializer</i> method (from TestWebappClassLoaderWeaving unit test in Tomcat) with other stereotypes detected by TeStereo	99
4.3	Source code of the existingConfigurationReturned unit test method in the Apache-accumulo	100
4.4	Source code of the testConstructorMixedStyle unit test method in the Apache-ant system	101
4.5	Source code of the testRead unit test method in the ode system	102
4.6	TeStereo Architecture. The solid arrows denote the flow of data. Numbers denote the sequence of operations	103
4.7	Diversity of the 261 Apache projects used in the study. The figure includes: a) size of methods in unit tests; b) distribution of method stereotypes per system; c) histogram of method stereotypes identified by TeStereo; and d) histogram of number of methods organized by the number of stereotypes detected on individual method	107
4.8	Diversity of the 261 Apache projects used in the study. The figure includes: c) histogram of method stereotypes identified by TeStereo; and d) histogram of number of methods organized by the number of stereotypes detected on individual methods	108

4.9	<i>Logger</i> missed by TeStereo	116
4.10	TeStereo documentation for a test case in the Tomcat project	123
4.11	<i>InternalCallVerifier</i> missed by TeStereo	125

Automatically Documenting Software Artifacts

Chapter 1

Introduction

1.1 Contribution

Software artifacts, such as database schema and unit test cases, constantly change during evolution and maintenance of software systems.

Previous work extensively studied the co-evolution of source code and DB schemas demonstrating that: (i) schemas evolve frequently, (ii) the co-evolution often happens asynchronously (i.e., code and schema evolve collaterally) [127, 49], and (iii) schema changes have significant impact on DCAs' code [49]. 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 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.

The number of unit test cases often grows as new functionalities are 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).

Source code comments are a source of documentation that could help developers understand database usages and unit test cases. 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 [66, 67]. In order to support developers in maintaining documentation for database schema usage and unit test cases, we propose novel approaches, `DBScribe` and `UnitTestScribe`. 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.

1.2 Dissertation Overview

Chapter 2: Documenting Database-Centric Applications

Database-centric applications (DCAs) usually contain a large number of tables, attributes, and constraints describing the underlying data model. Understanding how database tables and attributes are used in the source code along with the constraints related to these usages is an important component of DCA maintenance. However, documenting database-related operations and their constraints in the source code is neither easy nor common in practice.

In Chapter 2, we first present a two-fold empirical study aimed at identifying how developers document database usages at source code method level. Then,

we present a novel approach, namely `DBScribe`, aimed at automatically generating always up-to-date natural language descriptions of database operations and schema constraints in source code methods. `DBScribe` statically analyzes the code and database schema to detect database usages and then propagates these usages and schema constraints through the call-chains implementing database-related features. Finally, each method in these call-chains is automatically documented based on the underlying database usages and constraints.

We evaluated `DBScribe` in a survey with 52 participants analyzing generated documentation for database-related methods in five open-source DCAs. Additionally, we evaluated the descriptions generated by `DBScribe` on two commercial DCAs involving original developers. The results for the studies involving open-source and commercial DCAs demonstrate that generated descriptions are accurate and useful while understanding database usages and constraints, in particular during maintenance tasks. `DBScribe` was originally published in the 25th ACM International Symposium on Software Testing and Analysis (ISSTA'16) [100].

Chapter 3: Documenting Unit Test Cases

Maintaining unit test cases is important during the maintenance and evolution of a software system. In particular, automatically documenting these unit test cases can ameliorate the burden on developers maintaining them. For instance, by relying on up-to-date documentation, developers can more easily identify test cases that relate to some new or modified functionality of the system. We surveyed 212 developers (both industrial and open-source) to

understand their perspective towards writing, maintaining, and documenting unit test cases. In addition, we mined change histories of C# software systems and empirically found that unit test methods seldom had preceding comments and infrequently had inner comments, and both were rarely modified as those methods were modified.

In order to support developers in maintaining unit test cases, we propose a novel approach, `UnitTestScribe`, that combines static analysis, natural language processing, backward slicing, and code summarization techniques to automatically generate natural language documentation of unit test cases. To validate `UnitTestScribe`, we conducted a study with two groups of participants. In the study, we evaluated three quality attributes: completeness, conciseness, and expressiveness. The results of the study showed that `UnitTestScribe` descriptions are useful for understanding test cases. `UnitTestScribe` was originally published in the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16) [95].

Chapter 4: Stereotype-based Tagging of Unit Test Cases

Techniques to automatically identify the stereotypes of different software artifacts (e.g., classes, methods, commits) were previously presented. Those approaches utilized the techniques to support comprehension of software artifacts, but those stereotype-based approaches were not designed to consider the structure and purpose of unit tests, which are widely used in software development to increase the quality of source code. Moreover, unit tests are different than production code, since they are designed and written by following different principles and workflows.

In this chapter, we present a novel approach, called *TeStereo*, for automated tagging of methods in unit tests. The tagging is based on an original catalog of stereotypes that we have designed to improve the comprehension and navigation of unit tests in a large test suite. The stereotype tags are automatically selected by using static control-flow, data-flow, and API call based analyses. To evaluate the benefits of the stereotypes and the tagging reports, we conducted a study with 46 students and another survey with 25 Apache developers to (i) validate the accuracy of the inferred stereotypes, (ii) measure the usefulness of the stereotypes when writing/understanding unit tests, and (iii) collect feedback on the usefulness of the generated tagging reports.

Chapter 2

Documenting Database-Centric Applications

Database-centric applications (DCAs) are software systems that rely on databases to persist records using database objects such as tables, columns, constraints, among the others. These database objects represent the underlying application model, including business rules and terms. Also, developers can create queries and views as a mechanism to traverse or search over the persisted data by following the semantics defined by the database objects. DCA architectures are commonly used for different types of systems ranging from large enterprise systems to small mobile applications. It is not uncommon for many modern DCAs to contain databases comprised of many tables and attributes [128, 35, 29].

The source code and database schemas of DCAs are constantly evolving, oftentimes asynchronously [127]. This makes it particularly challenging for developers who need to understand both how the database is used in the source code and how the model is described by a schema [51].

In addition, database administrators who are in charge of database schemas

may not necessarily be in charge of related source code changes [51] or be able to effectively communicate to developers the modifications to the database schemas. Therefore, complete and up-to-date documentation of the database, the schema, and any constraints is an important artifact to support software evolution. Some existing artifacts designed to capture database schemas are data dictionaries describing all the database objects in a given schema or diagrams (conceptual and physical) depicting tables, attributes, and their relationships. However, navigating and understanding such artifacts can be tedious and time consuming tasks, in particular for large databases.

Source code comments are another source of documentation that can help developers understand nuances of the data model and database usages in the source code. However, recent studies on co-evolution of comments and code showed that the comments are rarely maintained or updated, when the respective source code is changed [66, 67]. Another study by Kajko-Mattsson showed that none of the organizations for eighteen enterprise systems have fully matched all their documentation requirements [83]. To understand if and how database-related statements are commented in source code, we mined Java applications in GitHub that use JDBC for the data access layer, and we found that 77% of 33K+ source code methods do not have header comments; in the case of existing comments, they rarely got updated when related source code was modified. To complement the mining-based analysis, we conducted a survey with 147 open-source developers (Section 2.1). As it turns out, developers rarely write comments detailing database schema constraints (e.g., unique values, non-null keys, varchar lengths) to which developers should adhere in the source code nor document changes in source code dealing with

databases. However, despite the preference of developers for using database documentation or schemas when understanding data models, 65.99% of the surveyed developers consider that tracing schema constraints along call chains in source code is not a “very easy” nor “easy” task. Therefore, there is a clear opportunity for researchers to propose approaches that support automated documentation of source code involving database operations.

In section 2.2, we proposed a novel approach, `DBScribe`, aimed at automatically generating always up-to-date documentation describing database-related operations and schema constraints that need to be fulfilled by developers during software maintenance tasks. `DBScribe` statically analyzes the source code and database schema of a given DCA in order to generate method level descriptions of the SQL-statements related to methods’ execution, and the schema constraints imposed over these statements. The descriptions are generated for source methods executing SQL-statements locally as well as for methods invoking the statements via delegation, which supports developers maintaining different layers or modules of a DCA (e.g., data access or GUI). To the best of our knowledge, this is the first work that proposes a complete solution for automatically documenting source code methods containing SQL operations embedded in a DCA’s code and the corresponding schema constraints.

We validated `DBScribe`’s descriptions in a study involving 52 participants, who were asked to analyze those in terms of completeness, expressiveness, and conciseness (Section 2.4) for five open source DCAs. In addition, we validated `DBScribe` by interviewing the original developers of two commercial web-based DCAs from a Colombian company (Section 2.4). The results show that descriptions generated by `DBScribe` are considered to be complete,

concise, readable, and useful while understanding database usages and constraints for a given DCA, in most of the cases. Moreover, participants consider that this type of summary is useful mostly for maintenance tasks such as program comprehension, debugging, and documentation.

2.1 An Empirical Study on Documenting DCAs

In order to understand developer practices regarding documenting database usages in source code, we designed an empirical study composed of (i) an online survey with contributors of open source Java projects at GitHub and (ii) a mining-based analysis of method comments in these projects. In particular, the *goal* of this study is to understand how developers document or comment methods in source code that invoke database queries or statements. As for the context, 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.

2.1.1 Research Questions:

Commenting database related operations and schema constraints in source code is not a common practice, because comments in source code are mostly promoted as a way to describe the purpose of a code entity (e.g., class, method). Also, there is the assumption of the existence of artifacts covering database documentation (i.e., external documentation). However, this is not always the case, because (i) external documentation can be outdated, and (ii)

understanding large database schemas is a time consuming task. Also despite of the existence of database documentation, it is possible that some database models are more clear for developers when the database objects are described in the context of features implemented in the source code.

One hypothesis that we started to explore in this chapter is that inferring database schema constraints from the source code is not an easy task, and less information about the database can be inferred from the source code methods at higher levels of the call-chains. Therefore, the source code methods that are closer to the user interaction (i.e., the GUI layer) are closer to high-level actions and decoupled from the internal details about the database usages and schema constraints. As an initial effort to explore our hypothesis, in this chapter we aimed at answering the following research questions:

RQ₁ *Do developers comment methods in source code that locally execute SQL queries and statements?*

RQ₂ *Do developers update comments of database-related methods during the evolution of a system?*

RQ₃ *How difficult is it for developers to understand propagated schema constraints along call-chains?*

RQ₁ examines the extent that methods in which SQL queries/statements occur are commented and developers' motivation for commenting (or not) the methods. We consider both responses by developers and a mining-based analysis of source code. **RQ₂** investigates whether comments in headers of methods related to database accesses are likely to be outdated or are modified prior to new releases of the DCA. In **RQ₂**, similarly to **RQ₁**, we compare the

responses from developers to an analysis of the source code at release granularity. While the co-evolution of comments and code have been studied before [66, 67], our study is the first one to investigate co-evolution of database related source code and comments. **RQ₃** investigates whether tracing schema constraints in source code methods that are involved in database-related call-chains is a difficult task.

2.1.2 Data Collection

In order to answer our research questions, we identified a list of 381,161 Java projects on GitHub. We used GitHub's public API [4] to extract all Java projects and locally clone them. We then applied a keyword search to all of the files in each repository with the *.java* extension. In order to identify the projects using SQL queries, we used the following keywords *import java.sql*, *jdbc:mysql*, or *DriverManager.getConnection*. This resulted in 18,828 Java projects using JDBC. We further refined this list by removing projects that were a fork and did not have at least one star or watcher (this filtering aims to avoid projects that are duplicates or abandoned projects). In the end, we had 3,113 projects in our dataset. We extracted the developers of each project and filtered their emails down to 12,887 by removing emails containing *@(none)* and *@localhost* as well as using a regular expression to enforce proper email formatting (`^[a-zA-Z0-9_+]+@[a-zA-Z0-9]+\.[a-zA-Z0-9-]+\.$`). Due to a limitation of the survey platform, we invited 10,000 developers to participate in our survey hosted on Qualtrics [15].

The survey consisted of five questions (**Q₁** - **Q₅**) to understand the extent to which developers document database interactions in source code and their

Question/Answer	Respondents	
Q₁. Do you add/write documentation comments to methods in the source code? (i.e., comments in the header of the method declaration)		
Yes	122	82.99%
No	25	17.01%
Q₂. Do you write source code comments detailing database schema constraints (e.g., unique values, non-null keys, varchar lengths) that should be adhered by the developers in the source code?		
Yes	32	21.77%
No	115	78.23%
Q₃. How often do you find outdated comments in source code?		
Never	1	0.68%
Rarely	28	19.05%
Sometimes	80	54.42%
Fairly Often	35	23.81%
Always	3	2.04%
Q₄. When you make changes to database related methods, how often do you comment the changes (or update existing comment) in the methods, the callers, and all the methods in the call-chains that include the changed methods?		
Never	37	25.17%
Rarely	34	23.13%
Sometimes	45	30.61%
Fairly Often	14	9.52%
Always	17	11.56%
Q₅. How difficult is it to trace the schema constraints (e.g., foreign key violations) from the methods with SQL statements to top-level method callers?		
Very Easy	14	9.52%
Easy	36	24.49%
Moderate	66	44.90%
Hard	23	15.65%
Very Hard	8	5.44%

Table 2.1: Developer survey questions and results.

experience with maintaining this documentation. The questions are listed in Table 2.1. In the table, we define “top-level method callers” as methods in a public interface that start a call-chain that trigger methods with SQL queries/statements. Concerning the mapping between the research questions and the survey questions: **Q**₁ and **Q**₂ relate to **RQ**₁; **Q**₃ and **Q**₄ relate to **RQ**₂; and **Q**₅ relates to **RQ**₃;

In addition to the survey, for **RQ**₁, we counted the number of source code methods with header-comments in the 3,113 projects. We analyzed the latest snapshot of each project by extracting the project’s abstract syntax tree (AST). Then, we automatically detected the methods with database queries/statements to extract the package, class, method name, number of parameters, and method comments when available. We focused on JDBC API calls executing SQL queries or statements. We performed the same analysis at release-level for 264 of the projects and compared the comments for each release to understand whether developers update the comments during the project’s evolution, in part answering **RQ**₂.

2.1.3 Results

RQ₁: It is worth noting that for **RQ**₁ we were interested in answers concerning the general usage of documentation as a practice instead of the frequency. Therefore, **Q**₁ and **Q**₂ are dichotomous questions, and the participants had the chance of complementing the answer with a open field explaining their rationale. Regarding the results, developers mostly recognize the importance of commenting source code methods as a way to increase program comprehension during evolution and maintenance. For instance, 122 developers

answered “Yes” to Q_1 and augmented the response with rationale like:

“Helps explain to others what the method is doing and how to use it, as well as remind the original developer (me) what was the intention of the code”

“Comments make it easier to remember what things do. They are super helpful when returning to old code and when sharing your code with others. I’ve programmed without comments in the past and learned the hard way that comments are, more often than not, indispensable”

From the 17.01% of participants answering “No” to Q_1 , we found typical rationale claiming that methods should be self-documented/self-explanatory.

Despite the high rate of participants recognizing the practice of commenting source code methods, the answers for Q_2 predominantly indicate that the developers do not comment database schema constraints with 78.23% of the respondents answering “No” to Q_2 . This contrast demonstrates that database-related information is not likely contained in the method declarations, because method comments are mostly for describing the purpose of the method instead of implementation details, and the documentation of database objects is an obligation of an external document or the database schema. Some responses from participants supporting the preference for external documentation are the following:

“The database schema and documentation takes care of that. I can always look at the table definition very easily.”

“I use [database-engine] and the constraints can be checked typing a sql consult or even using a Tool.”

“Although I strongly believe comments are important, database comments are the gray area. Comments related to the database schema and its constraints I consider to be irrelevant to the code using it. The schema, its details, and any quirks about it should be outlined in a separate document.”

“The comments should be stored in the database itself - which is not supported by most databases I know. Writing in the source code means duplicate effort - need to keep the source code synchronized with the schema. Also, the database is sometimes accessed by programs written in different languages - maintaining the comments up-to-date in ALL the source codes would be impossible.”

Another prominent reason for not documenting database usages and constraints in method headers is the usage of Java Annotations in ORM frameworks, which explicitly declare in the code the schema constraints:

“This can be mostly handled through proper design. If using an ORM we can specify field lengths in attributes, that can provide validation as well as documenting if for developers. ”

“ORM initialization makes it clear what the scheme constraints are”

“The schema is already described in ORM code.”

Results from the mining study confirm developers preferences. In the analyzed source code (i.e., 3,113 projects), we identified a total of 33,045 methods invoking SQL queries/statements. Of these methods, 25,450 did not have any comment, while 7,595 methods were commented. These numbers reinforce the result of Q_2 since 23% of the methods with database access were

documented and 21.77% of developers indicated that they do in fact document such database interactions.

Summary for RQ₁. 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.

RQ₂: In order to understand whether developers update comments relating to database queries/statements, we sought to understand the prevalence of outdated comments (**Q₃**). Combining “Never” and “Rarely,” 19.73% of developers suggest that comments are regularly updated in the systems that they implement or utilize. The remaining 80.27% of developers find outdated comments. Of those 80.27% of developers, 25.85% indicated a relative high frequency of encountering outdated comments. These results suggest that outdated comments are relatively prevalent (i.e., they are not a rare occurrence), which confirms that the comments are rarely maintained or updated, when the respective source code is change [66, 67].

When we consider the prevalence of developers updating their own comments regarding changes to database-related methods (**Q₄**), we found 50.3% of respondents rarely or never updated the comments. Only 21.08% of respondents updated comments with a relatively high frequency (i.e., fairly often or always). The remaining 28.62% indicated it was neither a rare nor frequent occurrence. Therefore, the survey results suggest that it would fairly probable for such comments to be outdated. These results reinforce the problems that we observed with **RQ₁**. Not only are methods with database queries undocumented,

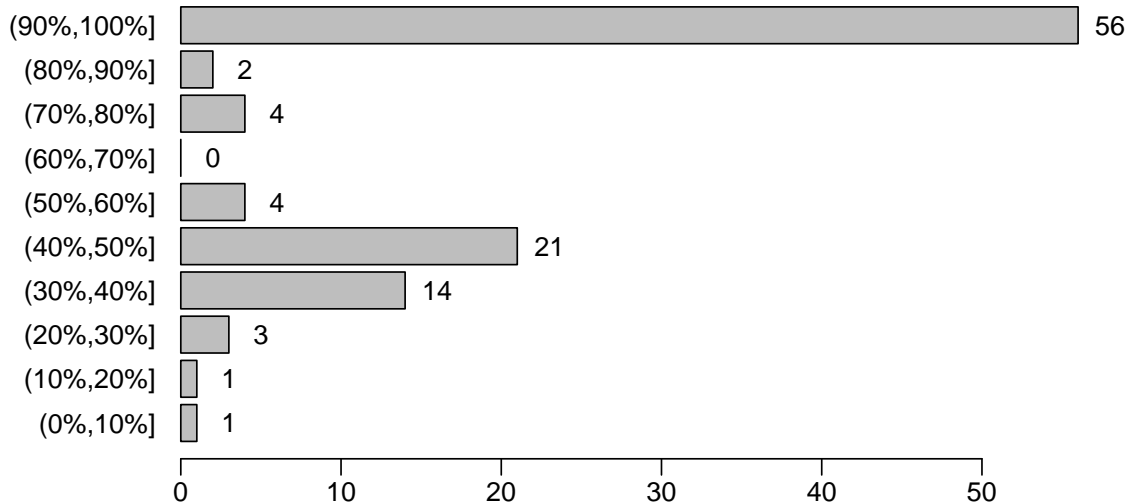


Figure 2.1: Frequency of methods grouped by the ratio between the number of changes to the header comment and number of method changes in methods invoking SQL queries/statements.

but in the case of commented methods they are also likely to be outdated.

We also analyzed RQ_2 by relying on open source systems. We mined 264 projects that had explicit releases in GitHub to identify whether methods invoking database queries/statements updated their comments. Overall, developers did not update the comments when the methods were changed. We found 2,662 methods that invoke SQL queries/statements in the 264 projects. Of these 2,662 methods, 618 methods were updated during the history of these projects and experienced a total of 1,878 changes. 512 out of the 618 methods that changed did not have changes to their comments. The 512 method experienced on average 2.5 changes ($min = 1$, $Q_1 = 1$, $Q_2 = 2$, $Q_3 = 2$, $max = 199$) during their entire history. The rest of 106 methods (17.15%) were changed 597 times and experienced on average 5.63 changes ($min = 1$, $Q_1 = 2$, $Q_2 = 3$, $Q_3 = 5.75$, $max = 198$). In those 106 methods, we found 459 out of 597 method changes also experienced an update to the method comment. Finally, we computed the

ratio between changes to header comments and methods changes; a 100% ratio means that every time a method was changed, the header comments was also changed. Figure 2.1 depicts the ratio of changes between header comments and source code for the 106 methods that experienced changes. For instance, we only found 66 methods in which more than 50% of the method changes were accompanied by a change to the header comment.

Summary for RQ₂. 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.

RQ₃: Answers to Q₅ show a different perspective. Despite most of the developers thinking database documentation and schema are enough to understand schema constraints and they do not document database-related methods, answers to Q₅ show that only 34.01% of respondents indicated that it was “easy” or “very easy” to trace database constraints along the call-chain to the top-level caller methods. The remaining 65.99% found it at least moderately difficult, with 21.09% indicating it was “hard” or “very hard.” These responses indicate that tracing database constraints along call-chains is a non-trivial task. A call-chain represents the implementation of a feature in source code. This suggest that, even if external documentation or database schema is available, maintaining or implementing a new feature of a system involving database operations may be a non-trivial task, because of the effort required to trace schema constraints across the call chains. However, more empirical validation is

required to support this claim.

Summary for RQ₃. 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.

2.1.4 Discussion

This preliminary study suggests that (i) documenting database usages and constraints is not a common practice in source code methods, (ii) developers do not update comments when changes are done to database-related methods, and (iii) tracing schema constraints through call-chains in the call graph is not an easy task in most of the cases. While results for **RQ₁** and **RQ₂** describe developers rationale for not documenting database related operations in source code, the findings in **RQ₃** present a different perspective in terms of whether current practices for documenting databases are enough or useful for supporting developers. Schemas and database documentation have the purpose of describing the physical data model supporting a system. However, it is still unclear if this type of documentation is effective and efficient when maintaining software systems supported on large databases. Another aspect is the quality of the documentation; when the schema is complicated or hard to access, documentation is the last line of defense for understanding/querying the schema. However, there is also always the problem of outdated documentation.

In addition, understanding/querying updated documentation can be a time-consuming task, when it is not designed to support easy browsing or a

specific task. For instance, let's assume a scenario in which a maintainer wants to identify the schema constraints that are involved in a feature implemented in the source code. A feature may involve different operations on several database objects, and database documentation and schemas are not intended to describe constraints and relationships at the feature-implementation level. Therefore, the maintainer has to identify the constraints by exploring the code and understanding the available documentations. Moreover, current approaches for automated documentation aim at describing the purpose of a code entity (e.g., class, method), but neither target specific evolution/maintenance tasks nor describe the entity as part of an architecture (i.e., the description of a code entity in the GUI layer should not follow the structure of a description for an entity in the data access layer).

In summary, future work should be devoted to providing developers with tools for automatic documentation that support specific tasks, in particular for evolution and maintenance of DCAs. Our results suggest that automatic generation of database-related documentation is required to support evolution/maintenance tasks. Using the results in **RQ₁** as a reference, developers working on features involving the 25,450 undocumented database-related methods — from the 3,113 analyzed projects — might find benefit in an automated approach that assures the methods are properly documented and updated. Also, the automated approach might benefit the 65.99% of the surveyed developers that did not consider tracing schema constraints along call chains as a “very easy” nor “easy” task.

2.1.5 Threats to Validity

The *construct* threat to validity relates to the observations from the developer survey and results of mining the database usage. In terms of our survey, we did not infer behavior from the survey and only reported the experience as indicated by developers and we do not provide rationale for the observations. Since we relied on projects on GitHub, it is possible that project histories are incomplete due to the relative young age of the forge or that releases were not properly tagged. In terms of *internal* threats to validity, it is possible that the developers responding to the survey had more difficulty with database-related documentation. However, the results of the survey suggest that participants had a range of experience and no single response was overly represented, which would indicate a clear bias. The *external* threats to validity relate to generalizing the results. We do not assert that the results apply to all developers or developers using other database models (e.g., ORM). Our results represent a subset of Java projects on GitHub, and other languages or forges may produce a different findings. However, GitHub is the most popular forge and our approach applies to Java projects using JDBC only.

2.2 DBScribe: Documenting Database Usages and Schema Constraints

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; 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. Our method-level documentation can provide developers with descriptions that work at different layers for a given DCA. This type of documentation is useful for (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. For instance, Table 2.3 lists the execution time in seconds of DBScribe when running on a MacBookPro laptop with a 2.4GHz Intel Core 2 Duo processor and 4GB of DDR3 RAM.

The architecture of DBScribe is depicted in Figure 2.2. 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 only the call-chains including the methods executing SQL-statements 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. One current limitation of DBScribe's implementation is that it currently covers only SQL-statements invoked by means of JDBC API calls. Future work will support ORM frameworks such as JPA, Hibernate, and iBATIS

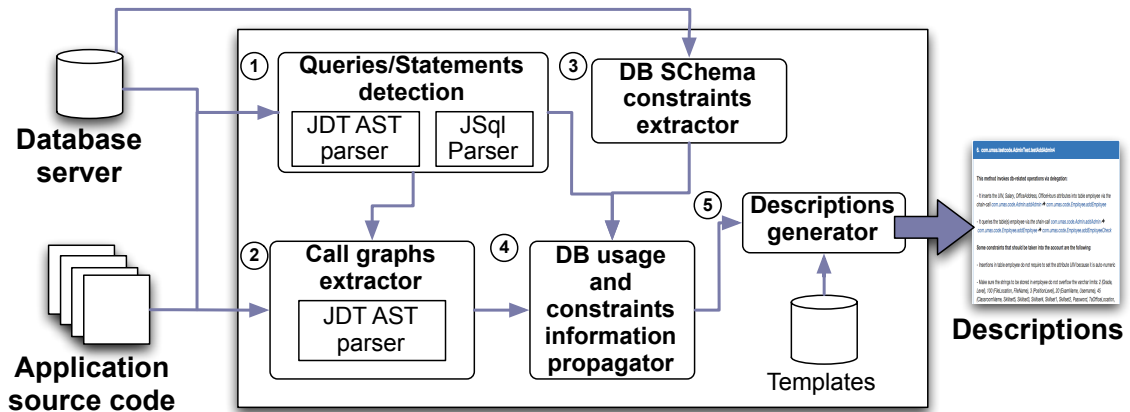


Figure 2.2: DBScribe components and workflow.

