Automatically Documenting Unit Test Cases

Boyang Li, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, Nicholas A. Kraft*
College of William and Mary, Williamsburg, VA, USA
*ABB Corporate Research Center, Raleigh, NC, USA
{boyang, cvendome, mlinarev, denys}@cs.wm.edu, nicholas.a.kraft@us.abb.com

***Abstract*—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. We evaluated `UnitTestScribe` on four subject systems by means of an online survey with industrial developers and graduate students. In general, participants indicated that `UnitTestScribe` descriptions are complete, concise, and easy to read.

I. Introduction

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 Maintaining these unit tests is important to reduce the introduction of regression bugs due to outdated unit tests (i.e., unit test cases that were not updated simultaneously with the update of the particular functionality that it intends to test).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) [14] has been employed by a myriad of developers and organizations to create and expand software systems [15, 16]. 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 II) with 212 open-source and industrial developers, we found that 89.15% of the developers acknowledged that is it very important to maintain unit tests 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 [28, 44, 64].

Consequently, in this paper, 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 [30], 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 [19, 47, 57]. 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, the paper 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.

II. An Empirical Study and A Survey

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 [27, 28, 44]; 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.

A. Research Questions

We investigated the following research questions (RQs):

- **RQ$_1$** *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.
- **RQ$_2$** *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.
- **RQ$_3$** *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 (**RQ$_1$**-**RQ$_3$**) 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.

B. Data Collection

We identified all of the C# projects on GitHub through GitHub's public API [1]. 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 [7].

TABLE I
DEVELOPER SURVEY QUESTIONS AND RESULTS.

Question/Answer
Q$_1$. How often do you write unit test cases for your project(s)?
Never: 13 (6.13%), Rarely: 23 (10.85%), Sometimes: 62 (29.24%), Fairly Often: 67 (31.60%), Always: 47 (22.17%)
Q$_2$. How often do you add/write documentation comments to unit test cases? (i.e., comments preceding the unit test method declaration)
Never: 41 (19.34%), Rarely: 71 (33.49%), Sometimes: 38 (17.92%), Fairly Often: 39 (18.40%), Always: 23 (10.85%)
Q$_3$. How often do you find outdated comments (at method level) in unit test cases?
Never: 37 (17.45%), Rarely: 64 (30.19%), Sometimes: 74 (34.90%), Fairly Often: 32 (15.09%), Always: 5 (2.36%)
Q$_4$. When you make changes to the unit tests, how often do you comment the changes (or update existing comments)?
Never: 46 (21.70%), Rarely: 49 (23.11%), Sometimes: 48 (23.64%), Fairly Often: 37 (17.45%), Always: 32 (15.09%)
Q$_5$. Maintaining good unit test cases and documentations is important to the quality of a system.
Strongly Disagree: 3 (1.41%), Disagree: 2 (0,94%), Neutral: 18 (8.49%), Agree: 89 (41.98%, Strongly Agree: 100 (47.17%)
Q$_6$. 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.
Very Easy: 22 (10.38%), Easy: 62 (29.24%), Moderate: 106 (50.00%), Hard: 17 (8.02%), Very Hard: 5 (2.36%)

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 I 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 III). 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` [9] by identifying the annotations