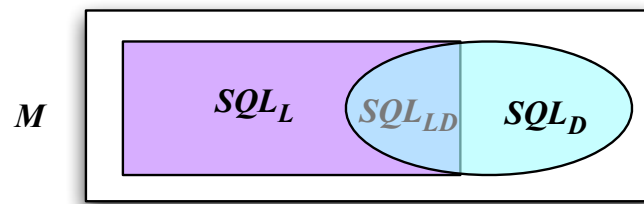


Figure 2.3: Sets of methods in a DCA. M is the set of all the methods in the DCA, SQL_L is the set of methods executing at least one SQL-statement locally, and SQL_D is the set of methods executing at least one SQL-statement by means of delegation.

(similarly to [114]).

Each phase in DBScribe's workflow is described in the following subsections, however, we first provide **formal definitions** that are required to understand the proposed model:

- M is the set of all the source code methods/functions in a DCA, and m is a method in M ;
- SQL_L is the set of methods $m \in M$ that execute at least one SQL-statement locally (Figure 2.3);
- SQL_D is the set of methods $m \in M$ that execute at least one SQL-statement by means of delegation through a path of the DCA call graph (Figure 2.3);

- G is the call graph of a DCA involving all the methods $m \in M$, and $G_{SQL} \subset G$ is the call graph including only the methods in $SQL_L \cup SQL_D$. It means that G_{SQL} is a call graph of all the methods in M that execute at least one SQL-statement locally or by delegation;
- P is the set of paths p (a.k.a., call-chains) starting at any method $m_i \in G_{SQL}$ and finishing at any method $m_j \in SQL_L$. It is possible that the number of methods in some p 's is equals to 1 (i.e., $|p_i| = 1$); those cases represent unused methods, or methods in a upper layer of the DCA invoking SQL-statements. It is worth noting that P can be a disconnected graph;
- The methods in a path p are an ordered set defined by the binary relationships (represented with the symbol $<$) between a callee and a caller. For instance, given a method $m_i \in SQL_L$, m_j a caller of m_i , m_k a caller of m_j , and so on, until all the methods in the path p are exhausted, the ordered set is $m_i < m_j < m_k < \dots < m_s$. The position in the ordered set, is the attribute l , which represents the “level” of the method in the path p , and is in the range $[0, |p| - 1]$. In the example, the l value for m_i is zero, and the l value for m_k is two. Conversely to the level, the depth d of a method in a path is the position in the ordered set but in the direction of callers or callees; for instance, the level l of m_i in our example is zero, but the depth d is $|p| - 1$.
- QS_m is the set of tuples $qs = \langle literal, T, A, type \rangle$ representing the SQL-statements executed locally in method m . Each tuple qs has a SQL string literal, tables (T) and attributes (A) from the database schema referenced in the literal, and the SQL-statement type.

- \vec{C}_m is the set of methods called by method m (i.e., callees), and \overleftarrow{C}_m is the set of methods calling method m (i.e., callers).

2.2.1 Detecting SQL-Statements

All the methods in M are analyzed by first identifying JDBC API calls that execute SQL-statements; using this approach we avoid SQL-statements that are declared as strings but never executed. In order to identify these JDBC API calls, we traverse the AST of each method in M ; during the traversal we keep a working map SV with all the `String` and `StringBuffer` variables instantiated in the method as well as the current values. We update the variables in SV after a new initialization or after a concatenation operation with the plus (+) operator or the `StringBuffer.append` method.

During the traversal, we also keep track of invocations to JDBC methods that execute or prepare SQL-statements: `Statement.execute`, `Statement.executeQuery`, `Statement.executeUpdate`, and `Connection.prepareStatement`. If the `String` argument in the API call is a literal, we add the literal to the list of SQL-statements QS_m declared in the method m ; if the string argument is an infix expression or a variable name, we infer the value of the argument by resolving the expression/variable state with the values in the map SV . The literals and inferred variable values are parsed by using the `JSqlParser` [8] to identify the statement type, and the tables and attributes involved in the SQL-statement (this information is required to generate the textual descriptions as described in Section 2.2.3). Then, we add the resolved SQL-statement (i.e., literal, tables, attributes, and type) to the list QS_m . One limitation in this procedure is that we do not perform inter-procedural analysis;

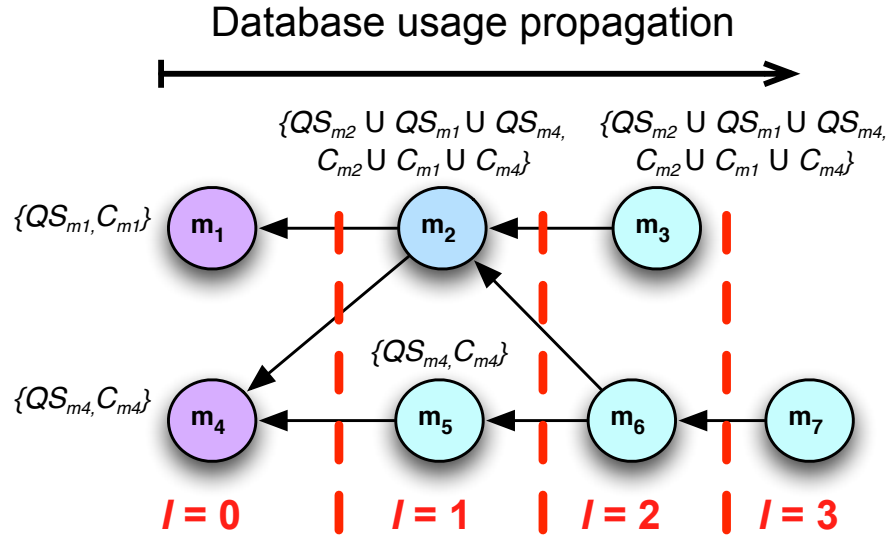


Figure 2.4: Iterative propagation of database usage information over the ordered sets defined by the paths in the partial call graph

thus, we do not resolve values that are returned by inter-procedural calls or values passed as arguments to the analyzed method (future work will be devoted to inferring the SQL queries/statements by using symbolic execution). This leads to cases in which some literals and variable values are not parsed by JSqlParser. For instance, Table 2.3 lists in column “NP” those cases in which DBScribe was not able to parse the SQL-statements in seven DCAs that we analyzed, and column “S” lists the number of JDBC API calls that execute SQL-statements. More details about DBScribe’s evaluation are provided in Section 2.4.

The \vec{C}_m and \overleftarrow{C}_m sets for each $m \in M$ are also collected during the traversal to avoid a second pass on the DCA code. Both sets are used to generate the G_{SQL} graph required to propagate SQL-statements and constraints.

Section: Local SQL-statements	
Type	Template
Header	This method implements the following db-related operations:
Insert	It inserts the <attr> attributes into table <table>
Update	It updates the <attr> attribute(s) in table <table>
Header	This method implements the following db-related operations: the following db-related operations:
Insert	It inserts the Username , Passwd attributes into table logindetails
Update	It updates the IsCurrent attribute(s) in table semester
Section: Delegated SQL-statements	
Header	This method invokes db-related operations by means of delegation:
Query	It queries the table(s) <table> via <method>
Delete	It deletes rows from table(s) <table> via <method>
Header	This method invokes db-related operations by means of delegation:
Query	It queries the table(s) People via the call-chain <code>JobApplication.addApplicationDetails</code> → <code>Student.checkIfStudent</code>
Delete	It deletes rows from table(s) gradingsystem via a call to the <code>GradeSystem.deleteGrade</code> method
Section: Schema constraints	
Header	Some constraints that should be taken into the account are the following:
Varchar	Make sure the strings to be stored in <table> do not overflow the varchar limits: <limits>
Non-null	Make sure the values in <table>.<attr> are not null
Header	Some constraints that should be taken into the account are the following:
Varchar	Make sure the strings to be stored in employee do not overflow the varchar limits: 2 (Grade, Level), 100 (FileLocation, FileName)
Non-null	Make sure the values in employee.Salary are not null

Table 2.2: Subset of Templates used by DBSCRIBE to generate the database-related descriptions at method level. Examples from the systems used in the study are also provided

2.2.2 Propagating Constraints and SQL-Statements through the Call Graph

The DB schema contains a set of constraints that need to be fulfilled when realizing insertions, updates, or deletions on the DB. For instance, changing/deleting the value of a column that serves as a foreign key in other tables cannot be performed when there are references from other tables to that column. As mentioned before, this type of constraints cannot be inferred easily from the source code, and the constraints provide useful information that can help in understanding source code methods in abstract layers that are not close to the DB (e.g., \mathbf{m}_7 in Figure 2.4). Therefore, in addition to linking source code methods to SQL-statements, we extract from the DB schema — by querying the master schema in the DB server— the constraints that are defined on the attributes and tables in the sets $QS_m, \forall m \in SQL_L$, i.e., the constraints that apply to all the attributes and tables involved in SQL-statements executed by the methods in the application. In particular, we extract the following constraints: (i) auto-numeric columns, (ii) non-null columns, (iii) foreign keys, (iv) varchar limits, and (v) columns that should contain unique values. Consequently, each method m in SQL_L has a list X_m of schema constraints that apply to the SQL-statements executed locally by m .

Both QS_m and X_m sets contain information for all of the methods in SQL_L ; however, the methods in SQL_L are not the only methods that can benefit from documentation describing DB usages and schema constraints. Developers inspecting, using, or updating methods that execute SQL-statements by means of delegation (see SQL_D in Figure 2.3), could require documentation describing DB usages and constraints across all methods involved in the DB-related

call-chains. Thus, we propagate the information in the QS_m and X_m lists to all the methods in SQL_D . Notably methods in the intersection of SQL_L and SQL_D have at least one local execution of an SQL-statement and at least one by delegation.

To describe the propagation algorithm, we use Figure 2.4 as a reference. The figure depicts a partial call graph with nodes representing source code methods and directed edges going from a caller to a callee. The background color of the nodes (the colors are the same from Figure 2.3) represents the set to which each method belongs. For instance, light purple is for methods in SQL_L , light blue for methods in SQL_{LD} , and cyan for methods in SQL_D . Only the methods in SQL_L (including SQL_{LD}) execute SQL-statements locally.

The propagation is done iteratively by using the node level l (see definition at the beginning of Section 2.2) as the iteration index, until the maximum l in the call graph is reached. This iterative execution over l assures that the values from methods with a lower level l are computed before the methods with a level $l + 1$. This step is required because one method can have more than one callee in the graph G_{SQL} . The nodes (i.e., methods) with $l = 0$ are methods belonging to SQL_L ; thus, the lists QS_m and X_m for those methods were computed previously. However, the lists of queries/statements and constraints that concern the methods with $l > 0$ are the unions of the local QS_m and X_m lists (if any) and the lists propagated from the callees. For example, the node \mathbf{m}_2 calls \mathbf{m}_1 and \mathbf{m}_4 , and belongs to SQL_{LD} , which means that \mathbf{m}_2 executes at least one SQL-statement locally and at least two by means of delegation in \mathbf{m}_1 and \mathbf{m}_4 . Therefore, the complete sets of SQL-statements and constraints that concern \mathbf{m}_2 are $\{QS_{m_2} \cup QS_{m_1} \cup QS_{m_4}\}$ and $\{X_{m_2} \cup X_{m_1} \cup X_{m_4}\}$, respectively. The case

for the methods in cyan is different, because they do not execute SQL-statements locally; consequently, the set of SQL-statements that concerns a method m in SQL_D is the union of SQL-statements and constraints from its callees, and the set of constraints that concerns the method is the union of the constraints from its callees. For example, \mathbf{m}_5 does not execute SQL-statements locally. Hence, the SQL-statements and constraints that concerns \mathbf{m}_5 are the same from its single callee \mathbf{m}_4 .

2.2.3 Generating Contextualized Natural Language Descriptions

The final phase in DBScribe is to generate contextualized natural language descriptions by using predefined templates. In general, a description for a method m in G_{SQL} consists of three parts: (i) a block (i.e., a header plus a list of sentences) describing SQL-statements executed locally, (ii) a block describing SQL-statements executed by means of delegation and the path in the call graph to the execution, and (iii) a block describing the constraints that should be taken into account as a result of the SQL-statements that are executed locally or by delegation. DBScribe only generates descriptions for methods that are related to database operations.

A subset of DBScribe templates is listed in Table 3.3, while the complete list is provided in our online appendix [2]. Each template has tokens identified with $\langle \dots \rangle$, which are replaced with values from the sets of SQL-statements and constraints that concern a method m . In the case of execution by delegation, the templates include the token $\langle \text{method} \rangle$; this token is replaced by a single method call (see template Delegation-Delete), or by a call-chain that goes over a path

from m to the method, where the corresponding SQL-statement is executed. The sentences in the constraints paragraph are generated based on the SQL-statements concerning the method m . For instance, the sentences generated with constraints-related templates in Table 3.3 are only included if the method executes insertions/updates locally or by delegation. Our current implementation of DBScribe generates the descriptions as HTML pages that have hyperlinks to the methods mentioned in the summary (as in the delegation-related sentences); this allows for easy browsing and navigation of the methods in the call-chain. Our decision for generating external documentation instead of source code comments was mainly driven by the results of our preliminary study to also cover preferences of developers that value external documentation over comments in source code.

2.3 DBScribe: Empirical Study Design

We conducted a user study to evaluate the usefulness of DBScribe at generating descriptions for source code methods that execute SQL-statements locally, by means of delegation, and the combination of both. The *goal* of this study is to measure the quality of the descriptions generated by DBScribe as perceived by developers. As for the *context*, we used seven DCAs listed in Table 2.3. The first five systems are open source DCAs hosted at GitHub and SourceForge. The last two DCAs are industrial web Java applications developed by a Colombian company (*LIMINAL Ltda*). It is worth noting that LOC reported in Table 2.3 only include .java files; for instance, web side files like JSP, HTML and CSS were not included. Also the numbers in columns “ML”, “MD”, and “MLD” are the ones

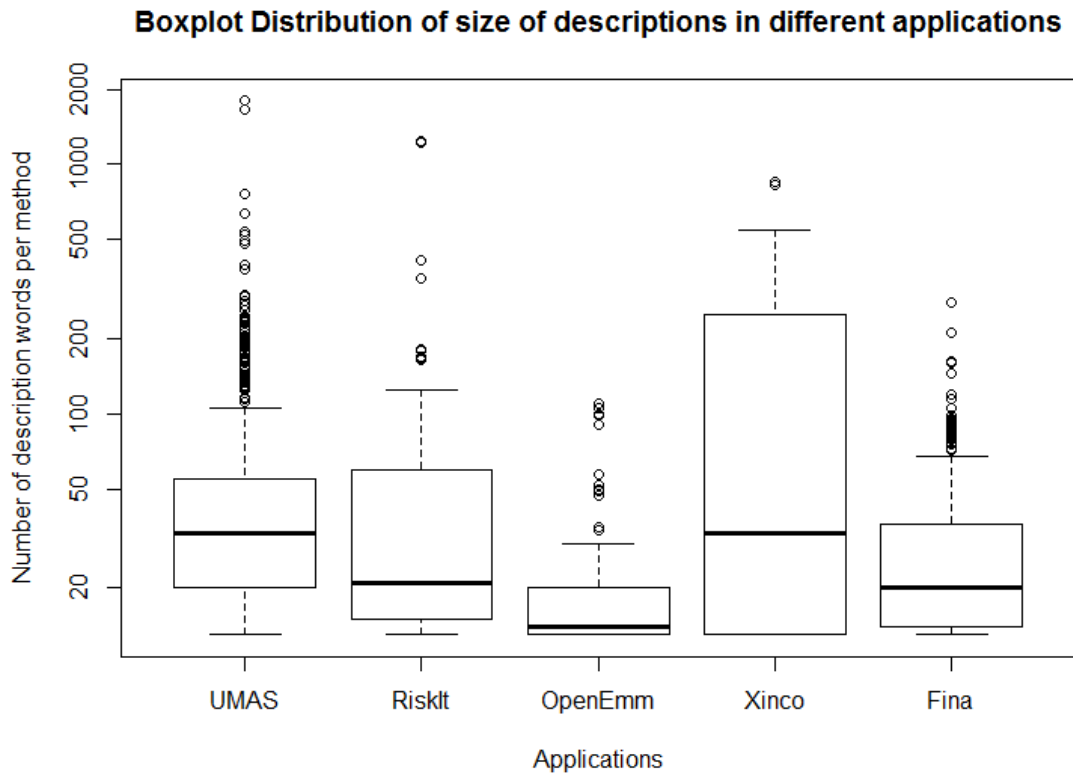


Figure 2.5: Iterative propagation of database usage information over the ordered sets defined by the paths in the partial call graph

reported by DBScribe.

We selected subject systems with the following constraints in mind: (i) the systems should rely on JDBC and MySQL for the data access layer, since the current version of DBScribe was designed to detect SQL-statements from JDBC API calls and extract schema constraints from MySQL DBs, and (ii) the systems should pervasively use SQL-statements. Figure 2.5 depicts distribution of size of descriptions in different applications.

2.3.1 Research Questions:

In the context of our study, we formulated the following five research questions (RQ):

RQ₁ *How complete are the database usage descriptions generated by DBScribe?*

RQ₂ *How concise are the database usage descriptions?*

RQ₃ *How expressive are the database usage descriptions?*

RQ₄ *How well can DBScribe help developers in understanding database related source code methods?*

RQ₅ *Would developers of DCAs use DBScribe descriptions?*

RQ₁ to **RQ₃** aim at measuring the quality of the descriptions as perceived by developers that have explored the source code and the database schema. **RQ₄** aims at identifying whether the descriptions are useful for developers and the software development tasks that can take advantage of this type of description. **RQ₅** is for exploring the potential usefulness of DBScribe for supporting DCAs and potential adoption by industrial DCA developers and maintainers.

2.3.2 Data Collection

We used descriptions generated by DBScribe for methods of the open-source DCAs in an open survey with students, faculty, and developers. 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

System	LOC	TB	S	NP	ML	MD	MLD	T
UMAS[23]	32K	122	211	4	125	431	67	29.53
Riskit rev.96[16]	12.7K	13	111	2	35	9	44	15.02
FINA 3.4.2[3]	139.5K	52	710	26	312	118	99	130.78
Xinco rev.700[26]	25.6K	23	76	15	26	22	21	31.41
OpenEmm 6.0[14]	102.4K	68	200	110	73	12	1	104.78
System 1*	73.2K	53	398	27	262	660	24	71.07
System 2*	28.4K	24	164	8	106	247	44	40.13

Table 2.3: Systems' statistics: Lines Of Code, TaBles in the DB schema, # of JDBC API calls involving SQL-statements, # of SQL statements that DBScribe was Not able to Parse, # of Methods declaring SQL-statements Locally (ML), via Delegation (MD), Locally + Delegation (MLD), execution Time in sec.

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 summaries per system to make sure our survey could be completed in one hour to avoid an early survey drop-out. For the evaluation, we relied on the same framework previously used for assessing automatically generated documentation [142, 110, 50]. Therefore, the descriptions were evaluated in terms of *completeness*, *conciseness*, and *expressiveness*. In addition, we sought to understand the preferences of the participants concerning DBScribe's descriptions.

We designed and distributed the survey using the Qualtrics [15] tool. We asked participants to evaluate DBScribe's descriptions by following a two-phase procedure. In the first phase, we asked developers to *manually* write a summary documenting the SQL-statements executed (locally and by means of delegation)

as part of a call to a given source code method and the constraints that should be considered by developers when understanding that method. In particular, each developer was provided with the source code of the DCA, an entity-relationship diagram, and six source code methods to document; we also provided the SQL script to create the database schema as an optional artifact that can be used during the task. We decided to use only six methods per DCA, because writing each summary requires detailed inspection of the source code and the databases. This phase was designed to make sure that the participants understood the source code before evaluating DBScribe's descriptions.

In the second phase, we asked participants to compare their own (manual) summaries to DBScribe's descriptions. For each DBScribe description, the participants rated the three quality criteria (i.e., completeness, conciseness, expressiveness) with the options listed in Table 2.4. In addition, we asked them to provide a rationale for their choices; the evaluation criteria and the rationale provided the answers to **RQ₁** to **RQ₃**. For **RQ₄**, we included two questions regarding the usefulness of the descriptions in the survey. To measure the programming experience of the participants, we included background/demographic questions [65].

For the case of the industrial systems (i.e., **RQ₅**), two original developers of System 1 and System 2 from *LIMINAL Ltda* [10] were interviewed. We provided them with a complete DBScribe report (i.e., an HTML page with descriptions for all the methods with hyperlinks) and asked to read the report and analyze the code. The report also organizes the methods in the three groups in Figure 2.3 to enable easier browsing. While all the reports for the open source DCAs are provided in our online appendix, we were not allowed to publicize the reports for

the industrial DCAs.

During the interview, in addition to the questions from the open survey, we asked the following: (i) *Only focusing on the content of the document without considering the way it has been presented, do you think all the database-related methods are listed in the document?*; (ii) *Is the document useful for understanding the database usages in the system?*, (iii) *How could we improve the document?*; (iv) *What kind of information would you like to include/remove?*

2.3.3 Threats to Validity

In order to reduce the threats to *internal* validity and maximize the reliability of the results of evaluation, we confirmed that the participants explored and understood the source code before evaluating the summaries generated by DBScribe. In terms of evaluation, we used a well-known framework that has been applied previously to evaluate the quality of natural language summaries of software artifacts. Also, in order to avoid any type of bias because of the expectations of the participants during the study, we informed the participants that they had to evaluate generated descriptions only after completing the phase in which they needed to write their own summaries. Concerning the threats to *external* validity, we do not assert that the results in the second study apply to all developers or developers using other database models (e.g., ORM frameworks). However, the set of participants is diverse in terms of academic/industry experience, and 42.3 percent of the participants have more than five years of experience in Java. Although the study was done on only five open-source and two industrial DCAs, when designing the study we selected a diverse set of source code methods belonging to different layers of the systems' architecture,

which means that we evaluated methods executing SQL queries/statements locally, through delegation, and a combination of both. In addition, we do not perform branch analysis of source code in DBScribe. We catch as many branches as possible, likely overestimating the results. In the future, we will rely on static analysis techniques to improve the precision of our approach [45].

2.4 DBScribe: Empirical Study Results

We obtained responses from 52 participants: 15 responses for both UMAS and Riskit, eight responses for Xinco, and seven responses for the other two open source DCAs, Openemm and Fina. In terms of background, we had the following distribution of the participants: three undergraduates and 35 graduate students (M.S/Ph.D), two post-docs, seven developers/industry researchers, and five faculty members. Three participants participated in the study twice (voluntarily), i.e., they analyzed the descriptions for two different systems. 24 of our participants (46.1%) asserted that they had past experience in industry. Concerning the programming experience in Java, 22 of participants (42.3%) had at least five years of experience; the mean value is four years of experience.

We evaluated the quality of DBScribe generated descriptions by considering three attributes: *completeness*, *conciseness*, and *expressiveness*. For *completeness*, we aimed at assessing whether the descriptions cover all the important information (**RQ₁**). For *conciseness*, we aimed at evaluating whether the descriptions contain useless information (**RQ₂**). For *expressiveness*, we aimed at checking whether the summaries are easy to understand (**RQ₃**). Since we asked participants to evaluate three attributes for six descriptions for each

Completeness: Only focusing on the content of the description without considering the way it has been presented, do you think the message is complete?	Rating
• The description does not miss any important information	205(65.7%)
• The description misses some important information to understand the unit test case	91(29.2%)
• The description misses the majority of the important information to understand the unit test case	16(5.1%)
Conciseness: Only focusing on the content of the description without considering the way it has been presented, do you think the message is concise?	Rating
• The description contains no redundant/useless information	221(70.8%)
• The description contains some redundant/useless information	77(24.7%)
• The description contains a lot of redundant/useless information	14(4.5%)
Expressiveness: Only focusing on the content of the description without considering the completeness and conciseness, do you think the description is expressive?	Rating
• The description is easy to read and understand	241(77.3%)
• The description is somewhat readable and understandable	60(19.2%)
• The description is hard to read and understand	11(3.5%)

Table 2.4: Study questions and answers.

DCA, we had a total of 312 answers for each attribute (6×52). Table 2.4 reports both raw counts and percentages of answers provided by the participants; the detailed results are also publicly available in our online appendix [2].

RQ₁ (Completeness): 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. In other words, our approach is able to generate DB-related descriptions for source code methods that cover all essential information in most of the cases (**RQ₁**). We also examined answers with the lowest ratings. One comment mentioned: “*The description does not make it clear that the time-slot is not always added to the*

table.” The reason for this comment is that we did not apply branch analysis when generating descriptions (see Section 2.3.3). However, this would not influence the completeness of summaries, since we chose to over-approximate in order to catch important information.

When comparing the summaries written by participants to DBScribe’s descriptions we found that only 24 out of 312 human-written descriptions include specific information about the schema constraints. Most of the human-written descriptions (i.e., 5 out of 7 for RiskIt) detailing constraints correspond to the methods at the lowest level of the call-chains (i.e., the methods executing statements locally). These findings corroborate our hypothesis that the methods in the higher levels in the call-chains may be more difficult to understand with respect to relevant DB operations and the schema constraints. Therefore, DBScribe descriptions are not only useful for methods that are architecturally close to the DB, but also for source code methods in layers that are more close to the end-user (e.g., GUI layer).

RQ₂ (Conciseness): 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.

Again we examined the answers with the lowest ratings. Participants’ comments included the following: *“too much detail and in the end these kinds of errors are less likely to occur”*; *“This data feels too low level.”* We closely checked our generated documents for those methods. Our observation is that DBScribe’s documentation sometimes contains unnecessary information for the task we assigned. The extra information is unavoidable because the documentation is produced without taking into account a particular task on

which a developer may be working. In addition, since we provided call-chains in the documentation, the descriptions for methods in the top level of the call hierarchy may appear rather verbose. For example, most of low ratings (10 out of 14) were for the descriptions of methods situated in the middle or higher levels of call-chains.

RQ₃ (Expressiveness): in 77.24% of the answers, DBScribe's descriptions were evaluated as easy to read, while only 3.53% answers indicated that the descriptions were hard to read. We analyzed the user feedback from the participants who provided the lowest ratings for expressiveness. Those participants who thought some comments were hard to read also claimed that the contents of documents are complicated. Similar to *Conciseness*, our descriptions are attempting to capture more important information, which may come at the expense of *expressiveness*. We also observed that there were only two out of 11 responses with the lowest rating for the methods situated in the lowest level of call-chains over all systems. Thus, descriptions for methods only invoking SQL-statements locally, methods that are the easiest to read.

The following comments illustrate some of the reasons why participants evaluated DBScribe's descriptions mostly positively (completeness, conciseness, and expressiveness):

"It is useful when we need to know all the entities (i.e. tables, constraints, indexes, etc) involved in a database operation. This information helps a lot if someone needs to modify/extend the code."

"This description definitely outperformed the description that I just made. The details shown by this description really help to understand all the entities involved in the creation of a new user, and this information is helpful when

someone tries to modify/extend the source code.”

“It summarizes the database accesses efficiently that might be spread over many different methods. Even if all related database operations are contained in the respective method directly, the summary is much easier to read than finding the specific statements in the code.”

“The generated summaries are useful because they show all related db operations and also show another information related with the constraints, data types. With the constraints and validations allows to developer to understand any business logic restrictions.”

RQ₄ (User preferences): 48 participants (92.3%) claimed that DBScribe generated descriptions would be useful for understanding the database usages in source code methods. When looking into the details for each system, we found that 100% of the answers were positive for UMAS, Xinco, and Openemm; we have three negative responses for RiskIt and one negative response for Fina. Although we do not have enough evidence to claim a relationship between uses preferences and the type of DCA system for which descriptions are generated, the results suggest that DBScribe is more helpful specifically for DCAs with larger DBs and more complex chain-calls. In our case, Riskit has less complicated call hierarchy and database design than others (see Table 2.3).

We also asked the participants for which software engineering tasks they would use these descriptions. We categorized the answers in Table 2.5. The most answered tasks are related to incremental change, such as program comprehension, implementing new features, and impact analysis. The second

Category	Subcategories
Incremental change (21)	Program comprehension (11), Add new features (4), Impact analysis (4), Concept location (1), Change database schema (1)
Bugs (10)	Debugging (6), Bug fixing (4)
Maintenance (10)	Refactoring (2), Re-modularization (2), Re-engineering (2), Maintenance (4)
Others (15)	Documentation (9), Systems integration (1), Test cases design (2) Change db-related code (3)

Table 2.5: Answers to “What software engineering tasks will you use this type of summary for?”

most reported category is “Bugs”, where participants mentioned debugging six times, and bug fixing four times. Examples of the answers are:

“This information can be useful when:-I need add a new feature, I can understand the related db actions of any existing method.-To fix a bug. To build a business process.”

“Understanding the code in general. I could also imagine that it is particularly helpful for debugging database-related errors (wrong updates, wrong implications drawn from the data) as well as performance problems due to unnecessary database queries.”

“Bug fixing (to find out useful information possibly related with the bug) and refactoring/remodularization (in order to make sure that modification will not invalidate some constraints)”

RQ₅ (Usefulness and Adoption for maintenance of real DCAs): Two practitioners from *LIMINAL Itda* analyzed the reports generated by DBScribe for two industrial DCAs. Both systems were developed using a multi-tier web architecture and MySQL as the database engine. The database schemas use

referential integrity. *System 1* uses JSP, CSS and Javascript for the presentation tier; a set of servlets operates as controllers between the web components and a tier of business entities that implement the logic and persistence operations. *System 2* uses Java Server Faces for the presentation tier; JSF beans and a JSF Front Controller are used as controllers between the GUI and application services (i.e., business tier); the application services invoke database operations by means of a persistence tier implemented with JDBC Data Access Objects; Value Objects are used to transfer data over all the tiers. *System 1* has been in production for about ten years, and *System 2* has been in production for over seven years. Both systems are currently maintained by *LIMINAL Ltda.*

Concerning the practitioners, their current positions are project managers, but they were also the original developers of both systems. They have ten years of industrial experience developing Java web applications. One of the practitioners asked us to anonymize his name; thus, we refer to him as Practitioner 1. The second practitioner is Néstor Romero, who also was the developer in charge of *System 2* maintenance for one year. We asked both practitioners to indicate the architectural layer(s) in which they are more proficient: Practitioner 1 responded “Business Layer, Data Access, and Utilities”, Néstor responded “GUI and Business Layer”.

Both practitioners ranked completeness, conciseness, and expressiveness for the two reports, giving them the highest values (i.e., does not miss any important info, contains no redundant info, is easy to read). Also, both agreed positively on the usefulness of the reports. For instance, Néstor noted “*Based on the descriptions you can be aware all dependencies a table could have. It would let you estimate in a better way the impact due to future changes.*”; “*The text*

provided helps to create a basic understanding of the functionality". With respect to the software engineering tasks, they would use the reports for program understanding, impact analysis, and technical documentation. In particular, they mentioned: *"It helps you create a quick vision of the system with the basic method and code structure without looking at actual source code"*; *"Creating a data - dictionary for a system"*.

Finally, both practitioners agreed on features for improving navigation features in the reports. For instance, Participant 1 claimed *"It would be useful to search a table name in order to see what dependencies it has"*, and Néstor claimed *"The link system for call-chains works only in one way, one could get lost navigating a complex system as there is no visual or textual reference of my location within the entire document. A navigation tree might be useful in this case."* Practitioner 1 augmented the response with some desirable features: *"you should extend the approach to include JPA"*, and *"it would be better to have it in the IDE, something like right click, then generate"*.

Summary of the results: *The DBScribe's descriptions are complete, concise and readable, in most of the cases. The participants consider the descriptions to be useful for understanding the DB usages in DCAs. Moreover, this type of descriptions is useful for understanding DB related source code. Concerning software engineering tasks, the participants consider that the summaries can be mostly useful for incremental change-related tasks, debugging, and bug fixing.*

2.5 Related Works on Documenting DCAs

Despite recent studies showing that there is a strong evolutionary coupling between database schema and source code [127, 74, 139, 102], as of today, no approach has been proposed to automatically document/comment database usages in source code. Some approaches have been proposed to extract information directly from the schema, but without considering the source code [125, 124, 30, 29, 90]. There is also some previous work for automated comment generation of software artifacts [45, 142, 110, 113, 50, 115, 123, 80, 48, 46, 111, 103]; however, none of the existing approaches focus on generating database-related documentation for supporting evolution and maintenance of source code.

2.5.1 Studies on Co-evolution of Schema and Code

Maule et al. [102] use program slicing and dataflow-based analysis to identify the impact of database schema changes. Qiu et al. [127] conducted an empirical study into co-evolution between database schemas and source code, and the authors demonstrated that database schemas frequently evolve with many different types of changes at play. Sjøberg [139] presents a technique for measuring the changes of database schemas and performed a study on health management systems over several years, where additions and deletions are found to be the most frequent operations. Recently, Li et al. [94] proposed a novel recommendation system to detect potential integrity violations in DCAs. *These studies and findings serve as our main motivation for developing DBScribe to help developers understand evolving DB usages and schema*

constraints.

2.5.2 Extracting Database Information

Several studies focus on extracting database-related information [125, 124]. The approach proposed by Petit et al. first extracts the database table names and attributes from the database schema [125]. Then, the approach builds semantic relations between the entities by investigating set operations and join operations. Alhaji et al. presented an algorithm to identify candidate and foreign keys of all relations from an existing legacy database [30].

Another group of studies focused on analyzing data in the database and extracting associative constraints [29, 90]. The associative rule mining problem was first introduced by Agrawal et al. [29], where the associative rule mining algorithm is able to generate a set of implications $A \rightarrow B$ based on the given relational table. Au et al. [90] applied a fuzzy association rule mining technique to a bank database system and identified some hidden patterns in the data. Li et al. [93] used association rule mining technique to correct the semantic errors in generated data. *These approaches represent an extension opportunity for DBScribe, since the relationships between DB objects can be inferred even when schemas lack referential integrity. We will extend DBScribe to automatically infer such relationships as part of the future work.*

2.5.3 On Documenting Software Artifacts

Buse and Weimer [45] present an approach for generating human-readable documentation of exceptions in Java. More specifically, they use a method call graph and symbolic execution techniques to extract the conditions of exceptions.

Then, they use a predefined template to generate natural language comments. Sridhara et al. present an approach for automatically generating summary comments for Java methods [142]. The authors demonstrate how to identify different kinds of important lines of code based on various characteristics of the code. Once these important lines are identified, the technique converts them into natural language phrases within their method bodies. Moreno et al. later extended the scope of the comment generation to class level granularity [110, 113]. Their description is based on superclass, stereotypes of the class, and behaviors of blocks. McBurney and McMillan [103] uses contextual information (i.e., most important method in the context of a target method) to generate method summaries that include natural language descriptions of how to use the method and the purpose of the method as a part of a codebase.

Differently from the previous work, a number of papers focused on summarizing differences between program versions [50, 97, 115, 123, 80, 48, 46, 111, 82]. Linares-Vásquez et al. implemented a tool, ChangeScribe, for automatically generating commit messages [50, 97]. They extracted changes between two adjacent versions of a project and identified involved change types in addition to performing commit level stereotype analysis. Moreno et al. [111] introduced an approach, ARENA, for automatic generation of release notes of Java systems. Jackson and Ladd [80] present a tool named SematicDiff, which uses a program analysis technique to summarize the semantic differences between two versions of a project. Canfora et al. [48] present Ldiff to find line-based differences between two versions. More specifically, Ldiff is able to track lines moved away from the original position by comparing all combinations of *diff* fragments. Buse and Weimer [46]

present an approach for generating human readable documentation for program differences based on symbolic execution. Automatic summarization techniques have also been applied to exceptions [45], bug reports [130], crash reports [108], developer discussions [121, 152, 157], loops [153] and code examples [161, 162]. *However, none of the existing approaches focus on generating DB-related descriptions; DBScribe is the first to analyze source code and DB schemas for generating method-level documentation to support DCA maintenance.*

2.6 Conclusion

In this chapter, we presented DBScribe a novel approach for generating automatically natural language summaries at source code method level that describe database usages and constraints for a given system. The summaries are generated by detecting methods executing local SQL queries/statements, and then propagating the schema constraints through all the methods that execute the queries/statements by means of delegation. DBScribe is motivated in a preliminary study in which we found that 77% of 33K+ methods in 3.6K+ open-source Java projects with database accesses were completely undocumented.

To validate DBScribe, we conducted a second study with 52 participants, in which we asked them (i) to rate the completeness, conciseness, and expressiveness of the summaries, and (ii) to describe the usefulness of the summaries. The results of the second study shows that DBScribe summaries are useful for understanding database usages and constraints across a given

system, and the summaries can be used for debugging/profiling/understanding database related source code. All in all, DBScribe provides solution to a challenging and common problem by relying on static analysis of source code and database schemas, and summarization techniques.

2.7 Bibliographical Notes

The papers supporting the content described in this Chapter were written in collaboration with the members of the SEMERU group at William and Mary:

- Linares-Vásquez, M., **Li, B.**, Vendome, C., and Poshyvanyk, D. “How do Developers Document Database Usages in Source Code?.” in Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 36-41. IEEE, 2015.
- Linares-Vásquez, M., **Li, B.**, Vendome, C., and Poshyvanyk, D. “Documenting database usages and schema constraints in database-centric applications.” in Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), pp. 270-281. ACM, 2016.
- **Li, B.** “Automatically Documenting Software Artifacts.” in Proceedings of 32nd International Conference on Software Maintenance and Evolution (ICSME), pp. 631-635. IEEE, 2016

Chapter 3

Documenting Unit Test Cases

During evolution and maintenance of software systems, 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) [41] has been employed by a myriad of developers and organizations to create and expand software systems [43, 42]. 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.

In a survey (Section 3.1) with 212 open-source and industrial developers, we found that 89.15% of the developers acknowledged that it is very important to maintain unit test cases. In particular, developers acknowledged that unit test cases benefit maintenance of legacy code, reduce the burden of understanding on new members of a project, and provide confidence in the quality of new code added to an existing system, among other reasons. Moreover, developers reported that they do not frequently update comments pertaining to unit test

cases despite the fact that they consider maintaining unit test cases an important task. While some developers suggested that comments were not necessary for unit tests, we observed that a majority of developers found understanding of unit test cases to be at least moderately difficult.

We also performed an empirical study on the change histories of 1,414 software systems to understand the prevalence of unit test case comments and whether developers update such comments between releases of the system. We found that approximately 3.56% of unit test cases had preceding comments and 14.02% of unit test cases had inner comments out of a total of 53,735 unit test cases. We observed that these comments rarely got updated during the development of these systems (1.54% of the unit test method changes for preceding comments and 15.23% of the unit test method changes for inner comments).

The results from the survey and the mining-based study highlight that (i) developers consider having up-to-date documentation and comments within source code regarding the unit test cases to be useful, but (ii) commenting unit test cases is not a widely used practice (in-the-wild). In order to effectively maintain test cases, it is important that developers understand the impact of each unit test case and the particular functionality that it aims to test. Prior studies demonstrated that developers seldom update comments in source code when they modify those regions of code to which the comments relate [67, 99, 163].

Consequently, in this chapter, we present an approach, called `UnitTestScribe`, to automatically generate natural language (NL) documentation of unit test cases. Our approach aims to ameliorate the burden of

maintaining unit test cases for developers and ideally help developers rapidly identify outdated unit test cases to avoid regressions in their systems. `UnitTestScribe` is a novel combination of static analysis, natural language processing, backward slicing, and code summarization techniques to generate descriptions at unit test method level. `UnitTestScribe` generates the descriptions by detecting focal methods [72], assertions, and data dependencies in unit test methods.

To validate the quality of the descriptions generated by `UnitTestScribe`, we conducted a study with both open source and industrial systems, and followed a widely used framework for evaluating automatically generated documentation [50, 110, 142]. We asked the participants (i) to evaluate the *completeness*, *conciseness*, and *expressiveness* of the generated descriptions, and (ii) to describe the usefulness of the description and the techniques.

This work is the first to investigate documentation practices of unit test cases. In addition, this chapter makes the following contributions:

- an empirical study to understand whether developers comment and update comments of unit test cases, which have been modified, from a large dataset of C# projects;
- a survey of both open-source and industrial developers to understand their perspective and practices with respect to documenting unit test cases;
- an approach for automatically documenting test cases that generates NL descriptions for unit test methods depicting focal methods, assertions, and data dependencies.

3.1 An Empirical Study on Documenting Unit Tests

To the best of our knowledge, no study has been performed to identify practices of developers when documenting test cases. Co-evolution of comments and source code have been investigated previously [66, 67, 99]; however, this work is the first to investigate evolution of comments and unit test cases. Hence, we performed a study aimed at identifying specific requirements for an approach to automatically document unit test methods. In particular, the preliminary study had two parts: i) an online survey with open-source C# developers and industrial practitioners (all of whom are Visual Studio users) and ii) a mining-based study that investigates the prevalence of comments in open-source C# systems from GitHub. The *goal* of this study was to determine the extent that developers write and update comments for unit test cases during evolution and maintenance of software systems. Additionally, we were interested in the answers from both open source and industrial developers with respect to documenting unit test cases. The *context* of the study 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.

3.1.1 Research Questions

We investigated the following research questions (RQs):

RQ₁ *To what extent do unit test cases contain comments?* This RQ aims to address the prevalence of both a preceding comment and inner comments

for the unit test cases.

RQ₂ *To what extent do developers update unit test case comments?* This RQ investigates how often developers modify and update the unit test case comments (both preceding and inner) during software evolution.

RQ₃ *To what extent, do developers have difficulty understanding unit test cases?* This RQ investigates whether there are obstacles in understanding unit tests cases and the need by developers for support in this task.

The RQs (**RQ₁-RQ₃**) were answered by combining the results from an online survey and a mining-based analysis. The rationale for this combined approach is that we aimed to gather answers directly from practitioners, and also to leverage empirical evidence from change histories of a large dataset of open source projects.

3.1.2 Data Collection

We identified all of the C# projects on GitHub through GitHub's public API [4]. We first extracted a comprehensive list of all hosted projects and extracted all of the projects identified as C#. We applied a filter to the projects to ensure the projects were not a fork and contained at least one star, watcher, or were forked. We avoided forks to prevent data duplication and we used the other three criteria as a way to remove abandoned projects. Our filtered dataset contained 2,209 projects that we locally cloned. We identified the developers of each project and sorted the unique email addresses that followed a regex format validation `^[a-zA-Z0-9_+-.]+@[a-zA-Z0-9-]+.[a-zA-Z0-9-]+$`, which sought to remove invalid email addresses, as well as remove email addresses with the patterns

@(none) and *@localhost*. We had 4,115 email addresses from open-source developers. Additionally, we contacted 565 industrial developers from ABB. The survey was distributed to the potential participants via email and the survey was hosted on Qualtrics [15].

The survey included three questions with demographic purposes (**D**₁ - **D**₃) and six questions (**Q**₁ - **Q**₆) that investigated whether developers document and maintain unit test cases. Table 3.1 lists **Q**₁ to **Q**₆. Respondents were also given an opportunity to describe their rationale in a free response field after each question. **Q**₁ and **Q**₂ relate to **RQ**₁; **Q**₃ and **Q**₄ relate to **RQ**₂; **Q**₅ and **Q**₆ relate to **RQ**₃ and serve to directly motivate our proposed approach for documenting unit test methods (Section 3.2). In addition to the survey, for **RQ**₁, we analyzed the latest snapshot of 1,414 projects randomly selected out of the 2,209 projects, and counted the number of unit test methods that were documented in source code; the random sampling is justified because of limitations on computation time.

Concerning the source code pre-processing, we split the inner and preceding comments in our analysis. We extracted the unit test methods with `srcML` [19] by identifying the annotations `[Test]`, `[TestMethod]`, `[TestCase]` - these annotations are used by NUnit [13] and Microsoft unit testing frameworks [12]. We also included some special annotations for other frameworks such as `[Fact]` and `[Theory]`. We automatically extracted the data and subsequently ran the analysis at release-level for the 246 projects with tagged releases and compared both the inner and preceding comments (as in **RQ**₁, we split this analysis) in order to determine the extent that developers are modifying comments when the unit test method is modified during the project's development (**RQ**₂).

Question/Answer
<p>Q₁. How often do you write unit test cases for your project(s)?</p> <p>Never: 13 (6.13%), Rarely: 23 (10.85%), Sometimes: 62 (29.24%), Fairly Often: 67 (31.60%), Always: 47 (22.17%)</p>
<p>Q₂. How often do you add/write documentation comments to unit test cases? (i.e., comments preceding the unit test method declaration)</p> <p>Never: 41 (19.34%), Rarely: 71 (33.49%), Sometimes: 38 (17.92%), Fairly Often: 39 (18.40%), Always: 23 (10.85%)</p>
<p>Q₃. How often do you find outdated comments (at method level) in unit test cases?</p> <p>Never: 37 (17.45%), Rarely: 64 (30.19%), Sometimes: 74 (34.90%), Fairly Often: 32 (15.09%), Always: 5 (2.36%)</p>
<p>Q₄. When you make changes to the unit tests, how often do you comment the changes (or update existing comments)?</p> <p>Never: 46 (21.70%), Rarely: 49 (23.11%), Sometimes: 48 (23.64%), Fairly Often: 37 (17.45%), Always: 32 (15.09%)</p>
<p>Q₅. Maintaining good unit test cases and documentations is important to the quality of a system.</p> <p>Strongly Disagree: 3 (1.41%), Disagree: 2 (0.94%), Neutral: 18 (8.49%), Agree: 89 (41.98%), Strongly Agree: 100 (47.17%)</p>
<p>Q₆. How difficult is it to understand a unit test? I.e., identifying focal methods under test (F-MUT) in unit test, where the F-MUTs are responsible for system state changes that are verified through assertions in the unit test.</p> <p>Very Easy: 22 (10.38%), Easy: 62 (29.24%), Moderate: 106 (50.00%), Hard: 17 (8.02%), Very Hard: 5 (2.36%)</p>

Table 3.1: Developer Survey Questions and Results.

3.1.3 Results

The survey questions and the results from 212 developers are summarized in Table 3.1 and the developer demographics information shows in Figure 3.1, Figure 3.2, and Figure 3.3.



Figure 3.1: Developer programming experience

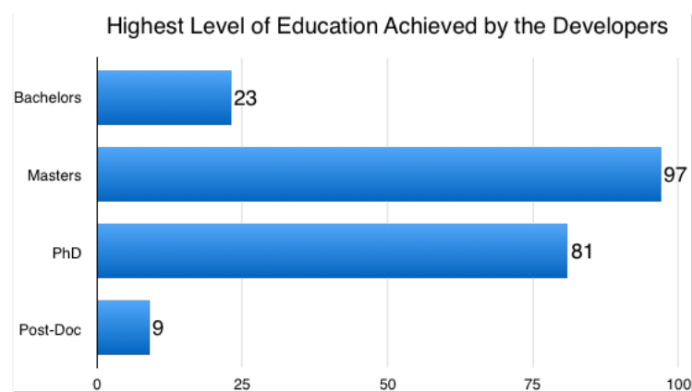
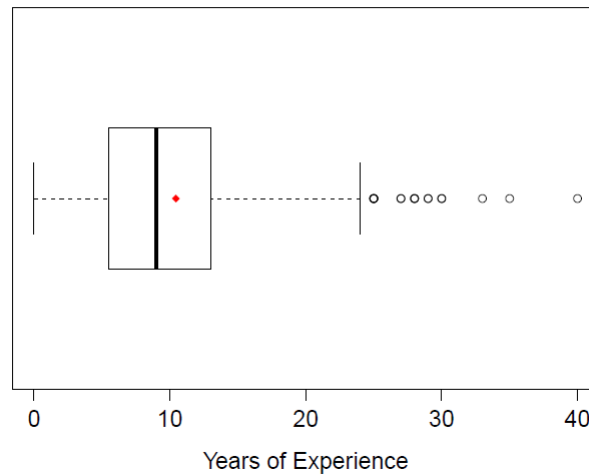


Figure 3.2: Highest level of education achieved by the developers

Developer Industry or Open Source Experience

**Figure 3.3:** Developer industry or open source experience

RQ₁: Our primary interest in answering **RQ₁** is to understand whether developers comment unit test methods - as preceding comments or as comments inside the method. To this end, we first asked **Q₁** to understand how often developers employ unit test cases in their systems. We observed that only 16.98% “rarely” or “never” write unit test cases, while 53.77% “fairly often” or “always” write unit test cases (or 83.01% of developers if we also consider “sometimes” respondents, since this does suggest a mid-level usage). Hence, more than half of the surveyed developers relatively frequently rely on unit tests. The following comments demonstrate their rationale:

“I wish I could do it Always, but most of the time my employer doesn’t want to pay the price of it, OR, the practice is not well-perceived by other team members, therefore abandoned. sadly.”

“When quality is required and time/budget allows”

“Always for commercial software. Only occasionally for personal projects.”

Q₃ demonstrates that developers are less prone to writing comments for the unit tests. We observe that 52.83% of developers “rarely” or “never” write comments and 17.92% “sometimes” write comments. This observation indicates that while a majority of developers utilize unit testing, typically developers are not writing comments for unit tests. Some of the rationale provided by the participants is:

“Comments need to be maintained which adds complexity to the task.”

“I use very verbose naming of tests to be the documentation, along with meaningful naming of methods and variables used in the test”

“I do, unless the test is really obvious”

Thus, we observed that in many cases developers find unit test cases to be simple enough for comprehension or try to encode the meaning in the naming. However, some developers also indicated documentation is necessary to understand the intent or importance of the unit test cases.

The mining-based study contradicts the developers’ preferences in that most of the projects do not have unit test methods and these methods are predominantly not documented with comments. In the analyzed source code (i.e., 1,414 projects), we identified that 395 projects (27.93%) had unit test classes and extracted a total of 53,735 unit test methods from these projects. In total, 51,821 unit test methods did not have outer comments (96.44%) and 46,201 did not have inner comments (85.98%). These results contradict our observations from Q₁ and Q₂, since we observe that the vast majority of unit test methods do not have preceding or inner comments. The contradictory results can be explained as an artifact of the sample analyzed in the mining-based

study, and the diversity of projects; it is worth noting that GitHub also hosts personal projects that might not require unit testing. However, the diverse set of projects in GitHub provide us with a general view of developer practices.

Summary for RQ₁. 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”).

RQ₂: In addition to the prevalence of comments for unit test methods, we were interested in whether developers update these comments or find outdated comments related to unit test methods. In terms of outdated comments, 47.64% of the developers indicated that they “rarely” or “never” find outdated comments in **Q₃**. This observation demonstrates that outdated comments are relatively common since 52.36% of developers find them at least “sometimes” to “always.” However, the results do indicate that only 17.45% of developers find the problem to be pervasive. The following are some offered explanations:

“Comments very quickly get out of sync. As code evolves the comments almost never get updated.”

“Because I don’t write comments for tests.”

These responses indicate that developers usually do not consider comments and assume them to be outdated. The latter assumption seems to be validated for database-related methods, since only 17% of these methods that were modified had preceding comments that were also updated at least once between releases

for a dataset including 3,113 systems [99]. Similarly, the developer feedback suggests that the “never” and “rarely” categories are over-represented in that developers do not find outdated comments because the code lacks comments.

Following these observations, answers to **Q₄** suggest that only 32.54% of developers frequently (“fairly often” and “always”) update comments when making changes to unit test cases. A plurality, 44.81%, either “rarely” or “never” update comments. These results somewhat contradict the former observation from **Q₃** in that more developers indicate that they do not update unit test comments than the developers that indicated finding outdated comments. It suggests that more of those comments are likely to be outdated than it may seem. For instance, we got the following rationale from the participants:

“I usually remove comments when I find them”

“There aren’t any since tests should be self documenting.”

“When I feel a need to comment on “why” I made the changes I prefer to add them as commit comments”

Interestingly, the developers indicated that many of the “updates” are the removal of comments. Additionally, developers indicated that such documentation of changes are logged in the commit messages. We also observe that the lack of comments impacts the results of **Q₄** (i.e., developers that do not comment unit test cases also will not update these non-existing comments). However, 32.54% of developers acknowledged that existing comments were updated frequently.

The mining study at release-level opposes the developer survey results in that it demonstrates unit test methods are not typically updated. For the 246 projects

with releases, we identified 101 projects that utilized unit test cases. From those 101 projects, we identified 1,075,076 unit test method changes from 3,160 total methods (aggregated numbers). In 16,561 of those test method changes, we observed the preceding comment was modified (1.54%), while 163,737 unit test method changes had inner comments that were modified (15.23%). These results contradict our observations from Q_3 and Q_4 , since we observe far fewer updates to unit test methods than expected from the developer survey.

Summary for RQ₂. 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.

RQ₃: Finally, we were interested in understanding whether developers had difficulty understanding unit test cases because of the comments, and the perceived importance of maintaining unit test cases. Overwhelmingly, developers indicated that maintaining unit test cases is important with 89.15% of developers responding “agree” or “strongly agree” to Q_5 . Thus, developers acknowledged that maintenance is important, but we also observe that understanding unit test cases is not trivial. 60.38% of developers indicated a “moderate” to “very hard” difficulty with respect to understanding. Thus, the unit test cases are important, but they are commonly not easy to understand. We found the following potential causes for this lack of understanding:

“This depends primarily on your level of immersion in the project, which if high makes understanding easier than if you are less immersed.”

“Depends on the complexity of the unit and the setup/fixtures required it can

be hard”

“It depends of how well you know the system and how the system is build”

While 39.62% of respondents indicated that they should be easy to understand, we observed that project familiarity and complexity of what is tested to be common causes of difficulty. 31.91% of the respondent providing rationale **Q₆** indicated unit test cases should be simple or should follow the “Arrange, Act, Assert” paradigm [24], which also aids in understandability.

Summary for RQ₃. 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.

3.1.4 Threats to Validity

The *construct* threat to validity relates to bias in our observations from the two perspectives of analysis (survey and mining of unit test cases). We do not offer rationale beyond the rationale provided by participants avoid inaccurate inferences. Additionally, the projects on GitHub may not contain the complete history of the projects due to the maturity differential of the sampled projects and GitHub. It is also possible developers did not tag all of the releases for the projects. However, these limitations are inherent to any mining study utilizing GitHub [84]. Threats to *internal* validity relate to response bias by developers that either had more difficulty or did not have problems while understanding or maintaining unit test cases. Based on the results of the survey, we observed that

responses were not dominantly distributed to extremes that indicates that these developers were particularly biased based on such difficulty. The *external* threats to validity relate to generalizing the conclusions from this work. In our conclusions, we state that these results are based on open-source developers from GitHub and industrial developers, but do not claim that these results generalize to all developers in other industrial companies, contributing to other forges, and developing systems in other languages. We do present demographic information in our online appendix [25] that suggests that we have a diverse sample of open-source and industrial C# developers.

3.2 UnitTestScribe: Documenting Unit Tests

Based on the findings from the study (Section 3.1), it is clearly important to have an approach to support developers in maintaining unit test case documentation. Therefore, we designed and implemented an approach, called `UnitTestScribe`, to support unit test cases documentation. `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

```
public AddWordsSeveralTimes(){ 1
    int listLength = 20; 2
    int cooccurrenceCount = 3; 3
    var words = GenerateRandomWordList(listLength); 4
    for(int i = 0; i < cooccurrenceCount; i ++){ 5
        matrix.HandleCoOcurrentWordsSync(words); 6
    } 7
    for(int i = 0; i < listLength - 1; i ++){ 8
        var word1 = words.ElementAt(i); 9
        var word2 = words.ElementAt(i + 1); 10
        var count = 11
            matrix.GetCoOccurrenceCount(word1,word2); 12
        Assert.IsTrue(count > 0); 13
    } 14
} 15
```

Figure 3.4: `CoOccurrenceMatrixTests.AddWordsSeveralTimes` unit test method of the Sando system

that are examined through assertions in unit tests [72]. We recognized focal methods as an important piece of information to be included in the resulting summary. In addition, results from our second study (Section 3.3) showed that identifying and highlighting focal methods would help developers better understand respective unit test cases (see Table 3.6).

Assertions are a key programming mechanism that is often used in unit test cases for comparing expected results to actual results after executing one (or more) method(s) from the software system under test. In addition, assertions are often related to focal methods in test methods. Therefore, the description of a focal method can be augmented with those assertions related to a focal method. Let us consider the `CoOccurrenceMatrixTests.AddWordsSeveralTimes` test method in the Sando [17] system (Fig. 3.4). The assertion in line 13 validates that the variable `count` is greater than zero after calling the focal method `matrix.getCoOccurrenceCount`. Thus, describing the focal methods in the test method and the assertions related to those methods by data dependencies might be useful for understanding the purpose of unit test methods.

Type	Category	Description	Modified rules
Getter	Accessor	Returns the value of a data member	No class field is changed && Return type is not void && Only return one class field.
Predicate	Accessor	Returns a Boolean result based on a data member(s)	No data member is changed && Return type is bool && Do not directly return any data member
Property	Accessor	Returns information about a data member	No data member is changed && Return type is not bool or not void. && Do not directly return any data member
Setter	Mutator	Changes the value of a data member	Only 1 data member is changed&&Return type is void or 0/1
Command	Mutator	Executes complex changes on data members	More than 1 class field is changed&&Return type is void or 0/1
Collaborator	Collaborator	Works on objects of classes different from the method	At least one of the method's parameters or local variables is an object Invokes external method(s)
Factory	Creator	Creates an object and returns it	Not returns primitive type Local && (A local variable is instantiated and returned Creates and returns a new object directly)

Table 3.2: Taxonomy of method stereotypes proposed by Dragan et al.[63] with our proposed modifications

```
public ExpandMoreLetters(){  
    var queries = expander.GetExpandedQueries("abfdsafafdc");  
    Assert.IsNotNull(queries);  
    queries = expander.GetExpandedQueries("bcfdasfdsad");  
    Assert.IsNotNull(queries);  
    queries = expander.GetExpandedQueries("defdasfdsaf");  
    Assert.IsNotNull(queries);  
}
```

Figure 3.5: AcronymExpanderTests.ExpandMoreLetters unit test method of the Sando system

The purpose of an assertion can be inferred and translated automatically into NL sentences by analyzing the assertion signature (e.g., `Assert.AreEqual` and `Assert.AreSame` methods in the C# API) and the arguments. For instance, the assertion `Assert.IsNotNull(queries)` in the `AcronymExpanderTests.ExpandMoreLetters` unit test method in the Sando system (Fig. 3.5) can be translated into “Validate that the queries are not null”. Additionally, arguments in focal methods and assertions have data dependencies with variables defined in the test method. These data dependencies can be described by slicing paths (analyzing data flows) ending at a focal method or an assertion call. Consequently, the descriptions generated by `UnitTestScribe` combine (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.

3.2.1 UnitTestScribe Architecture

The architecture of `UnitTestScribe` is depicted in Fig. 4.6. 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

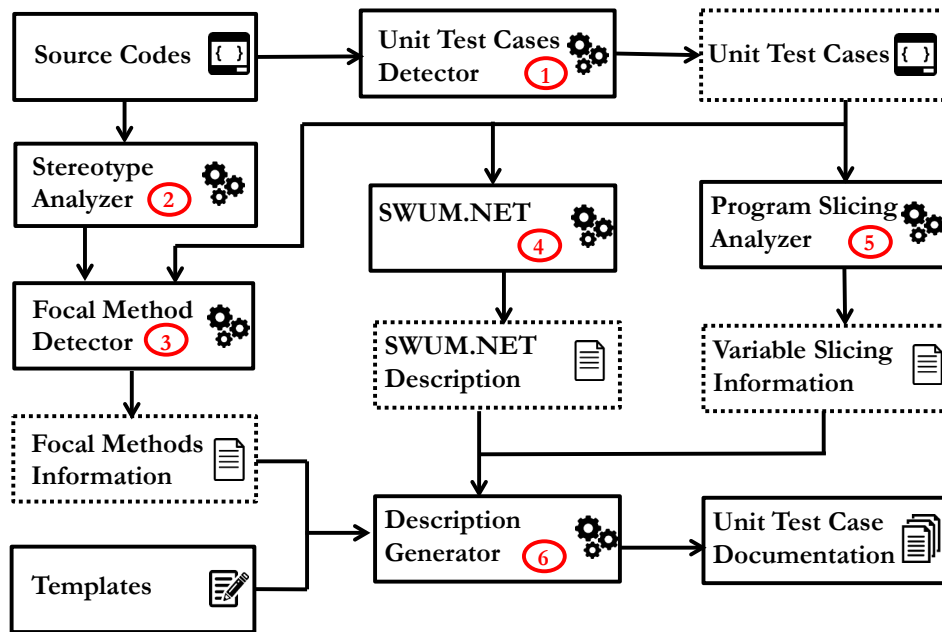


Figure 3.6: UnitTestScribe Architecture. The solid arrows denote the flow of data. Numbers denote the sequence of operations.

stereotypes at method level [63] 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 [21, 75] 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 [81] ⑤. Finally, the extracted information (focal methods, assertions, slices, and SWUM sentence) are structured in NL description by using predefined templates 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`.

3.2.2 Unit Test Detector

Our implementation focuses on systems that utilize NUnit [13] and Microsoft unit testing frameworks [12] for unit testing (because of the systems that were available for analysis and evaluation through our industrial collaboration). Unit test methods designed by developers are annotated with `[Test]` and `[TestMethod]` for NUnit and Microsoft testing frameworks respectively, which was utilized by our detection algorithm (we also include `[TestCase]`, `[Fact]`, and `[Theory]` for some special cases or new frameworks).

3.2.3 Method Stereotype Analyzer

Method stereotypes are labels/categories that indicate the intent and the role of a method in a class [63], e.g., *getter*, *setter*, *collaborator*. We modified the rules proposed by Dragan *et al.* [63] for C++ to have the corresponding stereotypes for C#. The Method Stereotype Analyzer in `UnitTestCrib` analyzes data flows provided by `SrcML.NET` [20], and then detects the stereotypes with the rules listed in Table 3.2. In order to collect all information for identifying method stereotypes for each method, we track all the changes to local variables and data members by examining statements that may cause a variable to change. We also analyze the call graph of a given project to record internal and external function calls for a given method. The main goal behind method stereotype analyzer is to accurately classify the method's intent, which is later used in the algorithm for identifying the focal methods.

Placeholder	Template	Example
<Part1>	This unit test case method is to <Action> <Theme> <Preposition> <SecondaryArg>	This unit test case method is to test class with declared variable.
<Part2>	This unit test case includes following focal methods: {<FocalMd>}	This unit test case includes following focal methods: ...
<Part3>	This unit test case validates that: {<Validatn>}	This unit test case validates that: ...
<FocalMd>	<Statement> This focal method is related with assertions at <LineNumber>	col.Add("black", "hole"); (@line 49) This focal method is related to assertions at line 50
<Validatn>	<AsrtDesc>. {<Variable> is obtained from variable <Variable> through slicing path <Path>}. {<Variable> >>> }	globalScope.IsGlobal is true. globalScope is obtained from variable xml through slicing path xml >>> globalScope.
<Path>	{<Variable> >>> }	xml >>> xmlElement >>> globalScope >>> actual

Table 3.3: A subset of placeholder templates with examples

Algorithm 1: An Algorithm for Focal Method Detection

```

Input: MethodDefinition m, AssertionStatement assert
Output: Set<FunctionCall> fmSet
1 begin
2   fmSet ← new Set<FunctionCall>()
3   v ← GetEvaluatedVariable (assert)
4   queue.Push(v)
5   while queue.Size > 0 do
6     v ← queue.Pop()
7     decl_stmt.v ← FindDeclaration (m, v)
8     b ← IsExternalObject (decl_stmt.v)
9     if b == true then
10      vSet ← GetRelatedVariables (m, v)
11      queue.PushAll(vSet)
12    else
13      call ← FindTheLastMutatorCall (m, v)
14      fmSet.Add(call)
15  return fmSet

```

3.2.4 Focal Method Detector

Because a test unit can have more than one assertion, we consider each call to an assert method as a testing sub-goal of the test method. Focal methods are responsible for application state changes that are verified through assertions in the unit test [72]. If there is a focal method associated with an assertion, then the focal method is the “core” of the corresponding testing sub-goal. `UnitTestScribe` identifies the focal methods by following the approach proposed by Ghafari et al. [72]. Unlike Ghafari et al.’s implementation, which only works with Java, our implementation works across the main modern object oriented programming languages, i.e., C#, Java, and C++, since we rely on a multi-language parsing tool, `srcML`, for generating XML files for source code and then analyzing them.

For each assertion, the Focal Method Detector in `UnitTestScribe` applies the following steps to find its focal methods; the procedure is listed in Algorithm 1. First, we identify the variables and literals used as arguments in the assertion call and distinguish the expected values from the actual values according to the

API documentation. For example, in the assertion statement `Assert.AreEqual(1, parts.Count)`, the value of `parts.Count` is the actual value and the integer literal `1` is the expected value. We push the variable of actual value to the analysis queue `queue` (line 3-4). Then, we check whether `queue` is empty since `queue` contains all the variables which potentially invoke focal methods (line 5). If `queue` has element(s), we pop up a variable, `v`, from `queue` (line 6). Next, we find the declaration statement `decl_stmt_v` of the assertion argument by using static backward slicing and analyze the type of `v` (line 7-8). If the type of `v` is an external class to the system (e.g libraries, build-in types), we then find a variable set `vSet` containing all of the variables that initialized `v` or are called by `v` as parameters (line 10); for each variable `v_new` in `vSet`, we push `v_new` to `queue` for further analysis (line 11). Otherwise, i.e., if the type of `v` belongs to the project code, `v` is marked as a focal variable for the current sub-goal and one of the focal methods for the current sub-scenario is defined to be the last mutator/collaborator function that the focal variable `v` calls before the assertion (line 13-14). The algorithm returns a set of detected focal methods when `queue` is empty (line 15).

3.2.5 General Description Extractor

Class/method/argument signatures usually contain verb phrases, noun phrases, and preposition phrases that are useful when constructing NL descriptions of code units [142, 76]. In addition, programmers do not arbitrarily select names and tend to choose descriptive and meaningful names for code units [96]. `UnitTestScribe` relies on the SWUM approach by Hill et al. [76], in particular the `SWUM.NET` tool implemented by ABB in C# [21], to extract natural language

Placeholder	Explanation
<code><Action></code>	Action phrase from SWUM.NET for the entity
<code><Theme></code>	Theme phrase from SWUM.NET for the entity
<code><Preposition></code>	Preposition from SWUM.NET for the entity
<code><SecondaryArg></code>	The second object phrase from SWUM.NET
<code><Statement></code>	A source code statement
<code><LineNumber></code>	An integer value indicating the line number
<code><AsrtDesc></code>	NL description for an assertion statement
<code><Variable></code>	A source code variable

Table 3.4: Leaf level placeholders

phrases that are used in composing general descriptions for unit test methods.

3.2.6 Slicing Path Analyzer

`UnitTestScribe` performs over-approximate analysis for each variable v in an assertion statement to compute all potential paths that may influence the value of v by using backward slicing [81]. Although `UnitTestScribe` does not track any branch conditions in the method (some paths may not be executed with a certain input), the over-approximate approach guarantees that potential slices are not missed in the description of the unit test case.

3.2.7 Description Generator

The Description Generator in `UnitTestScribe` uses the collected information from the previous steps and the predefined templates to generate NL descriptions for test methods. A description of a test unit method contains three parts:

- `<Part1>`: General sentence describing the purpose of a test method (based on class, method, and argument signatures) generated with SWUM.NET;

15. Sando.Core.UnitTests.Tools.CoOccurrenceMatrixTests.AddWordsSeveralTimes

Code with line numbers

■ This unit test case method tests add words several times .

▲ This unit test case includes following focal methods:
 (1) `var count = matrix.GetCoOccurrenceCount(word1,word2);(@line 47)`
 This focal method is related to assertions at [line 48](#)

▲ This unit test case validates that:
 (1) `count > 0` is true.
`count` is obtained from
 1) variable `listLength` through [slicing path](#)
 2) variable `i` through [slicing path](#)
 3) variable `listLength` through [slicing path](#)

`listLength(@line 36) >>> words(@line 38) >>> word2(@line 46) >>> count(@line 47)`

Figure 3.7: An example of UnitTestScribe Description for Sando’s method `CoOccurrenceMatrixTests.AddWordsSeveralTimes`

- ⟨Part2⟩: Descriptions of focal methods;
- ⟨Part3⟩: Description of assertions in the unit test method, including slicing paths of the variables validated with an assertion.

The templates are listed in Table 3.3 . The placeholders ⟨ . . . ⟩ in the templates mark tokens to be replaced (the placeholders are described in Table 3.4) by the Description Generator. We provide a complete list of templates, placeholders, and report examples in our online appendix [25]. A description for the method in Fig. 3.4 generated by UnitTestScribe is shown in Fig. 3.7. The ① marker indicates the general sentence describing the purpose of the test method; ② indicates the focal method of the unit test method; ③ highlights the assertions in the test method; and ④ indicates the variable’s slicing path when users hover over the hyper link.

3.3 UnitTestScribe: Empirical Study Design

We conducted a user study in which the descriptions generated by `UnitTestScribe` were evaluated by developers at ABB, computer science students, and researchers from different universities. The *goal* of this study was to measure the quality of `UnitTestScribe` descriptions as perceived by users according to a well-established framework for evaluating automatically generated documentation [50, 110, 142]. The *context* consisted of four C# open source software systems that use either NUnit or Microsoft unit testing frameworks, and 20 descriptions of unit test methods generated by `UnitTestScribe` (five methods for each system). The *perspective* was of researchers interested in evaluating the quality of a method for automated documentation generation. The *quality focus* was on the three attributes in the evaluation framework: completeness, conciseness, and expressiveness.

3.3.1 Data Collection

The list of analyzed systems included two open-source systems from ABB Corporate Research Center and two popular C# systems hosted on Github. Those subject applications are: 1) the `SrcML.NET` framework [20] used by ABB Corporate Research for program transformation and source code analysis; 2) the `Sando` [17] system developed by ABB Corporate Research, which is a Visual Studio Extension for searching C, C++, and C# projects; 3) `Glimpse` [5], which is an open-source diagnostics platform for inspecting web requests; and 4) the `Google-api-dotnet` library [6] for accessing Google services such as Drive, YouTube, Calendar in .NET applications.

We selected these four subject systems according to the following criteria: 1) the system should be a C# project and use either NUnit or Microsoft unit testing framework; 2) the system should be mature and under active maintenance. At the time that we selected the systems, Glimpse had 149 watches and 1,484 stars on Github, while `Google-api-dotnet` had 27 watches and 102 stars. Detailed information about the systems are shown in Table 3.5. Note that the lines of code for the test cases is in the range between 3 and 44 (average = 8.3, median = 7).

For the evaluation we ran `UnitTestScribe` on each subject system using an Intel Core i7-4700MQ CPU 2.4GHZ machine with 16GB RAM. We randomly selected five descriptions for each software system while covering the following criteria: 1) the selected method should have at least one assertion and 5 LOC (We define LOC as the lines of codes including method signature and brackets belong to the method in the unit test case file); 2) two descriptions must contain at most 4 assertions (simple cases); 3) three descriptions must have more than four assertions (complex cases). Our decision for including only five methods per system was based on the fact that analyzing the descriptions require inspection and navigation of the source code; on average it may take 4-5 minutes to investigate each test case and we had to restrict the study to 45 mins to avoid early-drop. After the study, we also randomly interviewed some participants to collect their opinions on limitations, usefulness, and suggestions for improvement.

We did not generate descriptions for test methods with less than five lines of code, since we assume developers should be able to quickly read those test cases and understand them without additional analysis. In other words, given the results of our empirical study, it was clear that developers prefer test case

documentation for more complex test cases. We computed the ratio of comments in test cases of our subject systems. We found that 28% of test cases with more than or equal to 5 LOC had comments, while only 13% of test cases with fewer than 5 LOC had comments. The observation suggests that larger unit test cases are commented more than smaller unit test cases, and unit test cases in our subject systems are rarely commented. Based on all the above, we claim that (i) developers need more help on complex test cases rather than simple ones; (ii) the test cases are rarely documented, which is consistent with our motivation study in Section 3.1.

3.3.2 Research Questions

The RQs aimed at evaluating the three quality attributes in the evaluation framework [50, 110, 142] (i.e., completeness, conciseness, and expressiveness); in addition, we evaluated whether focal methods are useful for describing the purpose of test methods, and whether the descriptions are useful for understanding test methods. Consequently, in the context of our study, we defined the following research questions:

RQ₄ *How **complete** are the unit test case descriptions generated by UnitTestScribe?*

RQ₅ *How **concise** are the unit test case descriptions generated by UnitTestScribe?*

RQ₆ *How **expressive** are the unit test case descriptions generated by UnitTestScribe?*

System	NF	MD	CLS	NS	TS	RT
SrcML.NET	332	2,867	306	42	410	546s
Sando	505	6,566	946	93	313	466s
Glimpse	909	6,503	1,045	153	943	1,281s
Google-api-dotnet	189	1,448	246	44	166	229s

Table 3.5: Subject systems: number of Files (NF), number of methods (MD), number of classes (CLS), number of namespaces (NS), number of test cases (TS), Running Time (RT).

RQ₇ *How important are focal methods and program slicing for understanding unit test cases?*

RQ₈ *How well can UnitTestScribe help developers understand unit test cases?*

3.3.3 Analysis Method

To answer the RQs, we organized the participants in two groups: developers/researchers from ABB, and academic researchers/students. The former group evaluated the descriptions generated by UnitTestScribe for SrcML.NET and Sando, and the latter group evaluated the descriptions for Glimpse and Google-api-dotnet. For each group, we created an on-line survey using the Qualtrics tool [15]. The survey included (i) demographic background questions, and (ii) questions aimed at answering the RQs (Table 3.6 lists the questions and possible answers). For each method, we also asked the participants to provide the rationale for their answers. We analyzed the collected results based on participants' choices on each question as well as free-text answers. For more detail, we analyzed the collected data based on the distributions of responses in diverse combinations (ABB vs. academic group,

simple methods vs. complex methods). We also checked the free-text responses in depth to understand the rationale behind the choices.

3.3.4 Threats to Validity

One threat to *internal* validity is that participants may not be familiar with the test case methods and subject systems. In order to reduce this threat, we let participants first understand each selected method and then answer questions about the method. Since we also provided source code for each system, participants could navigate the context related to the method. In addition, to avoid any type of bias, we did not tell the participants whether the documentation was automatically generated or not. One threat to *external* validity is that our current implementation only focuses on NUnit or Microsoft frameworks, however, `UnitTestScribe` can be easily extended to other testing frameworks. The other threat to *external* validity is that we only had limited number of methods in our user study. However, we selected a diverse set of methods to cover both simple and complex test cases. One more threat to *external* validity is that only C# unit tests and projects are analyzed in the study. However, since C# is a standard OOP language and we may consider that the results would be approximately the same with other standard OOP languages such as Java.

3.4 UnitTestScribe: Empirical Study Results

We collected 26 valid responses from the participants in two groups. In particular, the valid results contain responses from 7 developers/researchers from ABB (group 1) and 19 responses from students/researchers (group 2). It

Completeness: Only focusing on the content of the description without considering the way it has been presented, do you think the message is complete?	Group 1	Group 2
<ul style="list-style-type: none"> • The description does not miss any important information 	33(47.14%)	132(69.47%)
<ul style="list-style-type: none"> • The description misses some important information to understand the unit test case 	28(40.00%)	50(26.32%)
<ul style="list-style-type: none"> • The description misses the majority of the important information to understand the unit test case 	9(12.86%)	8(4.21%)
Conciseness: Only focusing on the content of the description without considering the way it has been presented, do you think the message is concise?	Group 1	Group 2
<ul style="list-style-type: none"> • The description contains no redundant information 	36(51.43%)	100(52.63%)
<ul style="list-style-type: none"> • The description contains some redundant information 	25(35.71%)	77(40.53%)
<ul style="list-style-type: none"> • The description contains a lot of redundant information 	9(12.86%)	13(6.84%)
Expressiveness: Only focusing on the content of the description without considering the completeness and conciseness, do you think the description is expressive?	Group 1	Group 2
<ul style="list-style-type: none"> • The description is easy to read and understand 	43(61.43%)	114(60.00%)
<ul style="list-style-type: none"> • The description is somewhat readable and understandable 	16(22.86%)	53(27.89%)
<ul style="list-style-type: none"> • The description is hard to read and understand 	11(15.71%)	23(12.11%)
Preferences: Identifying of focal methods would help developers to understand the unit test case	Group 1	Group 2
<ul style="list-style-type: none"> • Yes 	7(100%)	17(89%)
<ul style="list-style-type: none"> • No 	0(0%)	2(11%)
Preferences: Identifying of slicing path would help developers to understand the unit test case	Group 1	Group 2
<ul style="list-style-type: none"> • Yes 	6(86%)	13(68%)
<ul style="list-style-type: none"> • No 	1(14%)	6(32%)
Preferences: Are our generated description useful for understanding the unit test cases in the system?	Group 1	Group 2
<ul style="list-style-type: none"> • Yes 	4(57%)	17(89%)
<ul style="list-style-type: none"> • No 	3(43%)	2(11%)

Table 3.6: Study questions and answers.

should be noted that participants from group 1 were/are developers of the `Sando` and `SrcML.NET` projects. Therefore, we assume that participants in group 1 have better understanding on the unit test cases in the subject projects. Conversely, we consider participants in group 2 as newcomers since they did not have prior experience with those systems.

RQ₄ - **RQ₆** focus on three quality attributes: *completeness*, *conciseness*, and *expressiveness*. For completeness, we examined whether the descriptions of `UnitTestScribe` contain all important information (**RQ₄**). For conciseness, we evaluated whether the descriptions of `UnitTestScribe` contain redundant information (**RQ₅**). For expressiveness, the focus was whether the descriptions of `UnitTestScribe` are easy to read (**RQ₆**). Since we asked participants to evaluate these three attributes for five test case methods in each application, the total number of answers that we collected for each attribute by group 1 is $5 \times 2 \times 7 = 70$ answers, while the collected answers for each attribute by the group 2 is $5 \times 2 \times 19 = 190$ answers. In addition, we answered **RQ₇** and **RQ₈** based on the results shown in the preferences criteria in Table 3.6. Generated descriptions and anonymized study results from open-source developers are publicly available at our online appendix [25].

3.4.1 Demographic Background

The participants had on average 13.5 years (median = 15 years) of programming experience for group 1, and 7.1 years (median = 7) for group 2. When considering only industrial/open source experience, the participants in group 1 had on average 9 years (median = 5), and the participants in group 2 had on average 1.2 years (median = 0.5). Regarding the highest academic

degree achieved, group 1 had 4 participants with MS and 3 participants with PhDs, and group 2 had 8 participants with BS, 10 participants with MS, and 1 participant with PhD.

3.4.2 Completeness (RQ₄)

For group 1, 47.14% of the answers indicate that `UnitTestScribe` descriptions do not miss any important information, while only 12.86% of the answers indicate that the descriptions miss some important information to understand the unit test case. For group 2, 69.47% of the answers indicate that the descriptions do not miss any important information, while only 4.21% of the answers indicate that the descriptions miss important information. If we only focus on the first two options, we have 89% and 96% answers indicating that some or no important information is missing. More importantly, this demonstrates that only a very few answers indicated that some key information was missing.

We also observed that `UnitTestScribe` was evaluated more positively on complex methods rather than simple methods. For example, most of the answers (66.7%, 6 out of 9) with the lowest ratings by group 1 came from the first two methods in two systems (based on our study design, the first two methods in each system had fewer assertions and statements than the other methods). We also examined the comments with lower ratings. Participants' comments included the following: *"The main problem is that `DataAssert.StatementsAreEqual` is not recognized as an assert."* This comment is due to the fact that "`DataAssert.StatementsAreEqual`" was not included in any standard unit test framework assertions that we used for detecting. We mentioned this in Section 4.4.4.

Summary for RQ₄. Overall, the results suggest that `UnitTestScribe` is able to generate descriptions for test case methods that cover all essential information in most of the cases.

3.4.3 Conciseness (RQ₅)

For group 1, 51.43% of the answers indicate that `UnitTestScribe` descriptions contain no redundant/useless information, while only 12.86% of the answers indicate the description contain significant amount of redundant/useless information. For group 2, 52.63% of the answers indicate the descriptions contain no redundant/useless information, while only 6.84% of the answers indicates otherwise. Most of the responses with lower scores were from test case methods with the number of assertions greater than four (based on our study design, the last three methods in each system had more statements and assertions than the other two). For example, for the lowest rating in group 2, 84.6 % (11 out of 13) came from complex test case methods. One corresponding comment included the following: *“As the same variable is updated and used multiple times, this unit test description is very redundant.”* Our explanation is that the descriptions for larger test case methods may appear rather verbose, since we provided more descriptions for each assertion and slicing. The descriptions are trying to cover all important information that could also come at the expense of expressiveness. To overcome the redundancy, `UnitTestScribe` does not describe the assertions that are already described in the focal methods when the assertions include the focal methods.

Summary for RQ₅. Overall, the results support our claim that our designed templates for the `UnitTestScribe` generate descriptions with less redundant information.

3.4.4 Expressiveness (RQ₆)

For group 1, 61.43% of the answers indicate that `UnitTestScribe` descriptions were easy to read and understand, while only 15.71% of the answers indicated the descriptions were hard to read and understand. In group 2, we observed 60% of the answers indicating that `UnitTestScribe` descriptions were easy to read and understand, while only 12.11% of the answers indicated otherwise. 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. Hence, the conclusion is supported by the following comment from our participants: *“Again, I think that for long unit test methods, the description becomes difficult to read, perhaps summarizing the assertions for longer methods to give at a glance information.”*.

Summary for RQ₆. Overall, the results support that `UnitTestScribe` descriptions are easy to read and understand.

3.4.5 User Preferences (RQ₇ - RQ₈)

Seven participants (out of 7) in group 1 and 17 participants (out of 19) in group 2 answered that focal methods were important to understand test case methods. In case of usefulness of slices, 6 out of 7 answers in group 1, and 13 out of 19

answers in group 2 indicated that slices were useful for understanding the test case methods.

In the study, we also asked whether the generated descriptions are useful for understanding the unit test cases. For group 1, 4 out of 7 participants answered “Yes”, while 17 out of 19 participants also answered “Yes” in group 2. Based on the participants’ responses, we also suggest that the `UnitTestScribe` descriptions can be more useful for developers who are not familiar with the source/test code (89% of participants in group 2 agreed on that generated descriptions were useful for understanding the unit test cases). Participants’ comments with this rationale included the following: *“Once I see the SrcML.NET system, I know what’s going on. Its usefulness drops off if you’re talking to someone experienced with the code base, though. So I suppose this depends on who this is aimed at.”* from a participant in group 1 and *“It is useful if I am not familiar with an application.”* from a participant in group 2.

In addition, we collected following comments that illustrate some reasons why participants evaluated `UnitTestScribe` descriptions positively in usefulness:

“I saw these as being good from the perspective of trying to figure out if this method is of any real interest before investigating further to see what the method actually does. So if I were fixing a bug and wanted to know some quick information about this method, sure, I could see these as being helpful.”

“If I was quickly trying to understand what the code was doing on a high level, then I could delve into the source code with more understanding.”

“I think these types of descriptions would be really useful in understand unit tests for the purpose of writing/rewriting them for maintenance purposes as code evolves over time.”

Category	Subcategories
Bugs	Bug reporting(1), Bug detection(1)
Software maintenance	Program comprehension (7), Maintenance (4), Code reviews (1)
Testing	Test case changes (4), test case generation (3)
Others	Commenting (2), Learning a library (2)

Table 3.7: “What SE tasks would you use *UnitTestScribe* descriptions for?”

Summary for RQ₇ and RQ₈. Overall, participants agreed on that focal methods and program slicing for understanding unit test cases are important. *UnitTestScribe* is useful for understanding unit test methods.

3.4.6 Participants’ Feedback

In the interviews after the study, we also asked the participants to indicate for which SE tasks they would use *UnitTestScribe*. The answers and the categories are listed in Table 3.7. Participants also pointed out some limitations of our current implementation, which include the following:

“mock-style tests are not well described.”

“The description didn’t describe that the focal method or assertions are inside a loop or not”

“slicing path is showing only the name of the variables and not their types.”

We also collected suggestions from participants, which include the following:

“Providing more context of the method would be helpful”

“Unit test can contain API usage examples. Perhaps this approach can serve a purpose in showing relevant examples of how to use some API”

These are examples of very useful comments that we are planning on incorporating in our future work.

3.5 Related Works on Documenting Unit Tests

3.5.1 Approaches and studies on unit test cases

Kamimura and Murphy [85] 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 [159] 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. [126] presented an approach for automatically recommending test code examples when programmers make changes in the code. Panichella et al. [122] presented an approach for automatically generating test case summaries for JUnit test cases. Runeson [136] 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 [91, 92], prioritization [135, 56], automatic test case generation [69, 64, 52], test templates [164], data generation [101, 93]. However, none of the existing approaches focuses on generating unit test case documentation as NL summaries. Our approach, `UnitTestScribe`, **is the first to describe unit test cases** by combining different description granularities: *i*) general description in NL, and *ii*) detailed descriptions by highlighting focal methods and showing relevant program slices.

3.5.2 Studies on classifying stereotypes

A program entity (method or class) stereotype reflects a high level description of the role of the program entity [63, 61]. Dragan et al. [63] first conducted an in-depth study of stereotypes at method level. They presented a well-defined taxonomy of method stereotypes. Then, Dragan et al. [61] extended the stereotype classification to class level granularity. A class stereotype is computed based on method stereotypes in the class by considering frequency and distribution of the method stereotypes. Later, Dragan et al. [60] presented commit level stereotypes based on the types of the changing methods/classes in the commits. Moreno and Marcus [112] implemented a tool, `JStereoCode`, for automatically identifying method and class stereotypes in Java systems.

A group of techniques apply stereotype identification for other goals. Dragan et al. [62] showed that method stereotypes could be an indicator of a system's design. Moreno et al. [110, 113] utilized class stereotypes to summarize the responsibilities of classes. Linares-Vásquez et al. [50, 97] relied on commit stereotypes for generating commit messages. Abid et al. [28] presented an approach that automatically generates NL documentation summaries for C++ methods based on stereotypes. Overall, none of the existing approaches (but `UnitTestScribe`) apply stereotype identification for generating unit test case documentation.

3.6 Conclusion

We presented a novel approach `UnitTestScribe` 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. `UnitTestScribe` is motivated by a study in which we surveyed 212 developers to understand their perspective towards unit test cases. We found that developers believe that maintaining good unit test cases is important for the quality of a software system. We also mined changes of 1,414 open-source projects and found that 3.56% of unit test cases had preceding comments and 14.02% of those had inner comments and both were not frequently updated between the releases.

To validate `UnitTestScribe`, we conducted a second study with two groups of participants (the original developers on two industrial and graduate students on the other two open-source systems). In the study, we evaluated three quality attributes: completeness, conciseness, and expressiveness. The results of the second study showed that `UnitTestScribe` descriptions are useful for understanding test cases. In general, developers determined that our approach generated descriptions that did not miss important information (87% and 96%), did not contain redundant information (87% and 93%), and were both readable and understandable (84% and 88%).

3.7 Bibliographical Notes

The papers supporting the content described in this Chapter were written in collaboration with the members of the SEMERU group at William and Mary and researchers from the ABB Corporate Research Center:

- **Li, B.**, Vendome, C., Linares-Vásquez, M., Poshyvanyk, D. and Kraft, A. N. “Automatically Documenting Unit Test Cases.” in Proceedings of 9th IEEE

International Conference on Software Testing, Verification and Validation (ICST), pp. 341-352, IEEE, 2016.

- **Li, B.** “Automatically Documenting Software Artifacts.” in Proceedings of 32nd International Conference on Software Maintenance and Evolution (ICSME), pp. 631-635. IEEE, 2016

Chapter 4

Stereotype-based Tagging of Unit Test Cases

Unit testing is considered to be one of the most popular automated techniques to detect bugs in software, perform regression testing, and, in general, to write better code [78, 104]. In fact, unit testing is (i) the foundation for approaches such as Test First Development (TFD) [42] and Test-Driven Development (TDD) [41, 34], (ii) one of the required practices in agile methods such as XP [42], and (iii) has inspired other approaches such as Behavior-Driven Development (BDD) [116]. In general, unit testing requires writing “test code” by relying on APIs such as the XUnit family [9, 13, 44] or Mock-based APIs such as Mockito [11] and JMockit [7].

Besides the usage of specific APIs for testing purposes, unit test code includes calls to the system under test, underlying APIs (e.g., the Java API), and programming structures (e.g., loops and conditionals), similarly to production code (i.e., non-test code). Therefore, unit test code can also exhibit issues such as bad smells [40, 151, 119, 118, 148, 147], poor readability, and textual/syntactic characteristics that impact program understanding [137, 138]. In addition, despite the existence of tools for automatic generation of unit test

code [36, 69, 70, 71, 117, 133], automatically generated test cases (i.e., unit tests) are difficult to understand and maintain [122]. As a response to the aforementioned issues, several guidelines for writing and refactoring unit tests have been proposed [78, 104, 151].

To bridge this gap, this work proposes a novel automated catalog of stereotypes for methods in unit tests; the catalog was designed with the goal of improving the comprehension of unit tests and navigability of large test suites. The approach is a complementary technique to the existing approaches [122, 95, 85], which generate detailed summaries for each test method without considering method stereotypes at the test suite level.

While code stereotypes reflect high-level descriptions of the roles of a code unit (e.g., a class or a method) and have been defined before for production code [62, 28, 61], our catalog is first to capture unit test case specific stereotypes. Based on the catalog, this chapter also presents an approach, coined as `TeStereo`, for automatically tagging methods in unit tests according to the stereotypes to which they belong. `TeStereo` generates a browsable documentation for a test suite (e.g., an html-based report), which includes navigation features, source code, and the unit tests tags. `TeStereo` generates the stereotypes at unit test method level by identifying (i) any API call or references to the JUnit API (i.e., assertions, assumptions, fails, annotations), (ii) inter-procedural calls to the methods in the same unit test and external methods (i.e., internal methods or external APIs), and (iii) control/data-flows related to any method call.

To validate the accuracy and usefulness of test case stereotypes and `TeStereo`'s reports, we designed and conducted three experiments based on

231 Apache projects as well as 210 test case methods, which were selected from the Apache projects by using a sampling procedure aimed at getting a diverse set of methods in terms of size, number, and type of stereotypes detected in the methods (Section 4.3.2). In these projects, TeStereo detected an average of 1,577 unit test stereotypes per system, which had an average of 5.90 unit test methods per test class (total of 168,987 unit test methods from 28,644 unit test classes). When considering the total dataset, the prevalence of any single stereotype ranged from 482 to 67,474 instance of the stereotype. In addition, we surveyed 25 Apache developers regarding their impressions and feedback on TeStereo's reports. Our experimental results show that (i) TeStereo achieves very high precision and recall for detecting the proposed unit test stereotypes; (ii) the proposed stereotypes improve comprehension of unit test cases during maintenance tasks; and (iii) most of the developers agreed that stereotypes and reports are useful for test case comprehension.

In summary, this chapter makes the following contributions: (i) a catalog of 21 stereotypes for methods in unit tests that extensively consider the JUnit API, external/internal inter-procedure calls, and control/data-flows in unit test methods; (ii) a static analysis-based approach for identifying unit test stereotypes; (iii) an open source tool that implements the proposed approach and generates stereotype-based reports documenting test suites; and (iv) an extensive online appendix [22] that includes test-case related statistics of the analyzed Apache projects, the TeStereo reports of the 231 Apache projects, and the detailed data collected during the studies.

4.1 Unit Test Case Stereotypes

In this section, we provide some background on stereotypes and describe the catalog of stereotypes that we have designed for unit tests methods.

Code stereotypes reflect roles of program entities (e.g., a class or a method) in a system, and those roles can be used for maintenance tasks such as design recovery [38, 39], feature location [59, 58, 131, 57], program comprehension, and pattern/anti-pattern detection [63, 61, 31, 98]. Although detecting stereotypes is a task that can be done manually, it is prohibitively time-consuming in the case of a large software system [63, 61]. Therefore, automated approaches have been proposed to detect stereotypes for entities such as classes, methods, and commits [63, 61, 60]. However, while the previously proposed catalog of stereotypes were not designed to consider the structure and purpose of unit tests; Unit test cases are different than other artifacts, since unit tests are designed by following different principles and workflow than non-test code [53, 78, 104].

Consequently, we designed a catalog of 21 stereotypes for unit test methods (Table 4.1 and Table 4.2), under the hypothesis that stereotypes could help developers/testers to understand the responsibilities of unit tests within a test suite. Also, stereotypes may reflect a high-level description of the role of a unit test case. For instance, stereotypes such as *“Exception verifier”*, *“Iterative verifier”*, and *“Empty test”* are descriptive “tags” that can help developers to (i) identify the general purpose of the methods without exploring the source code, and (ii) navigate large test suites. Therefore, the stereotypes can be used as “tags” that annotate the unit test directly in the IDE, or in external documentation (e.g., an html report). The tags can be assist navigation/classification of test

Type	Description	Rules
Boolean verifier	Verifies boolean conditions	Contains assertTrue Contains assertFalse
Null verifier	Verifies whether objects are null	Contains assertNull Contains assertNotNull
Equality verifier	Verifies whether objects/variables are equal to an expected value	Contains assertEquals Contains assertNotEquals
Identity verifier	Verifies whether two objects/variables refer to the same object/variable	Contains assertEquals Contains assertNotSame
Utility verifier	Verifies (un)successful execution of the test case by reporting explicitly a failure	Contains fail
Exception verifier	Verifies that exceptions are thrown during the test case execution	Has Expected attribute with the value class Exception or classes inherited from Exception
Condition Matcher	Verifies logic rules using matcher-style statements	Contains assertThat
Assumption setter	Sets implicit assumptions	Contains assumeThat Contains assumeTrue
Test initializer	Allocates resources before the execution of the test cases	Has annotation @Before Has annotation @BeforeClass
Test cleaner	Releases resources used by the test cases	Has annotation @After Has annotation @AfterClass
Logger	Invokes logging operations	Calls functions in PrintStream class Calls functions in Logger class
Ignored method	Is not executed with the test suite	Has annotation @Ignore
Hybrid verifier	Contains more than one JUnit-based stereotype	Number of matched JUnit-based stereotype > 1
Unclassified	Is not categorized by any of the available tags	Number of matched JUnit-based stereotype == 0

Table 4.1: JUnit API-Based Stereotypes for Methods in Unit Test Cases.

Type	Description	Rules
Branch verifier	Verifies assertions inside branch conditions	Number of assertions within branch conditions > 0
Iterative verifier	Verifies assertions in iterations	Number of assertions within iterations > 0
Public field verifier	Verifies values related to public fields	Actual values in assertions are from public field accesses
API utility verifier	Verifies values of variables related to API calls (External libraries)	Actual values in assertions are values of objects/variables related to API calls
Internal call verifier	Verifies values of variables related to AUT calls	Actual values in assertions are values of objects/variables related to AUT calls
Execution tester	Executes/invoke methods but assertions are verified	Number of assertions == 0 && number of function calls > 0
Empty tester	Is an empty test case	Number of lines of codes in method body == 0

Table 4.2: C/D-Flow Based Stereotypes for Methods in Unit Test Cases.

methods in large test suites. For example, it is time-consuming to manually identify test initializers that are also verifiers in a project like `Openejb` with 317 test classes and 1641 test method tags.

Note that the catalog we propose in this chapter focuses on unit tests relying on the JUnit API; we based this decision on the fact that in a sample of 381,161 open source systems from GitHub that we analyzed — by relying on a mining-based study—, only 134 of the systems used a mock-style-only APIs while 8,556 systems used JUnit-only APIs.

The full list of stereotypes are described with their explanations in the following subsections and in Table 4.1 and Table 4.2, where we list the stereotypes, a brief description, and the rules used for their detection. The stereotypes were defined by considering how values or objects are verified in a unit test case, the responsibilities of the test case, and the data/control-flows in the unit test case. Therefore, we categorized the stereotypes in two categories that reflect the usage of the JUnit API, and the data/control-flows in the methods. Note that the categories and stereotypes are not mutually exclusive, because our goal is to provide developers/testers with a mechanism to navigate large test suites or identify unit test methods with multiple purposes. For example, the method in Figure 4.1 is an *“Empty tester”* and *“Test Cleaner”*; assuming that the methods are annotated (in some way) with the tags (i.e., stereotypes), developers/testers can locate all unimplemented methods in the test suite (i.e., the empty testers), which will also be executed the last during the test unit execution (i.e., the test cleaners). Another example of potential usage of the tags, is detecting strange or potentially smelly methods, such as the *“Test initializer”* (i.e., a method with the `@Before` annotation) method depicted in Figure 4.2, which has other tags such

```
@After public void tearDown() throws Exception { }
```

Figure 4.1: *Test Cleaner* and *Empty Tester* method from `SessionTrackerCheckTest` unit test in Zookeeper.

```
@Before @Override public void setUp() throws Exception {
    super.setUp();
    this.tomcat=getTomcatInstance();
    this.context=this.tomcat.addContext("/weaving",WEBAPP_DOC_BASE);
    this.tomcat.start();
    ClassLoader loader=this.context.getLoader().getClassLoader();
    assertNotNull("The class loader should not be null.",loader);
    assertEquals("The class loader is not
        correct.",WebappClassLoader.class,loader.getClass());
    this.loader=(WebappClassLoader)loader;}
}
```

Figure 4.2: *Test initializer* method (from `TestWebappClassLoaderWeaving` unit test in Tomcat) with other stereotypes detected by TeStereo.

as “*Internal Call Verifier*”, and “*Null Verifier*”; we think this is a smelly methods because test initializer are not supposed to have assertions.

4.1.1 JUnit API-based Stereotypes

Assertions in the JUnit API have well-defined semantics that can be used to automatically infer or document the purpose of a test case [95, 122]. However, besides assertions for validating logical conditions between expected and real results (e.g., `assertEquals(int,int)`), JUnit provides other APIs for defining assumptions, expected exceptions, matching conditions, explicit declaration of failures, fixture setters, and cleaners, which have not been considered in prior work in automatic generation of documentation. Our catalog includes stereotypes for each one of those cases, because those APIs can reflect different purposes and responsibilities of the methods using the unit testing APIs.

The stereotypes in this category are detected by (i) building an Abstract

```

@Test public void existingConfigurationReturned(){           1
    Configuration conf=new Configuration(false);             2
    conf.set("foo","bar");                                   3
    Configuration conf2=CredentialProviderFactoryShim        4
        .getConfiguration(conf,"jceks:///file/accumulo.jceks"); 5
    Assert.assertSame(conf,conf2);                           6
    Assert.assertEquals("bar",conf.get("foo"));              7

```

Figure 4.3: Source code of the `existingConfigurationReturned` unit test method in the `Apache-accumulo`.

Syntax Tree (AST) for each method, (ii) looking for invocations to methods and annotations with the same signature from the JUnit API, and (iii) using the set of rules listed in Table 4.1. For instance, Figure 4.3 is an example of *Identity verifier* and *Equality verifier*. The difference between those two stereotypes is that the former focuses on testing whether two objects are the same reference, while the latter focuses on verifying that the objects are the same (by using the `equals` method). In Figure 4.3, the assertion (in line 6) is an identity assert, since the function call `assertSame` asserts that `conf2` should be the same object reference as `conf` (indicated as *Identity verifier*). In line 7, `assertEquals` asserts that the returned string, by calling `conf.get("foo")`, is equal to "bar" (indicated as *Equality verifier*).

In addition to the API-based stereotypes, we also defined two stereotypes for cases in which a unit test case contains more than one JUnit-based stereotype (i.e., *Hybrid verifier*), and cases where `TeStereo` was not able to detect any of the stereotypes (i.e., *Unclassified*). Because of space limitations, we do not show examples for all the stereotypes; however, more examples can be found in our online appendix [22].

```

@Test public void testConstructorMixedStyle(){
    Path p = new Path(project, "\\a;\\b:/c");
    String[] l = p.list();
    assertEquals("three items, mixed style", 3, l.length);
    if (isUnixStyle) {
        assertEquals("/a", l[0]);
        assertEquals("/b", l[1]);
        assertEquals("/c", l[2]);
    } else if (isNetWare) {
        assertEquals("\\a", l[0]);
        assertEquals("\\b", l[1]);
        assertEquals("\\c", l[2]);
    } else { ... }
}

```

Figure 4.4: Source code of the `testConstructorMixedStyle` unit test method in the Apache-ant system.

4.1.2 Data-/Control-flow Based Stereotypes

The API-based stereotypes (Section 4.1.1) describe the purpose of the API invocations; however, those stereotypes neither describe how the unit test cases use the APIs nor from where the examined data (i.e., arguments to the API methods) originates. Therefore, we extended the list of stereotypes with a second category based on data/control-flow analyses, because these analyses can capture the information missing in API-based stereotypes.

Using control-flow information, we defined two stereotypes for reporting whether the JUnit API methods are invoked inside a loop (i.e., an *Iterative Verifier*) or inside conditional branches (i.e., a *Branch Verifier*). For example, the unit test case in Figure 4.4 is a *Branch Verifier*, which verifies that the constructor of class `Path` is able to handle mixed system path styles (i.e., Unix, NetWare, etc.).

Using data-flow information, we defined stereotypes that describe when the arguments to the JUnit API calls are from (i) accesses to public fields (*Public Field Verifier*), (ii) API calls different to JUnit (*API Utility Verifier*), or (iii) calls to

```

@Test public void testRead() throws Exception {
    testConfigure();
    Locator locator=new Locator("evar1",_pid,_iid);
    Value value=new Value(locator,_el1,null);
    value=_engine.writeValue(_varType,value);
    Value readVal=_engine.readValue(_varType,value.locator);
    assertEquals(_iid,readVal.locator.iid);
    assertEquals(_pid,readVal.locator.pid);
    assertEquals(2,DOMUtils.countKids((Element)
    readVal.locator.reference,Node.ELEMENT_NODE));
}

```

Figure 4.5: Source code of the `testRead` unit test method in the `ode` system. the application under test (AUT) (*Internal Call Verifier*). For example, the unit test case in Figure 4.5 is a *Public Field Verifier*, which verifies the attributes in `readVal.locator` have the expected values. The data flow is from line 4 where the `value` object is created, to line 6 where the public field `value.locator` is accessed and used as an argument for a method invocation that is assigned to `readVal`. There is also a stereotype for methods in unit tests that do not verify assertions but invoke internal or external methods (*Execution Tester*). Finally, we included a stereotype that describes empty methods (*Empty tester*); developers/testers can use this type of stereotype to easily locate unimplemented methods in the test suite. Our online appendix [22] contains more examples of each stereotype.

4.2 Documenting Unit Test Cases with TeStereo

TeStereo is an approach for automatically documenting unit test suites that (i) tags methods in test cases by using the stereotypes and rules defined in Section 4.1, and (ii) builds an html-formatted report that includes the tags, source code, and navigation features, such as filters and navigation trees. In addition, for each method in a unit test, TeStereo generates a summary based

on the descriptions of each stereotype. TeStereo is a novel approach that combines static analysis and code summarization techniques in order to automatically generate tags and natural language-based descriptions aiming at concisely documenting the purpose of test cases.

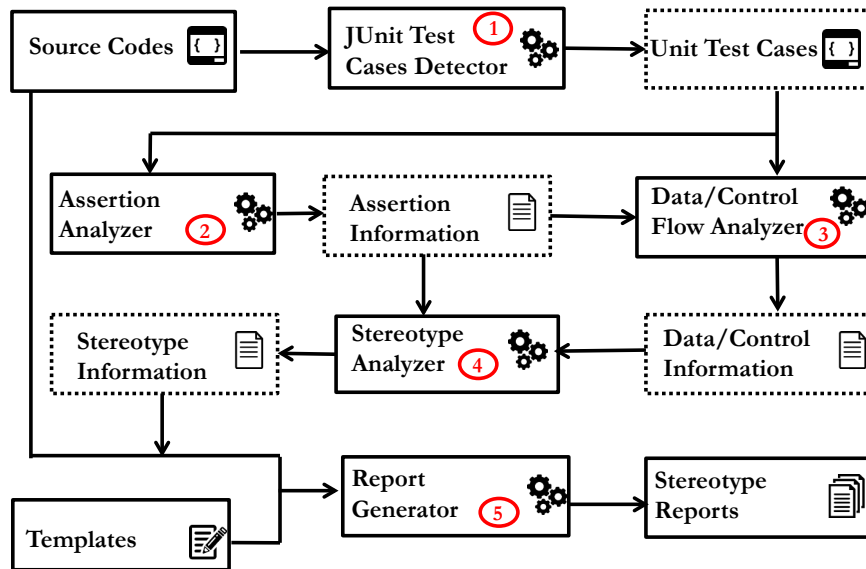


Figure 4.6: TeStereo Architecture. The solid arrows denote the flow of data. Numbers denote the sequence of operations.

The architecture of TeStereo is depicted in Fig. 4.6. TeStereo can be summarized in the following workflow:

1. Test case detection. The starting point of TeStereo is the source code of the system (including the test cases). TeStereo first analyzes the source code to identify all the unit test cases by detecting the methods in the source code that are annotated with `@Test`, `@Before`, `@BeforeClass`, `@After`, `@Afterclass` and `@Ignore`;

2. JUnit API call detection. The source code methods identified as test cases are then analyzed statically by scanning and detecting invocations to annotations and methods from the JUnit API;

3. Data/Control-flow analyses. Data-flow dependencies between the JUnit API calls and the variables defined in the analyzed method are identified by performing static backward slicing [81]; in addition, the collected references to the API calls are augmented with boolean flags reporting whether the calls are made inside loops or conditional branches. `TeStereo` performs a lightweight over-approximate analysis for each argument v in an JUnit API call to compute all potential paths (including internal function calls, Java API calls, and public field accesses) that may influence the value of v by using backward slicing [81]. Although `TeStereo` does not track any branch conditions in the unit test case (some paths may not be executed with certain inputs), the over-approximation guarantees that potential slices are not missed in the backward slicing relationships;

4. Stereotype detection. `TeStereo` uses the data collected in the previous steps, and then applies the rules listed in Table 4.1 and Table 4.2 to classify the unit tests into defined stereotype categories;

5. Report generation. Finally, each method is documented (as in Figure 4.10) and all the method level documents are organized in an html-based report. We encourage an interested reader to see the reports generated for 231 Apache projects in our online appendix [22].

4.3 Empirical Study

We conducted an empirical study aimed at (i) validating the accuracy of `TeStereo`-generated test case stereotypes, and (ii) the usefulness of the stereotypes and the reports for supporting evolution and maintenance of unit

tests. We relied on CS students, researchers, and the original developers of Apache projects to perform the study. In particular, the *context* of the study encompasses 210 methods randomly selected from unit tests in 231 Apache projects, 231 TeStereo reports, 420 manually generated summaries, 25 Apache developers, and 46 students and researchers. The *perspective* is of researchers interested in techniques and tools for improving program comprehension and automatic documentation of unit tests.

4.3.1 Research Questions

In the context of our study, we investigated the following three research questions (RQs):

RQ₁: *What is TeStereo's accuracy for identifying unit test stereotypes?* Before using the stereotypes in experiments with students, researchers, and practitioners, we wanted to measure TeStereo's accuracy in terms of precision and recall. The rules used for stereotype identification are based on static detection of API calls and data/control flow analyses. Therefore, with **RQ₁**, we aim at identifying whether TeStereo generates false positives or false negatives and the reasons behind them.

RQ₂: *Do the proposed stereotypes improve comprehension of tests cases (i.e., methods in test units)?* The main goal of method stereotypes is to describe the general purposes of the test cases in a unit test. Our hypothesis is that the proposed stereotypes should help developers in evolution and maintenance tasks that require program comprehension of unit tests. **RQ₂** aims at validating the hypothesis, in particular, when using the task of

manually generating summaries/descriptions for methods in unit tests (with and without stereotypes) as a reference.

RQ₃: *What are the developers' perspectives of the TeStereo-based reports for systems in which they contributed?* TeStereo not only identifies test stereotypes at method level, but also generates html reports (i.e., documentation) that includes source code, stereotypes, short stereotype-based summaries, and navigation features. Thus, **RQ₃** aims at validating with practitioners (i) if the stereotypes and reports are useful for software-related tasks, (ii) what features in the reports are the most useful, and (iii) what improvements should be done to the reports if any.

The three RQs are complementary for TeStereo's evaluation. **RQ₁** focuses on the quality of stereotype identification; we asked graduate CS students from a research university to manually identify the stereotypes on a sample of methods from unit tests; then, we computed micro and macro precision and recall metrics [140] between the gold-set generated by the students and the stereotypes identified by TeStereo on the same sample. With **RQ₁**, we also manually checked the cases in which TeStereo was not able to correctly identify the stereotypes, and then improved our implementation. **RQ₂** focuses on the usefulness of method stereotypes in unit tests; thus, we first asked students and researchers to write summaries of the methods (when reading the code with and without stereotypes). Then, giving the source code and manually written summaries, we asked another group of students and researchers to evaluate the summaries in terms of completeness, conciseness, and expressiveness [110, 50, 142, 95]. Note that there is no overlap among the participants assigned

to RQ_1 and RQ_2 . Finally, RQ_3 focuses on the usefulness of stereotypes and reports from the practitioners' perspective.

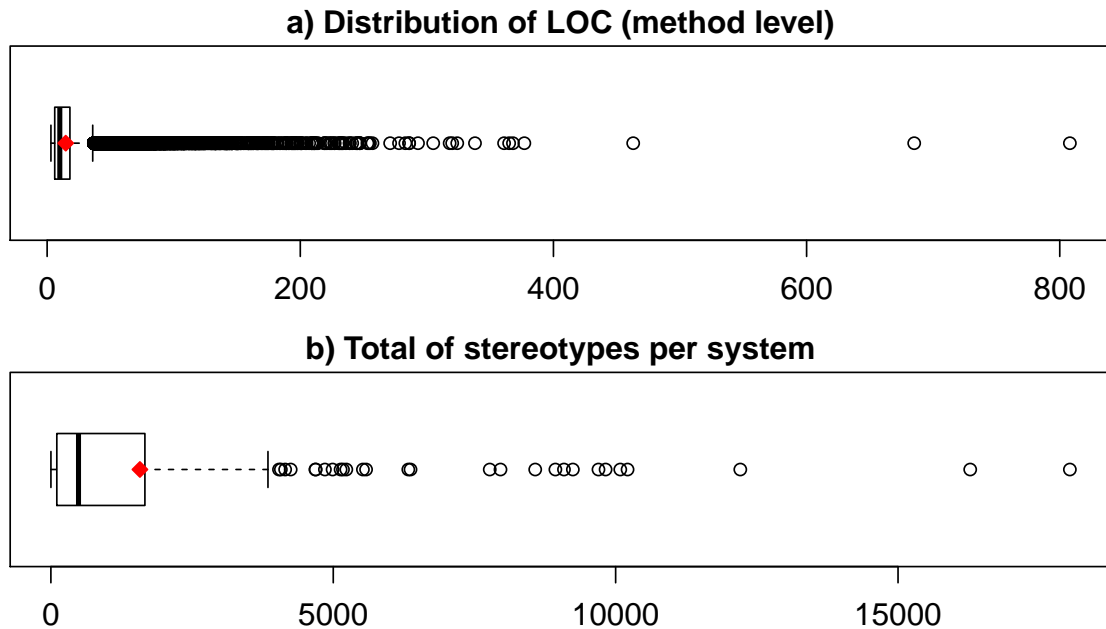


Figure 4.7: Diversity of the 261 Apache projects used in the study. The figure includes: a) size of methods in unit tests; b) distribution of method stereotypes per system; c) histogram of method stereotypes identified by TeStereo; and d) histogram of number of methods organized by the number of stereotypes detected on individual methods.

4.3.2 Context Selection

For the three RQs, we used the population of unit tests included in 231 Apache projects with source code available at GitHub. The list of projects is provided in our online appendix [22]. Our preference for Apache projects is motivated by the fact that they have been widely used in previous studies performed by the research community [107, 37, 132], and unit tests in these projects are highly diverse in terms of method stereotypes, methods size (i.e., LOC), and the number

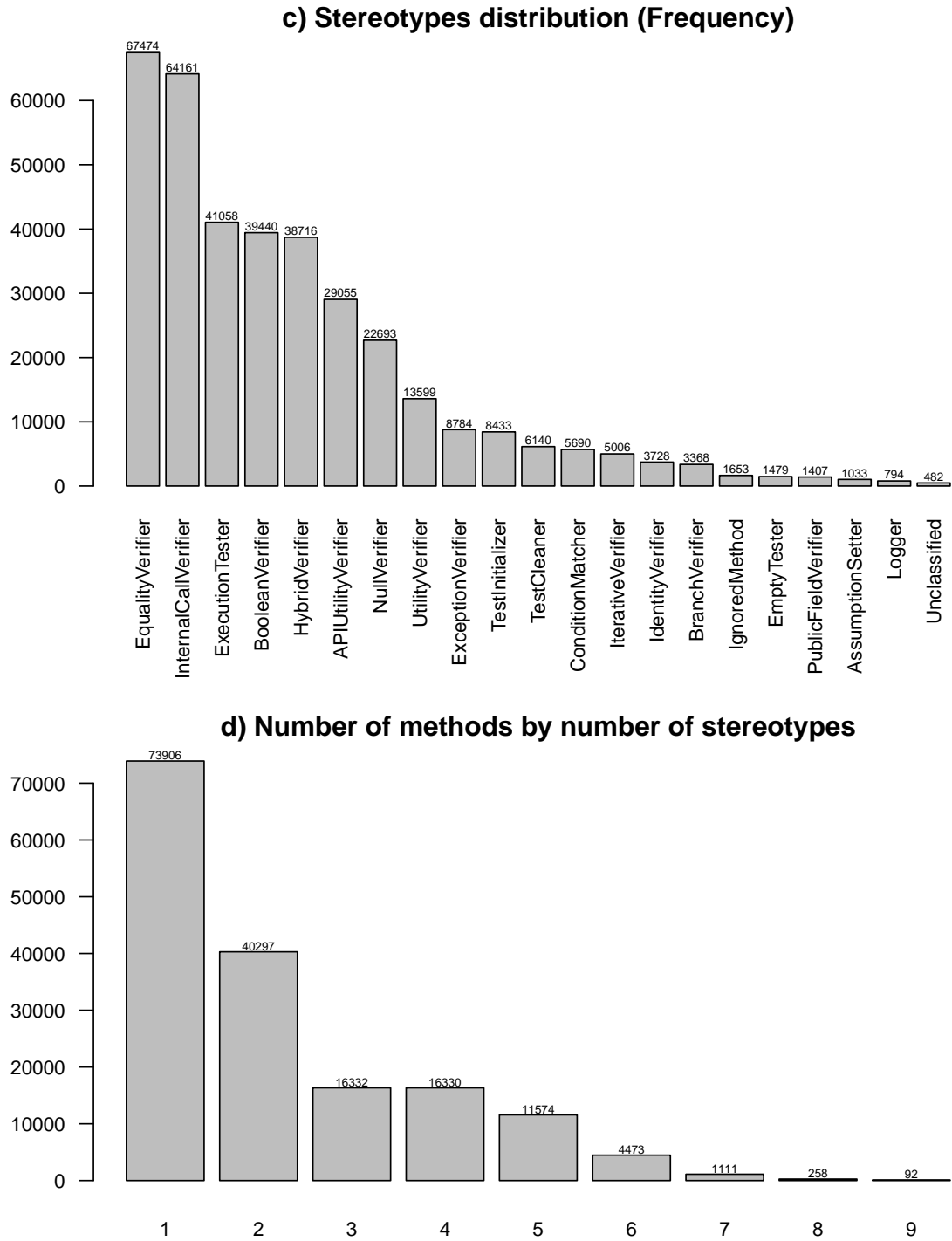


Figure 4.8: Diversity of the 261 Apache projects used in the study. The figure includes: c) histogram of method stereotypes identified by TeStereo; and d) histogram of number of methods organized by the number of stereotypes detected on individual methods.

of stereotypes. In the 231 projects, we detected a total of 27,923 unit tests, which account for 164,373 methods. Figures describing the diversity of the unit tests in 231 projects are in our online appendix [22]. On average, the methods have 14.67 LOC (median=10), the first quartile Q_1 is 6 LOC, and the third quartile Q_3 is 18 LOC. Concerning the number of stereotypes per system, on average, TeStereo identified 1,577 stereotypes in the unit tests (median=489). All 231 Apache projects exhibited at least 482 instances of each stereotype in the unit test methods, having *EqualityVerifier* as the most frequent method stereotype (64,474 instances). Finally, most of the methods (i.e., 73,906) have only one stereotype; however, there are cases with more than one stereotype, having a limit of 92 methods with 9 stereotypes each. In summary, the sample of Apache projects is diverse in terms of size of methods in the unit tests and the identified stereotypes (all 21 stereotypes were widely identified). Hereinafter, we will refer to the set of all the unit tests in 231 Apache projects as UT_{Apache} .

Because of the large set of unit test methods in UT_{Apache} (i.e., 164,373 methods), we sampled a smaller set of methods that could be evaluated during our experiments; we call this set M_{sample} , which is composed of 210 methods systematically sampled from the methods in UT_{Apache} . The reason for choosing 210 methods is that we wanted to have in the sample at least 10 methods representative of each stereotype (21 stereotypes \times 10 methods = 210). Subsequently, given the target size for the sample, we designed a systematic sampling process looking for diversity in terms of not only stereotypes and the number of stereotypes per method but also selecting methods with a “representative” size (by “representative” we mean that the size is defined by the 50% of the original population). Therefore, we selected methods with LOC

between $Q1 = 6$ and $Q3 = 18$. Consequently, after selecting only the methods with $LOC \in [Q1, Q3]$, we sampled them in buckets indexed by the stereotype ($B_{\langle stereotype \rangle}$), and buckets indexed by the number of stereotypes identified in the methods and the stereotypes ($B_{\langle n, stereotype \rangle}$); for instance, $B_{\langle NullVerifier \rangle}$ is the set of methods with the stereotype *NullVerifier*, and the set $B_{\langle 2, Logger \rangle}$ has all the methods with two stereotypes and one of the stereotypes is *Logger*. Note that a method may appear in different buckets $B_{\langle n, stereotype \rangle}$ for a given n , because a method can exhibit one or more stereotypes. We also built a second group of buckets indexed by stereotype ($B_{\langle stereotype \rangle}^{(2)}$), but with the methods with LOC in $(Q3, 30]$.

The complete procedure for generating M_{Sample} from the buckets $B_{\langle stereotype \rangle}$, $B_{\langle stereotype \rangle}^{(2)}$, and $B_{\langle n, stereotype \rangle}$ is depicted in Algorithm 1. The first part of the Algorithm (i.e., lines 5 to 10) is to assure that M_{Sample} has a least one method for each combination $\langle n, stereotype \rangle$; then, the second part (i.e., lines 11 to 25) is to balance the selection across different methods exhibiting all the stereotypes. Note that we use a work list to assure sampling without replacement. When we were not able to find methods in $B_{\langle stereotype \rangle}$, we sampled the methods from $B_{\langle stereotype \rangle}^{(2)}$. To verify the diversity of the M_{Sample} we computed the same statistics in Fig. 4.7 and Fig. 4.8.

Regarding the human subjects involved in the study, for the manual identification of stereotypes required for **RQ**₁, we selected four members of the authors' research lab that did not have any knowledge about the system selection or TeStereo internals to avoid bias that could be introduced by the authors, and had multiple years of object-oriented development experience; hereinafter, we will refer to this group of participants as the *Taggers*. For the

Algorithm 2: Sampling procedure of methods from the whole set of unit test in the 231 Apache projects.

```

Input:  $B_{\langle stereotype \rangle}, B_{\langle stereotype \rangle}^{(2)}, B_{\langle n, stereotype \rangle}$ 
Output:  $M_{sample}$ 
1 begin
2    $N = [1..9], ST = ["Logger" \dots "Unclassified"];$ 
3    $M_{sample} = \emptyset, workList = \emptyset;$ 
4    $Counter_{\langle stereotype \rangle} = \emptyset;$ 
5   foreach  $\langle n, stereotype \rangle \in N \times ST$  do
6      $m = pickRandomFrom(B_{\langle n, stereotype \rangle});$ 
7     if  $m \notin workList$  then
8        $workList.add(m);$ 
9        $M_{sample}.add(m);$ 
10       $Counter_{\langle stereotype \rangle} ++;$ 
11  while  $|M_{sample}| < 210$  do
12    foreach  $stereotype \in ST$  do
13      if  $Counter_{\langle stereotype \rangle} < 10$  then
14         $selected = FALSE;$ 
15         $m = pickRandomFrom(B_{\langle stereotype \rangle});$ 
16        if  $m \notin workList$  then
17           $selected = TRUE;$ 
18        if  $!selected$  then
19           $m = pickRandomFrom(B_{\langle stereotype \rangle}^{(2)});$ 
20          if  $m \notin workList$  then
21             $selected = TRUE;$ 
22          if  $!selected$  then
23             $workList.add(m);$ 
24             $M_{sample}.add(m);$ 
25             $Counter_{\langle stereotype \rangle} ++;$ 

```

tasks required with RQ_2 (i.e., writing or evaluating summaries), we contacted (via email) students from the SE classes at the authors' university and external students and researchers. From the participants that accepted the invitation, we selected three groups that we will refer to as $SW_{-TeStereo}$, $SW_{+TeStereo}$, and SR , which stand for summary writers without access to the stereotypes, summary writers with access to the stereotypes, and summary readers, respectively; note that there was no overlap of participants between the three groups. For the

evaluation in **RQ**₃, we mined the list of contributors of the 231 Apache projects; we call this group of participants as *AD* (Apache Developers). We identified the contributors of the projects and contacted them by email to participate in the study. We sent out e-mails listing only the links to the projects to which developers actually contributed (i.e., developers were not contacted multiple times for each project). In the end, we collected 25 completed responses from Apache developers.

4.3.3 Experimental Design

To answer **RQ**₁, we randomly split M_{Sample} into two groups, and then we conducted a user study in which we asked four *Taggers* to manually identify the proposed stereotypes from the methods in both groups (i.e., each *Tagger* read 105 methods). Before the study, one of the authors met with the *Taggers* and explained the stereotypes to them; *Taggers* were also provided with a list, which included the stereotypes and rules listed in Table 4.1 and Table 4.2. During the study, the methods were displayed to the *Taggers* in an html-based format using syntax highlighting. After the tagging, we asked the *Taggers* to review their answers and solve disagreements (if any) after a follow-up meeting. In this meeting, we did not correct the taggers, rather we explained stereotypes that were completely omitted (without presenting the methods from the sample) in order to clarify them; subsequently, the *Taggers* were able to amend the original tags or keep them the same as they saw fit (we did not urge them to alter any tags). In the end, they provided us with a list of stereotypes for the analyzed methods. We compared the stereotypes identified by TeStereo to the stereotypes provided by the *Taggers*. Because of the multi-label classification

nature of the process, we measured the accuracy of TeStereo by using four metrics widely used with multi-class/label problems [140]: micro-averaging recall (μRC), micro-averaging precision (μPC), macro-averaging recall (MRC), and macro-averaging precision (MPC). The rationale for using micro and macro versions of precision and recall was to measure the accuracy globally (i.e., micro) and at stereotype level (i.e., macro). We discuss the results of **RQ**₁ in Section 4.4.1.

To answer **RQ**₂, for each method in M_{Sample} , we automatically built two html versions (with syntax highlighting) of the source code: with and without stereotype tags. The version with tags was assigned to participants in group $SW_{+TeStereo}$, and the version without tags was assigned to participants in $SW_{-TeStereo}$. Each group of participants had 14 people; therefore, each participant was asked to (i) read 15 methods randomly selected (without replacement) from M_{Sample} , and (ii) write a summary for each method. Note that the participants in $SW_{-TeStereo}$ had no prior knowledge of our proposed stereotypes. In the end, we obtained two summaries for each method of the 210 methods m_i ($14 \times 15 = 210$ methods): one based only on source code ($c_{-TeStereo}^i$), and one based on source code and stereotypes ($c_{+TeStereo}^i$). After collecting the summaries, each of the 14 participants in the group SR (i.e., summary readers) were asked to read 15 methods and evaluate the quality of the two summaries written previously for each method. The readers did not know from where the summaries came from, and they got to see the summaries in pairs with the test code at the same time. The quality was evaluated by following a similar procedure and using quality attributes as done in previous studies for automatic generation of documentation [110, 50, 142, 95]. The

summaries were evaluated by the participants in terms of completeness, conciseness, and expressiveness. Section 4.4.2 discusses the results for **RQ₂**.

Finally, to answer **RQ₃**, we distributed a survey to Apache developers in which we asked them to evaluate the usefulness of TeStereo reports and stereotypes. The developers were contacted via email; each developer was provided with (i) a TeStereo html report that was generated for one Apache project to which the developer contributes, and (ii) a link to the survey. For developers who contributed to multiple Apache projects, we randomly assigned one report (from the contributions). The survey consisted of two parts of questions: background and questions related to TeStereo reports and the stereotypes. Section 4.4.3 lists the questions in the second part. The answers were analyzed using descriptive statistics for the single/multiple choice questions; and, in the case of open questions, the authors manually analyzed the free text responses using open coding [73]. More specifically, we analyzed the collected data based on the distributions of choices and also checked the free-text responses in depth to understand the rationale behind the choices. The results for **RQ₃** are discussed in Section 4.4.3.

4.4 Empirical Results

In this section, we discuss the results for each research question.

4.4.1 What is accuracy for identifying stereotypes?

Group	μPC	μRC	MPC	MRC
G1	0.87 (0.98)	0.82 (0.89)	0.82 (0.99)	0.77 (0.92)
G2	0.80 (0.95)	0.89 (0.94)	0.80 (0.94)	0.84 (0.94)

Table 4.3: Accuracy Metrics for Stereotype Detection. The table lists the results for the first round of manual annotation, and second round (in bold) after solving inconsistencies.

Four annotators manually identified stereotypes from 210 unit methods in M_{sample} . Note that the annotators worked independently in two groups, and each group worked with 105 methods. The accuracy of TeStereo measured against the set of stereotypes reported by the annotators is listed in Table 4.3. In summary, there was a total of 102 (2.31%) false negatives (i.e., TeStereo missed the stereotype) and 118 (2.68%) false positives (i.e., the *Taggers* missed the stereotype) in both groups.

We manually checked the false negatives and false positives in order to understand why TeStereo failed to identify a stereotype or misidentified a stereotype. TeStereo did not detect some stereotypes (i.e., false negatives) in which the purpose is defined by inter-procedural calls, in particular *Logger*, *APIUtilityVerifier* and *InternalCallVerifier*. For instance, the stereotype *Logger* is for unit tests methods performing logging operations by calling the Java `PrintStream` and `Logger` APIs; however, there are cases in which the test cases invoke custom logging methods or loggers from other APIs (e.g., `XmlLogger` from Apache ant). The unit test case in Figure 4.9 illustrates the issue; while it was tagged as a *Logger* by the *Taggers*, it was not tagged by TeStereo because

XmlLogger is different than the standard Java logging. Few cases of the false negatives were implementation issues; therefore, we used the false positives to improve the stereotypes detection.

```

@Test public void test() throws Throwable {
    final XmlLogger logger = new XmlLogger();
    final Cvs task = new Cvs();
    final BuildEvent event = new BuildEvent(task);
    logger.buildStarted(event);
    logger.buildFinished(event);}

```

Figure 4.9: *Logger* missed by TeStereo.

Because the *Taggers* were not able to properly detect some stereotypes (i.e., false positives), we re-explained to them the missed stereotypes (using the name and rules and without showing methods from the sample); in some cases, participants did not tag methods with the “Test Initializer” stereotype, because they did not notice the custom annotation `@Before`. Afterward, we generated a new version of the sample (same methods but with improved stereotypes detection), and then we asked the *Taggers* to perform a second round of tagging. We only asked the annotators to re-tag the methods in the false positive and false negative sets. Finally, we recomputed the metrics, and the results for the second round are shown in bold in Table 4.3. The results from the second round showed that TeStereo’s accuracy improved and the inconsistencies were reduced to 64 (1.45%) false negatives and 25 (0.57%) false positives. The future work will be devoted to improving the data flow analysis and fixing the false negatives.

Summary for RQ₁. TeStereo is able to detect stereotypes with high accuracy (precision and recall), even detecting cases in which human annotators fail. However, it has some limitations due to the current implementation of the data-flow based analysis.

4.4.2 Do the proposed stereotypes improve comprehension of tests cases (i.e., methods in test units)?

To identify whether the stereotypes improve comprehension of methods in unit tests, we measured how good the manually written summaries are when the test cases include (or not) the `TeStereo` stereotypes. We first collected manually generated summaries from the two participant groups $SW_{+TeStereo}$ and $SW_{-TeStereo}$ as described in Section 4.3. Then, the summaries were evaluated by a different group of participants who read and evaluated the summaries.

During the “writing” phase we asked the participants to indicate with “N/A” when they were not able to write a summary because of lack of either context or information or they were not able to understand the method under analysis. In 78 out of 420 cases, we got “N/A” as a response from the summary writers; 55 cases were from the participants using only the source code and 23 cases were from participants using the source code and the stereotypes. In total, 64 methods had only one version of the summary available (7 methods had two “N/A”); therefore, the summary readers only evaluated the summaries for 139 ($210 - 64 - 7$) methods in which both versions of the summary were available. Consequently, during the reading phase, 278 summaries were evaluated by 14 participants. It is worth noting that according to the design of the experiment each participant had to evaluate the summaries for 15 methods; however, because of the discarded methods, some of the participants were assigned with fewer than 15 methods. The results for *completeness*, *conciseness*, and *expressiveness* are summarized in Table 4.4.

Completeness. This attribute is intended to measure whether the summary writers were able to include important information in the summary, which

represents a high level of understanding of the code under analysis [110]. In terms of completeness, there is a clear difference between the summaries written by participants that had the `TeStereo` stereotypes and those that did not have stereotypes; while 80 summaries from $SW_{+TeStereo}$ were ranked as not missing any information, 46 from $SW_{-TeStereo}$ were ranked in the same category. On the other side of the scale, only 12 summaries from $SW_{+TeStereo}$ were considered to miss the majority of the important info, compared to 30 summaries from $SW_{-TeStereo}$. Thus, the writers assisted with `TeStereo` stereotypes were able to provide better summaries (in terms of completeness), which suggests that the stereotypes helped them to comprehend the test cases better. Something interesting to highlight here is the fact that some of the writers (from $SW_{+TeStereo}$) included in their summary information based on the stereotypes: *“This is a test initializer.”*, *“initialize an empty test case”*, *“This method checks whether ‘slngld’ is null and ‘equals’ equals to expected.”*, *“This is an empty test that does nothing.”*, *“This is an ignored test method which validates if the fixture is installed.”*, and *“this setup will be run before the unit test is run and it may throw exception”*.

Conciseness. This attribute evaluates if the summaries contain redundant information. Surprisingly, the results are the same for both types of summaries (Table 4.4); 95 summaries from each group ($SW_{+TeStereo}$ and $SW_{-TeStereo}$) were evaluated as not containing redundant information, and only nine summaries from each group were ranked as including significant amount of redundant information. This is surprising coincidence for which we can not have a clear explanation. However, examples of summaries ranked with a low conciseness show the usage of extra but unrelated information added by the writer: *“Not sure what is going on*

Do you think the message is complete?	<i>SW_{-TeStereo}</i>	<i>SW_{+TeStereo}</i>
• Does not miss any imp. info.	46(33.1%)	80(57.6%)
• Misses some important info.	63(45.3%)	47(33.8%)
• Misses the majority of imp. info.	30(21.6%)	12(8.6%)
Do you think the message is concise?	<i>SW_{-TeStereo}</i>	<i>SW_{+TeStereo}</i>
• Contains no redundant info.	95(68.3%)	95(68.3%)
• Contains some redundant info.	35(25.1%)	35(25.1%)
• Contains a lot of redundant info.	9(6.4%)	9(6.4%)
Do you think the description is expressive?	<i>SW_{-TeStereo}</i>	<i>SW_{+TeStereo}</i>
• Is easy to read and understand	90(64.7%)	78(56.1%)
• Is somewhat readable	35(25.2%)	42(30.2%)
• Is hard to read and understand	14(10.1%)	19(13.7%)

Table 4.4: Questions used for **RQ₂** and the # of answers provided by the participants for the summaries written without (*SW_{-TeStereo}*) and with (*SW_{+TeStereo}*) access to stereotypes.

here, but the end results is checking if `r7 == 'ABB: Hello A from BBB'`.”, “Maybe it’s testing to see if a certain language is comparable to another, but I can’t tell”, and “this one has an ignore annotation will run like a normal method which is to test the serialize and deserialize performance by timing it.”.

Expressiveness. This attribute aims at evaluating whether the summaries are easy to read. 90 summaries written without having access to the stereotypes were considered as easy to read compared to 78 summaries from the writers with access to the stereotypes. However, when considering the answers for the summaries ranked as easy-to-read or somewhat-readable, both *SW_{+TeStereo}* and *SW_{-TeStereo}* account for 86%-90% of the summaries, which are very close. One possible explanation for the slight difference in favor of *SW_{-TeStereo}* might be that the extra *TeStereo* tag information could increase the complexity of the summaries. For example, the summary “This is an ‘ignored’ test which also does

nothing so it makes sure that the program can handle nothing w/o blowing up (it throws an exception not just the stack trace).” is hard to read although it contains the keyword “ignore”. Another example is “*setup the current object by assigning values to the tomcat, context, and loader fields.*”

Rationale. We also analyzed the free-text answers provided by the summary readers when supporting their preferences for summaries from $SW_{-TeStereo}$ or $SW_{+TeStereo}$. Overall, 72 explanations claimed that the choice was based on the *completeness* of the summary. Examples include: ‘*The summary allows for a deeper understanding of what the program is doing and what it is using to make itself work*’, “*I prefer this summary because it is more detailed than the other.*”, and “*I like this one because it gives you enough information without going overboard*”. 52 out of the 72 explanations were for answers in favor of summaries from $SW_{+TeStereo}$. Thus, the rationale provided by the readers reinforces our findings that $TeStereo$ helped developers to comprehend the test cases and write better test summaries that include important info.

26 explanations mentioned the *expressiveness* as the main attribute for making their choice: “*This summary is very easy for programmers to understand.*” and “*Easier to read, while I can hardly understand what Summary1 is trying to say.*”. In this case, 12 explanations are for readers in favor of summaries from $SW_{+TeStereo}$. Finally, 4 decisions were made based on the *conciseness* of the summaries: “*Slightly more concise*”, “*Concise*”, “*This concisely explains what is going on with no extra material but it could use a little more information.*”, and “*Too much extra stuff in Summary 1*”.

Summary for RQ₂. The evaluation of the scenario of writing and reading summaries for unit test methods suggests that the proposed unit test stereotypes improve the comprehension of tests cases. The results showed that manually written summaries with assistance from TeStereo tags covered more important information than the summaries written without it. In addition, by comparing the evaluation between summaries with and without using TeStereo tags, the results indicated that TeStereo tags did not introduce redundant information or make the summaries hard to read.

4.4.3 What are the developers perspectives of the TeStereo-based reports for systems in which they contributed?

We received completed surveys from 25 developers of the Apache projects. While the number of participants is not very high, participation is an inherent uncontrollable difficulty when conducting a user study with open source developers. In terms of the highest academic degree obtained by participants, we had the following distribution: one with a high school degree (4%), seven with a Bachelor's degree (28%), sixteen with a Master's degree (64%), and one with Ph.D. (4%). Concerning the programming experience, the mean value is 20.8 years of experience and the median value is 20 years. More specifically, participants had on average 12.9 years of industrial/open-source experience (the median was 14 years). The questions related to **RQ₃** and the answers provided by the practitioners are as the following:

SQ₁. Which of the following tasks do you think the tags are useful for? (Multiple-choice and Optional). 48% selected "*Test case comprehension/understanding*", 44 % selected "*Generating summary of unit test*

case”, 40% vote for the option “*Unit test case maintenance*”, and only 8% checked the option “*Debugging unit test cases*”.

SQ₂. Which of the following tasks do you think the reports are useful for? (Multiple-choice and Optional). 60% selected “*Test case comprehension/understanding*”, 48 % selected “*Generating summary of unit test case*”, 40% vote for the option “*Unit test case maintenance*”, and only 8% checked the option “*Debugging unit test cases*”.

SQ₃. What tasks(s) do you think the tags/report might be useful for? (Open question) To complement the first two SQs, SQ₃ aims at examining if the stereotypes and reports are useful from a practitioner’s perspective for other software-related tasks. We categorized the responses into the following groups:

- **Unit test quality evaluation:** The participants mentioned the following uses like “*evaluate the quality of the unit tests*”, “*a rough categorization [of unit tests] by runtime, e.g. ‘fast’ and ‘slow’*”, and “*quality/complexity metrics*”.
- **Bad test detection:** two participants suggested that the technique could be used for detecting bad tests. The responses include “*Fixing a system with a lot of bad tests*” & “*probably verifying if there’s good ‘failure’ message*”.
- **Code navigation:** One response suggested that the TeStereo report is “*a good way to jump into the source code*”. This response demonstrates that users can comprehend the test code easier by looking at the TeStereo report.

SQ₄. Is the summary displayed when hovering over the gray balloon icon useful for you? (Binary-choice). TeStereo’s reports include a speech balloon (Figure 4.10) icon that displays a summary automatically generated by

Class: org.apache.catalina.connector.TestResponse

```

@APIUtilityVerifier BranchVerifier UtilityVerifier EqualityVerifier HybridVerifier
@Test public void testBug49598() throws Exception {
    Tomcat tomcat=getTomcatInstance();
    Context ctx=tomcat.addContext("",null);
    Tomcat.addServlet(ctx,"servlet",new Bug49598Servlet());
    ctx.addServletMapping("/","servlet");
    tomcat.start();
    Map> headers=new HashMap<>();
    getUrl("http://localhost:" + getPort() + "/",new ByteChunk(),headers);
    for ( Map.Entry> header : headers.entrySet()) {
        if (header.getKey() == null) {
            List values=header.getValue();
            if (values.size() == 1 && values.get(0).startsWith("HTTP/1.1")) {
                continue;
            }
        }
        fail("Null header name detected for value " + values);
    }
}

```

This method/test case:

- Verifies values of objects/variables related to API calls (Java or TPL)
- Verifies assertions inside branch conditions
- Verifies (un)successful execution of the test case by reporting explicitly a failure
- Verifies whether objects/variable are equal to an expected value
- Contains more than 2 JUnit-based stereotypes

Figure 4.10: TeStereo documentation for a test case in the Tomcat project.

aggregating the descriptions of the stereotypes¹. We wanted to evaluate usefulness of this feature, and we obtained 14 positive and 11 negative responses. The positive answers were augmented with rationale such as ‘*It gives the purpose of unit test case glimpsly*’, ‘*Was hard to find, but yes, this makes it easier to grok what you’re looking at*’, and ‘*It is clear*’. As for the negative answers, the rationale described compatibility issues with mobile devices (‘*I am viewing this on an iPad. I can’t hover*’, ‘*hovers don’t seem to work*’). Yet, some participants found the summary redundant since the info was in the tags.

SQ₅. What are the elements that you like the most in the report? (Multiple-choice). Most of the practitioners selected source code box (14 answers, 56%) and test case tags (11 answers, 44%). This suggests that the surveyed practitioners recognize the benefit of the stereotype tags, and are more likely to use the combination of tags and source code boxes. We received 5 answers (20%) for ‘*gray balloon icon & summary*’, 3 (12%) for ‘*navigation*

¹Note that TeStereo’s reports (including the balloon and summary features) were only available for the Apache developers.

box”, and 4 (16%) for “filter”.

SQ₆. Please provide an example of the method that you think the tags are especially useful for unit test case comprehension (Open question).

For SQ₆, we collected nine responses in total, and this is related to the open question nature in which some participants filled blank spaces or other characters. One participant mentioned the `testForkModeAlways` method in project `maven` and explained his choice with the following rationale: “*This method is tagged 'BranchVerifier', and arguably its cyclomatic complexity is too great for a test.*” This explanation shows that the stereotype tags (i.e., *BranchVerifier* and *IterativeVerifier*) help developers identify test code that should not include branches/loops. Another response mentions the `testLogIDGenerationWithLowestID` method in project `Ace`; the method was tagged as *Logger* by `TeStereo` and the practitioner augmented his answer with the following: “*Logging in unit tests is usually a code smell, just by looking at this method I realize what `event.toRepresentation()` returns is not compared with an expected value.*” This example shows that stereotype tags are also useful for other software maintenance tasks such as code smell detection. Another example is the method `testLogfilePlacement` in `Ant`, and the developer claimed that this is a very good example because the tags helped him to identify that the test case is an internal call verifier. Some responses did not provide the signature of the method, but their comments are useful (e.g., “*TestCleaner is useful to show complexity (hopefully unneeded) of test cases*” and “*I believe they would be useful to check if developers are only developing shallow test cases*”).

SQ₇. Please provide an example of the method that you think the tags are NOT useful for unit test case comprehension (open question). For SQ₇,

```

@Test public void testSingleElementRange(){           1
    final int start=1;                               2
    final int max=1;                                 3
    final int step=-1;                               4
    final List seq=new ArrayList();                 5
    final IntegerSequence.Range                    6
        r=IntegerSequence.range(start,max,step);
    for ( Integer i : r) {seq.add(i);}              7
    Assert.assertEquals(1,seq.size());              8
    Assert.assertEquals(seq.size(),r.size());       9
    Assert.assertEquals(start,seq.get(0).intValue());10
}                                                    11

```

Figure 4.11: *InternalCallVerifier* missed by TeStereo.

we collected nine valid responses. One example highlights the need for improving the limitations mentioned in Section 4.4.1, in particular the method `nonExistentHost()` with the following comment from a developer: *“it’s about ‘verifies (un)successful execution’, but it’s about expected exception in particular case.”* This issue is due to the fact that TeStereo performs over-approximation during static analysis. Although TeStereo does not track any branch conditions in the method (some paths may not be executed with a certain input), the over-approximate approach guarantees that potential paths are not missed when TeStereo tags the unit test case.

SQ₈. **What are the elements that you think need improvement in the report? (Open question).** For SQ₈, we collected 13 valid responses. Some practitioners suggested augmenting the reports with summaries describing the method, for example: *“If it can explain about the function, it would be great”* and *“It’d be nicer if the tags conveyed more semantics concepts, rather than mere syntactic properties”*. Also, some comments asked for improvement of the user experience in the reports: *‘the report should highlight in red the test methods which do contain any assertions’* and *“Being able to collapse the code blocks to make it easier to see summaries”*.

SQ₉. **What additional information would you find helpful if it were**

included in the reports? (Open question). These are some sample answers:

- **Test suite quality:** some participants suggested that we need to create a new stereotype to identify redundant test cases, include test coverage info, show evolution of the tags per commits, and indicate size of the method which can be an indicator of methods that need refactoring;
- **Integration:** some practitioners also suggested that we add a link to the full source code on GitHub, so that the code can be seen in its larger context. They also suggested that we integrate TeStereo into SonarQube;
- **Detailed description:** these suggestions are more related to personal preferences; for example, *“highlighting the aspect in the code”*, *“I would be interested in being able to find which tests check which accessors or methods and vice versa.”*, and *“specify what is verified by this method”*. The last comment is aligned with the purpose of other summarization approaches such as TestDescriber [122] and UnitTestScribe [95], which generate natural language descriptions of the assertions and focal methods.

Summary for RQ₃. Overall, we obtained 25 responses from active Apache developers, who provided us with useful feedback for improving the stereotypes and the reports. Concerning the usefulness of the stereotypes and the reports, most of the surveyed developers believed that TeStereo’s tags and reports are useful for test case comprehension tasks. Other tasks reported by the developers, in which the tags and the reports might be useful, are code smell detection and source code navigation.

4.4.4 Threats to Validity

Threats to internal validity relate to response bias by participants that either had more difficulty or did not have problems while understanding unit test cases or writing summaries. Based on the results of the study and the large number of the participants, we observed that responses were not dominantly distributed to extremes, which would indicate that these developers were particularly biased based on such difficulty.

The external threats to validity relate to generalizing the conclusions from the study. In our study, we state that these results are based on our sample of unit test cases and participants, but do not claim that these results generalize to all developing systems in other languages and other developers. However, we do present the sampling procedure of unit tests from the whole set of unit test in the 231 Apache projects, which aims to minimize the threat. The selected methods are highly diverse in terms of methods size, method stereotypes, and the number of stereotypes. In addition, we present demographic information of the participants that suggests that we have a diverse sample of developers.

Another threat to validity is that `TeStereo` has some limitations due to the current implementation of the data-flow based analysis. For example, `TeStereo` cannot interpret the variable assignment relations since those require inter-procedural analysis, which leads to false negatives. For example, the unit test case in Figure 4.11 was annotated as `InternalCallVerifier` by the taggers since the method has a slicing path from variable `r` to `seq`, and `IntegerSequence.range(start,max,step)` at line 6 is an internal method call. However, `TeStereo` cannot interpret the variable assignment relations in the for-loop (line 7), since it needs to understand the assignment relations in “*Integer*

i:r” and “*seq.add(i)*”. Due to this limitation, TeStereo loses the backward tracking to the internal function call.

4.5 Related Work

There are some related techniques for studying unit test cases, which include unit test case minimization [92, 91], prioritization [56, 135, 143], test case descriptions [85, 122, 95, 165], code quality [33], test coverage [79], data generation [93, 101], unit test smells [40, 151, 104, 150, 149], fault localization [159], automatic test case generation [52, 64, 69, 145, 160, 155, 91], and automatic recommendation of test examples [126]. TeStereo is also related to (i) techniques for generating documentation for software artifacts [80, 63, 100, 97, 111], and (ii) other approaches for supporting code comprehension provided by difference tools [122, 95, 110, 113, 68]. Compared to the existing approaches, TeStereo is novel in that it considers stereotypes at the test suite level.

4.5.1 Stereotypes Definition and Detection

Several studies [63, 61, 60] focused on classifying software entities, such as methods, classes, and repository commits. Generally, the studies classify software entities as different stereotypes based on static analysis techniques and predefined rules [63, 61]. Dragan et al. first presented and defined taxonomy of method stereotypes [63]. The authors implemented a tool, namely StereoCode, which automatically identifies method stereotypes for all methods in a system. Later, Dragan et al. extended the classification of stereotypes to class

level granularity by considering frequency and composition of the method stereotypes in one class [61]. The results showed that 95% of the classes were stereotyped by their approach. Dragan et al. further refined stereotypes at the commit level [60]. The categorization of a commit is based on the stereotype of the methods that are added/deleted in the commit. Different from Dragan et al.'s implementation that works on C++, Moreno and Marcus [112] implemented a classification tool, named `JStereoCode`, for automatically identifying method and class stereotypes in Java systems. Andras et al. measured runtime behavior of methods and method calls to reflect method stereotypes [32]. Their observation showed that most methods behave as expected based on the stereotypes. Overall, none of the existing studies focus on stereotype classification of unit test cases. *Our approach is the first one to define and classify unit test case stereotypes by (i) analyzing unit test API calls and (ii) performing static analysis on data/control flows.*

4.5.2 Utilizing Stereotypes for Automatic Documentation

A group of approaches and studies utilize stereotype identification for other goals. Linares-Vásquez et al. [97, 50] implemented a tool, namely `ChangeScribe`, for automatically generating commit messages. `ChangeScribe` extracts changes between two adjacent versions of a project and identifies involved change types in addition to performing commit level stereotype analysis. Dragan et al. showed that the distribution of method stereotypes could be an indicator of system architecture/design [62]. In addition, their technique could be utilized in clustering systems with similar architecture/design. Moreno et al. [110, 113] and Abid et al. [28] utilized class stereotypes to summarize the

responsibilities of classes in different programming languages (Java and C++) respectively. Ghafari et al. [72] used stereotypes to detect focal methods (methods responsible for system state changes examined through assertions in unit tests) in a unit test case. *Overall, our work is first to improve unit test comprehension and test suite navigation by using unit test stereotypes.*

4.5.3 Automatic Documentation of Unit Test Cases

Kamimura and Murphy presented an approach for automatically summarizing JUnit test cases [85]. Their approach identified the focal method based on the number of invocations of the method. Panichella et al. [122] presented an approach, `TestDescriber`, for generating test case summaries on automatically generated JUnit test cases. The summary contains three different levels of granularity: class, method, and test level (i.e., branch coverage). Furthermore, Li et al. [95] proposed `UnitTestScribe` that combines static analysis, natural language processing, and backward slicing techniques to automatically generate detailed method-level summarization for unit test cases. Zhang et al. [164, 165] presented a natural language-based approach that extracts the descriptive nature of test names to generate test templates. *Overall, none of the existing techniques for documenting unit test cases focuses on unit test case stereotypes, besides our approach.*

4.6 Conclusion

In this chapter, we first presented a novel catalog of stereotypes for methods in unit tests to categorize JUnit test cases into 21 stereotypes; the catalog aims

at improving program comprehension of unit test, when the unit test methods are annotated with the stereotypes. We propose an approach, *TeStereo*, for automatically tagging stereotypes for unit tests by performing control-flow, data-flow, and API call based static analyses on the source code of a unit test suite. *TeStereo* also generates html reports that include the stereotype tags, source code, and navigation features to improve the comprehension and browsing of unit tests in a large test suite.

To validate *TeStereo*, we conducted empirical studies based on 231 Apache projects, 46 students and researchers and a survey with 25 Apache developers. Also, we evaluated 420 manually generated summaries and 210 unit test methods with and without stereotype annotations. Our results show that (i) *TeStereo* achieves very high precision and recall (0.99 & 0.94) in terms of annotating unit test stereotypes, (ii) the proposed stereotypes improve comprehension of unit test cases in software maintenance tasks, and (iii) most of the developers agreed that *TeStereo* stereotypes and reports are useful. Our results demonstrate that *TeStereo*'s tags are useful for test case comprehension tasks as well as other tasks, such as code smell detection and source code navigation.

4.7 Bibliographical Notes

The paper supporting the content described in this Chapter was written in collaboration with the members of the SEMERU group at William and Mary :

- **Li, B.**, Vendome, C., Linares-Vásquez, M., and Poshyvanyk, D. “Stereotype-based Tagging of Unit Test Cases.” , under submission.

Chapter 5

Conclusion

The proposed dissertation makes several research contributions: (i) empirical studies on documenting software artifacts in practice (e.g., database usage and constraints, unit test cases), (ii) a novel practical approach for documenting database usages, and (iii) a novel approach for documenting unit test cases.

In Chapters 2 and 3, the proposed works are first motivated by studies in which we surveyed open source/professional developers to understand their perspective of database operations and unit test cases. We found that developers believe maintaining good documentations is important for the quality of a software system. We also mined changes of a large amount of open source projects to show that in practice the projects lack related comments. Motivated by the findings of the studies, we proposed two novel approaches. *DBScribe* is a novel approach for automatically generating natural language (NL) documentation at the source code method level that describes database usages and constraints for a given DCA. In addition, we presented a novel approach *UnitTestScribe* that combines static analysis, natural language processing, backward slicing and code summarization techniques in order to automatically generate expressive NL descriptions that concisely document the purpose of

unit test methods. To evaluate our tools (`DBScribe` and `UnitTestScribe`), we conducted online surveys with industrial developers and graduate students. In general, participants indicated that descriptions generated by our tools are complete, concise, and easy to read.

In chapter 4, we first presented a novel catalog of stereotypes for methods in unit tests to categorize JUnit test cases into 21 stereotypes; the catalog aims at improving program comprehension of unit test, when the unit test methods are annotated with the stereotypes. We propose an approach, `TeStereo`, for automatically tagging stereotypes for unit tests by performing control-flow, data-flow, and API call based static analyses on the source code of a unit test suite. `TeStereo` also generates html reports that include the stereotype tags, source code, and navigation features to improve the comprehension and browsing of unit tests in a large test suite. Our results demonstrate that `TeStereo`'s tags are useful for test case comprehension tasks as well as other tasks, such as code smell detection and source code navigation.

Bibliography

- [1] Apache software foundation. <http://www.apache.org/>.
- [2] Dbscribe online appendix. <http://www.cs.wm.edu/semeru/data/ISSTA16-DBScribe>.
- [3] Fina <http://sourceforge.net/projects/fina/>.
- [4] GitHub API. <https://developer.github.com/v3/>. Last accessed: 2015/01/15.
- [5] Glimpse. <https://github.com/Glimpse/Glimpse/>.
- [6] Google-api-dotnet. <https://github.com/google/google-api-dotnet-client/>.
- [7] Jmockit. an automated testing tooltik for java. <http://jmockit.org/>.
- [8] Jsqparser. <http://jsqparser.sourceforge.net/>.
- [9] Junit. <http://junit.org/>.
- [10] Liminal ltda <http://www.liminal-it.com/>.
- [11] Mockito. <http://mockito.org/>.
- [12] Msdn. <https://msdn.microsoft.com/>.

- [13] Nunit. <http://www.nunit.org/>.
- [14] Openemm e-mail & marketing automation <http://sourceforge.net/projects/openemm/files/OpenEMM%20software/OpenEMM%206.0/>.
- [15] Qualtrics. <http://www.qualtrics.com/>.
- [16] Risk it repository. <https://riskitinsurance.svn.sourceforge.net/>.
- [17] Sando. <https://github.com/abb-iss/Sando/>.
- [18] Sonarqube <http://www.sonarqube.org/>.
- [19] Srcml. <http://www.srcml.org/>.
- [20] Srcml.net. <https://github.com/abb-iss/SrcML.NET/>.
- [21] Swum.net. <https://github.com/abb-iss/Swum.NET/>.
- [22] *TeStereo Online appendix.* <https://sites.google.com/site/testereonline/>.
- [23] Umas. <https://github.com/University-Management-And-Scheduling>.
- [24] Unit Test Basics. https://msdn.microsoft.com/en-us/library/hh694602.aspx#BKMK_Writing_your_tests. Last accessed: 2015/10/15.
- [25] Unittestscribe online appendix. <http://www.cs.wm.edu/semeru/data/ICST16-UnitTestScribe/>.
- [26] Xinco rev 700 <http://sourceforge.net/p/xinco/code/700/tree/trunk/>.
- [27] xunitpatterns. <http://xunitpatterns.com/>.

- [28] NAHLA J. ABID, NATALIA DRAGAN, MICHAEL L. COLLARD, AND JONATHAN I. MALETIC. Using stereotypes in the automatic generation of natural language summaries for c++ methods. In *Proc. ICSME*, pages 561–565. IEEE, 2015.
- [29] RAKESH AGRAWAL, TOMASZ IMIELIŃSKI, AND ARUN SWAMI. Mining association rules between sets of items in large databases. In *ACM SIGMOD Record*, volume 22, pages 207–216. ACM.
- [30] REDA ALHAJJ. Extracting the extended entity-relationship model from a legacy relational database. *Information Systems*, 28(6):597–618, 2003.
- [31] NOUH ALHINDAWI, NATALIA DRAGAN, MICHAEL L. COLLARD, AND JONATHAN I. MALETIC. Improving feature location by enhancing source code with stereotypes. 2013.
- [32] PETER ANDRAS, ANJAN PAKHIRA, LAURA MORENO, AND ANDRIAN MARCUS. A measure to assess the behavior of method stereotypes in object-oriented software. In *WETSoM 2013*. IEEE.
- [33] MAURICIO F ANICHE, GUSTAVO A OLIVA, AND MARCO A GEROSA. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In *CSMR'13*, pages 111–120. IEEE, 2013.
- [34] DAVID ASTELS. *Test-Driven Development: A Practical Guide: A Practical Guide*. Prentice Hall PTR, 2003.
- [35] KAPIL BAKSHI. Considerations for big data: Architecture and approach. In *IEEE Aerospace Conference*, pages 1–7, 2012.

- [36] LUCIANO BARESI AND MATTEO MIRAZ. Testful: Automatic unit-test generation for java classes. In *Proceedings of ICSE10*, pages 281–284, New York, NY, USA, 2010. ACM.
- [37] GABRIELE BAVOTA, GERARDO CANFORA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, AND SEBASTIANO PANICHELLA. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *ICSM*, pages 280–289. IEEE, 2013.
- [38] GABRIELE BAVOTA, MALCOM GETHERS, ROCCO OLIVETO, DENYS POSHYVANYK, AND ANDREA DE LUCIA. Improving software modularization via automated analysis of latent topics and dependencies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(1):4, 2014.
- [39] GABRIELE BAVOTA, ROCCO OLIVETO, MALCOM GETHERS, DENYS POSHYVANYK, AND ANDREA DE LUCIA. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering*, 40(7):671–694, 2014.
- [40] GABRIELE BAVOTA, ABDALLAH QUSEF, ROCCO OLIVETO, ANDREA DE LUCIA, AND DAVID BINKLEY. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *ICSM'12 28th*, pages 56–65. IEEE, 2012.
- [41] KENT BECK. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002.

- [42] KENT BECK AND CYNTHIA ANDRES. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2nd edition, 2004.
- [43] KENT BECK AND ERICH GAMMA. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.
- [44] SEBASTIAN BERGMANN. Phpunit. <https://phpunit.de>.
- [45] RAYMOND PL BUSE AND WESTLEY R WEIMER. Automatic documentation inference for exceptions. In *2008 ISSTA*.
- [46] RAYMOND PL BUSE AND WESTLEY R WEIMER. Automatically documenting program changes. In *Proceedings of the IEEE/ACM ASE*, pages 33–42, 2010.
- [47] RAYMOND PL BUSE AND THOMAS ZIMMERMANN. Information needs for software development analytics. In *ICSE'12*, pages 987–996. IEEE Press, 2012.
- [48] GERARDO CANFORA, LUIGI CERULO, AND MASSIMILIANO DI PENTA. Ldiff: An enhanced line differencing tool. In *Proceedings of the 31st International Conference on Software Engineering*, pages 595–598. IEEE Computer Society, 2009.
- [49] ANTHONY CLEVE, MAXIME GOBERT, LOUP MEURICE, JEROME MAES, AND JENS WEBER. Understanding database schema evolution: A case study. *Science of Computer Programming*, 97.
- [50] LUIS FERNANDO CORTÉS-COY, MARIO LINARES-VÁSQUEZ, JAIRO APONTE, AND DENYS POSHYVANYK. On automatically generating commit

- messages via summarization of source code changes. In *SCAM'14*, pages 275–284. IEEE, 2014.
- [51] BILL CURTIS, HERB KRASNER, AND NEIL ISCOE. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.
- [52] ERMIRA DAKA, JOSÉ CAMPOS, GORDON FRASER, JONATHAN DORN, AND WESTLEY WEIMER. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on FSE*, 2015.
- [53] ERMIRA DAKA AND GORDON FRASER. A survey on unit testing practices and problems. In *ISSRE'14*, pages 201–211, 2014.
- [54] SERGIO COZZETTI B DE SOUZA, NICOLAS ANQUETIL, AND KÁTHIA M DE OLIVEIRA. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, pages 68–75, 2005.
- [55] A. VAN DEURSEN, L. MOONEN, A. VAN DEN BERGH, AND G. KOK. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, M. Marchesi, editor, pages 92–95. University of Cagliari, 2001.
- [56] DANIEL DI NARDO, NADIA ALSHAHWAN, LIONEL BRIAND, AND YVAN LABICHE. Coverage-based test case prioritisation: An industrial case study. In *ICST'13*, pages 302–311. IEEE, 2013.

- [57] BOGDAN DIT, EVAN MORITZ, MARIO LINARES-VÁSQUEZ, DENYS POSHYVANYK, AND JANE CLELAND-HUANG. Supporting and accelerating reproducible empirical research in software evolution and maintenance using tracelab component library. *Empirical Software Engineering*, 20(5):1198–1236, 2015.
- [58] BOGDAN DIT, MEGHAN REVELLE, MALCOM GETHERS, AND DENYS POSHYVANYK. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, 2013.
- [59] BOGDAN DIT, MEGHAN REVELLE, AND DENYS POSHYVANYK. Integrating information retrieval, execution and link analysis algorithms to improve feature location in software. *Empirical Software Engineering*, 18(2):277–309, 2013.
- [60] NATALIA DRAGAN, MICHAEL L COLLARD, MAEN HAMMAD, AND JONATHAN MALETIC. Using stereotypes to help characterize commits. In *ICSM 2011*, pages 520–523. IEEE, 2011.
- [61] NATALIA DRAGAN, MICHAEL L COLLARD, AND JONATHAN MALETIC. Automatic identification of class stereotypes. In *ICSM 2010*, pages 1–10. IEEE.
- [62] NATALIA DRAGAN, MICHAEL L COLLARD, AND JONATHAN MALETIC. Using method stereotype distribution as a signature descriptor for software systems. In *ICSM 2009*, pages 567–570.
- [63] NATALIA DRAGAN, MICHAEL L COLLARD, AND JONATHAN MALETIC. Reverse engineering method stereotypes. In *ICSM'06*, pages 24–34, 2006.

- [64] FAEZEH ENSAN, EBRAHIM BAGHERI, AND DRAGAN GAŠEVIĆ. Evolutionary search-based test generation for software product line feature models. In *Advanced Information Systems Engineering*, pages 613–628. Springer, 2012.
- [65] JANET FEIGENSPAN, CHRISTIAN KÄSTNER, JÖRG LIEBIG, SVEN APEL, AND STEFAN HANENBERG. Measuring programming experience. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 73–82. IEEE, 2012.
- [66] BEAT FLURI, MICHAEL WURSCH, AND HARALD C GALL. Do code and comments co-evolve? on the relation between source code and comment changes. In *WCRE 2007.*, pages 70–79, Oct.
- [67] BEAT FLURI, MICHAEL WÜRSCH, EMANUEL GIGER, AND HARALD C GALL. Analyzing the co-evolution of comments and source code. *Software Quality Journal*, 17(4):367–394, 2009.
- [68] J. FOWKES, P. CHANTHIRASEGARAN, R. RANCA, M. ALLAMANIS, M. LAPATA, AND C. SUTTON. *IEEE Transactions on Software Engineering*, 2017.
- [69] GORDON FRASER AND ANDREA ARCURI. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.
- [70] GORDON FRASER AND ANDREA ARCURI. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, February 2013.

- [71] JUAN PABLO GALEOTTI, GORDON FRASER, AND ANDREA ARCURI. Extending a search-based test generator with adaptive dynamic symbolic execution. In *Proceedings of the 2014 ISSTA, ISSTA 2014*, pages 421–424. ACM, 2014.
- [72] MOHAMMAD GHAFARI, CARLO GHEZZI, AND KONSTANTIN RUBINOV. Automatically identifying focal methods under test in unit test cases. In *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*, pages 61–70. IEEE, 2015.
- [73] BARNEY G. GLASER AND ANSELM L. STRAUSS. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, 1967.
- [74] MATHIEU GOEMINNE, ALEXANDRE DECAN, AND TOM MENS. Co-evolving code-related and database-related changes in a data-intensive software system. In *CSMR-WCRE 2014*.
- [75] EMILY HILL. *Integrating natural language and program structure information to improve software search and exploration*. University of Delaware, 2010.
- [76] EMILY HILL, LORI POLLOCK, AND K VIJAY-SHANKER. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, pages 232–242. IEEE Computer Society, 2009.
- [77] MATTHEW J HOWARD, SAMIR GUPTA, LORI POLLOCK, AND K VIJAY-SHANKER. Automatically mining software-based, semantically-similar words from comment-code mappings. In *MSR'13*, pages 377–386, 2013.

- [78] ANDY HUNT AND DAVE THOMAS. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.
- [79] CHEN HUO AND JAMES CLAUSE. Interpreting coverage information using direct and indirect coverage. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–243. IEEE, 2016.
- [80] DANIEL JACKSON AND DAVID A LADD. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM'94*, pages 243–252, 1994.
- [81] RANJIT JHALA AND RUPAK MAJUMDAR. Path slicing. In *ACM SIGPLAN Notices*, volume 40. ACM, 2005.
- [82] SIYUAN JIANG, AMEER ARMALY, AND COLLIN MCMILLAN. Automatically generating commit messages from diffs using neural machine translation. In *in Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*.
- [83] MIRA KAJKO-MATTSSON. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering*, 10(1):31–55, 2005.
- [84] EIRINI KALLIAMVAKOU, GEORGIOS GOUSIOS, KELLY BLINCOE, LEIF SINGER, DANIEL M. GERMAN, AND DANIELA DAMIAN. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 92–101, New York, NY, USA, 2014. ACM.

- [85] MANABU KAMIMURA AND GAIL C MURPHY. Towards generating human-oriented summaries of unit test cases. In *ICPC'13*, pages 215–218. IEEE, 2013.
- [86] GREGORY M KAPFHAMMER AND MARY LOU SOFFA. A family of test adequacy criteria for database-driven applications. *ACM SIGSOFT Software Engineering Notes*, 28(5):98–107, 2003.
- [87] MIRYUNG KIM AND DAVID NOTKIN. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319. IEEE Computer Society, 2009.
- [88] MIRYUNG KIM, DAVID NOTKIN, DAN GROSSMAN, AND GARY WILSON JR. Identifying and summarizing systematic code changes via rule inference. *Software Engineering, IEEE Transactions on*, 39(1):45–62, 2013.
- [89] ANDREW J KO, BRAD A MYERS, MICHAEL J COBLENZ, AND HTET HTET AUNG. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on Software Engineering*, 32(12):971–987, 2006.
- [90] CHAN MAN KUOK, ADA FU, AND MAN HON WONG. Mining fuzzy association rules in databases. *ACM Sigmod Record*, 27(1):41–46, 1998.
- [91] YONG LEI AND JAMES H ANDREWS. Minimization of randomized unit test cases. In *Software Reliability Engineering, 2005*. IEEE, 2005.
- [92] ANDREAS LEITNER, MANUEL ORIOL, ANDREAS ZELLER, ILINCA CIUPA, AND BERTRAND MEYER. Efficient unit test case minimization. In *ASE 2007*.

- [93] BOYANG LI, MARK GRECHANIK, AND DENYS POSHYVANYK. Sanitizing and minimizing databases for software application test outsourcing. In *ICST 2014*, pages 233–242. IEEE, 2014.
- [94] BOYANG LI, DENYS POSHYVANYK, AND MARK GRECHANIK. Automatically detecting integrity violations in database-centric applications. In *Proceedings of the 25th International Conference on Program Comprehension*, pages 251–262. IEEE Press, 2017.
- [95] BOYANG LI, CHRISTOPHER VENDOME, MARIO LINARES-VÁSQUEZ, DENYS POSHYVANYK, AND NICHOLAS A KRAFT. Automatically documenting unit test cases. In *Proceedings of ICST’16*, April 2016.
- [96] BEN LIBLIT, ANDREW BEGEL, AND EVE SWEETSER. Cognitive perspectives on the role of naming in computer programs. In *Proceedings of the 18th annual psychology of programming workshop*, 2006.
- [97] MARIO LINARES-VÁSQUEZ, LUIS FERNANDO CORTÉS-COY, JAIRO APONTE, AND DENYS POSHYVANYK. Changescribe: A tool for automatically generating commit messages. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 709–712. IEEE Press, 2015.
- [98] MARIO LINARES-VÁSQUEZ, SAM KLOCK, COLLIN MCMILLAN, AMINATA SABANÉ, DENYS POSHYVANYK, AND YANN-GAËL GUÉHÉNEUC. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM, 2014.

- [99] MARIO LINARES-VÁSQUEZ, BOYANG LI, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. How do developers document database usages in source code? In *ASE'15 - New Ideas Track*, pages 36–41, 2015.
- [100] MARIO LINARES-VÁSQUEZ, BOYANG LI, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 270–281. ACM, 2016.
- [101] RUCHIKA MALHOTRA AND MOHIT GARG. An adequacy based test data generation technique using genetic algorithms. *Journal of information processing systems*, 7(2), 2011.
- [102] ANDY MAULE, WOLFGANG EMMERICH, AND DAVID S ROSENBLUM. Impact analysis of database schema changes. In *ICPC 2008*.
- [103] PAUL W MCBURNEY AND COLLIN MCMILLAN. Automatic documentation generation via source code summarization of method context. In *ICPC 2014*, pages 279–290. ACM.
- [104] GERARD MESZAROS. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
- [105] LOUP MEURICE, CSABA NAGY, AND ANTHONY CLEVE. Detecting and preventing program inconsistencies under database schema evolution. In *Software Quality, Reliability and Security (QRS), 2016 IEEE International Conference on*, pages 262–273. IEEE, 2016.

- [106] LOUP MEURICE, CSABA NAGY, AND ANTHONY CLEVE. Static analysis of dynamic database usage in java systems. In *International Conference on Advanced Information Systems Engineering*, pages 491–506. Springer, 2016.
- [107] AUDRIS MOCKUS, ROY T FIELDING, AND JAMES D HERBSLEB. Two case studies of open source software development: Apache and mozilla. *ACM TOSEM*, 11(3):309–346, 2002.
- [108] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. Automatically discovering, reporting and reproducing android application crashes. In *Software Testing, Verification and Validation (ICST), 2016 IEEE International Conference on*, pages 33–44. IEEE, 2016.
- [109] L. MORENO, G. BAVOTA, M. DI PENTA, R. OLIVETO, A. MARCUS, AND G. CANFORA. Arena: An approach for the automated generation of release notes. *TSE*, PP(99):1–1, 2016.
- [110] LAURA MORENO, JAIRO APONTE, GIRIPRASAD SRIDHARA, ANDRIAN MARCUS, LORI POLLOCK, AND K VIJAY-SHANKER. Automatic generation of natural language summaries for java classes. In *ICPC 2013*, pages 23–32. IEEE.
- [111] LAURA MORENO, GABRIELE BAVOTA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, ANDRIAN MARCUS, AND GERARDO CANFORA. Automatic generation of release notes. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 484–495. ACM, 2014.

- [112] LAURA MORENO AND ANDRIAN MARCUS. Jstereocode: automatically identifying method and class stereotypes in java code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 358–361. ACM, 2012.
- [113] LAURA MORENO, ANDRIAN MARCUS, LORI POLLOCK, AND K VIJAY-SHANKER. Jsummarizer: An automatic generator of natural language summaries for java classes. In *ICPC 2013*.
- [114] C. NAGY, L. MEURICE, AND A. CLEVE. Where was this sql query executed? a static concept location approach. In *SANER 2015*, pages 580–584, March 2015.
- [115] HOAN ANH NGUYEN, TUNG THANH NGUYEN, HUNG VIET NGUYEN, AND TIEN N NGUYEN. idiff: Interaction-based program differencing tool. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 572–575. IEEE, 2011.
- [116] D. NORTH. Introducing bdd. <http://dannorth.net/introducing-bdd>.
- [117] CARLOS PACHECO, SHUVENDU K. LAHIRI, MICHAEL D. ERNST, AND THOMAS BALL. Feedback-directed random test generation. ICSE '07. IEEE Computer Society, 2007.
- [118] FABIO PALOMBA, GABRIELE BAVOTA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, ANDREA DE LUCIA, AND DENYS POSHYVANYK. Detecting bad smells in source code using change history information. In *Automated software engineering (ASE), 2013 IEEE/ACM 28th international conference on*, pages 268–278. IEEE, 2013.

- [119] FABIO PALOMBA, GABRIELE BAVOTA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, DENYS POSHYVANYK, AND ANDREA DE LUCIA. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.
- [120] KAI PAN, XINTAO WU, AND TAO XIE. Generating program inputs for database application testing. In *ASE'11*, pages 73–82, 2011.
- [121] SEBASTIANO PANICHELLA, JAIRO APONTE, MASSIMILIANO DI PENTA, ANDRIAN MARCUS, AND GERARDO CANFORA. Mining source code descriptions from developer communications. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 63–72. IEEE, 2012.
- [122] SEBASTIANO PANICHELLA, ANNIBALE PANICHELLA, MORITZ BELLER, ANDY ZAIDMAN, AND HARALD GALL. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th ICSE*, 2016.
- [123] CHRIS PARNIN AND CARSTEN GÖRG. Improving change descriptions with change contexts. In *Proceedings of the 2008 international working conference on Mining software repositories*, pages 51–60. ACM, 2008.
- [124] J-M PETIT, JACQUES KOULOUMDJIAN, J-F BOULICAUT, AND FAROUK TOUMANI. Using queries to improve database reverse engineering. In *Entity-Relationship Approach—ER'94 Business Modelling and Re-Engineering*, pages 369–386. Springer, 1994.

- [125] J-M PETIT, FAROUK TOUMANI, J-F BOULICAUT, AND JACQUES KOULOUMDJIAN. Towards the reverse engineering of renormalized relational databases. In *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 218–227. IEEE, 1996.
- [126] RAPHAEL PHAM, YAUHENI STOLIAR, AND KURT SCHNEIDER. Automatically recommending test code examples to inexperienced developers. In *FSE'15*, pages 890–893. ACM.
- [127] DONG QIU, BIXIN LI, AND ZHENDONG SU. An empirical analysis of the co-evolution of schema and code in database applications. In *Proceedings of the FSE 2013*.
- [128] FOYZUR RAHMAN, DARYL POSNETT, ISRAEL HERRAIZ, AND PREMKUMAR DEVANBU. Sample size vs. bias in defect prediction. In *FSE'13*, pages 147–157, 2013.
- [129] SARAH RASTKAR, GAIL C MURPHY, AND GABRIEL MURRAY. Summarizing software artifacts: a case study of bug reports. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 505–514. ACM, 2010.
- [130] SARAH RASTKAR, GAIL C MURPHY, AND GLEN MURRAY. Automatic summarization of bug reports. *Software Engineering, IEEE Transactions on*, 40(4):366–380, 2014.
- [131] MEGHAN REVELLE, BOGDAN DIT, AND DENYS POSHYVANYK. Using data fusion and web mining to support feature location in software. In *Program*

- Comprehension (ICPC)*, 2010 IEEE 18th International Conference on, pages 14–23. IEEE, 2010.
- [132] PETER C RIGBY, DANIEL M GERMAN, AND MARGARET-ANNE STOREY. Open source software peer review practices: a case study of the apache server. In *Proceedings of the 30th ICSE*, pages 541–550. ACM, 2008.
- [133] JOSÉ MIGUEL ROJAS, JOSÉ CAMPOS, MATTIA VIVANTI, GORDON FRASER, AND ANDREA ARCURI. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.
- [134] JOSÉ MIGUEL ROJAS, JOSÉ CAMPOS, MATTIA VIVANTI, GORDON FRASER, AND ANDREA ARCURI. *SSBSE 2015*. Springer International Publishing, Cham, 2015.
- [135] GREGG ROTHERMEL, ROLAND H UNTCH, CHENGYUN CHU, AND MARY JEAN HARROLD. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27, 2001.
- [136] PER RUNESON. A survey of unit testing practices. *Software, IEEE*, 23(4):22–29, 2006.
- [137] SIMONE SCALABRINO, GABRIELE BAVOTA, CHRISTOPHER VENDOME, MARIO LINARES-VASQUEZ, DENYS POSHYVANYK, AND ROCCO OLIVETO. Automatically assessing code understandability: How far are we? In *32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

- [138] SIMONE SCALABRINO, MARIO LINARES-VÁSQUEZ, DENYS POSHYVANYK, AND ROCCO OLIVETO. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.
- [139] DAG SJØBERG. Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44, 1993.
- [140] MARINA SOKOLOVA AND GUY LAPALME. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009.
- [141] MARIA JOAO C SOUSA AND HELENA MENDES MOREIRA. A survey on the software maintenance process. In *ICSM'98*, pages 265–274, 1998.
- [142] GIRIPRASAD SRIDHARA, EMILY HILL, DIVYA MUPPANENI, LORI POLLOCK, AND K VIJAY-SHANKER. Towards automatically generating summary comments for java methods. In *ASE*, 2010.
- [143] PANAGIOTIS STRATIS AND AJITHA RAJAN. Test case permutation to improve execution time. In *31th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.
- [144] KUNAL TANEJA, YI ZHANG, AND TAO XIE. Moda: Automated test generation for database applications via mock objects. In *ASE'10*, pages 289–292, 2010.
- [145] HONGYIN TANG, GUOQUAN WU, JUN WEI, AND HUA ZHONG. Generating test cases to expose concurrency bugs in android applications. In

Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 648–653. ACM, 2016.

- [146] FRANK TIP. A survey of program slicing techniques. *Journal of programming languages*, 3(3):121–189, 1995.
- [147] MICHELE TUFANO, FABIO PALOMBA, GABRIELE BAVOTA, ROCCO OLIVETO, MASSIMILIANO DI PENTA, ANDREA DE LUCIA, AND DENYS POSHYVANYK. When and why your code starts to smell bad (and whether the smells go away). In *IEEE Transactions on Software Engineering (TSE)*.
- [148] MICHELE TUFANO, FABIO PALOMBA, GABRIELE BAVOTA, ROCCO OLIVETO, MASSIMILIANO DI PENTA, ANDREA DE LUCIA, AND DENYS POSHYVANYK. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.
- [149] MICHELE TUFANO, FABIO PALOMBA, GABRIELE BAVOTA, MASSIMILIANO PENTA, ROCCO OLIVETO, ANDREA LUCIA, AND DENYS POSHYVANYK. An empirical investigation into the nature of test smells. In *ASE'16*, 2016.
- [150] ARIE VAN DEURSEN AND LEON MOONEN. The video store revisited—thoughts on refactoring and testing. In *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 71–76. Citeseer, 2002.
- [151] ARIE VAN DEURSEN, LEON MOONEN, ALEX VAN DEN BERGH, AND GERARD KOK. *Refactoring test code*. CWI, 2001.

- [152] CARMINE VASSALLO, SEBASTIANO PANICHELLA, MASSIMILIANO DI PENTA, AND GERARDO CANFORA. Codes: mining source code descriptions from developers discussions. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 106–109. ACM, 2014.
- [153] XIAORAN WANG, LORI 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.
- [154] MATIAS WATERLOO, SUZETTE PERSON, AND SEBASTIAN G. ELBAUM. Test analysis: Searching for faults in tests (N). In *ASE 2015*, pages 149–154, 2015.
- [155] WESTLEY WEIMER. Generating readable unit tests for guava. In *SSBSE 2015, Bergamo, Italy, September 5-7, 2015*, volume 9275, page 235. Springer, 2015.
- [156] MARK WEISER. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
- [157] ELAINE WONG, JINQIU YANG, AND LIN TAN. Autocomment: Mining question and answer sites for automatic comment generation. In *ASE 2013 IEEE/ACM 28th International Conference on*, pages 562–567. IEEE, 2013.
- [158] BAOWEN XU, JU QIAN, XIAOFANG ZHANG, ZHONGQIANG WU, AND LIN CHEN. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

- [159] JIFENG XUAN AND MARTIN MONPERRUS. Test case purification for improving fault localization. In *FSE 2014*. ACM.
- [160] AKIHISA YAMADA, TAKASHI KITAMURA, CYRILLE ARTHO, EUN-HYE CHOI, AND ARMIN BIERE. Greedy combinatorial test case generation using unsatisfiable cores. In *ASE'16*, 2016.
- [161] ANNIE TT YING AND MARTIN P ROBILLARD. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 655–658. ACM, 2013.
- [162] ANNIE TT YING AND MARTIN P ROBILLARD. Selection and presentation practices for code example summarization. In *FSE*, pages 460–471. ACM, 2014.
- [163] ANDY ZAIDMAN, BART VAN ROMPAEY, SERGE DEMEYER, AND ARIE VAN DEURSEN. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.
- [164] BENWEN ZHANG, EMILY HILL, AND JAMES CLAUSE. Automatically generating test templates from test names. In *ASE 2015*.
- [165] BENWEN ZHANG, EMILY HILL, AND JAMES CLAUSE. Towards automatically generating descriptive names for unit tests. In *ASE'16*, 2016.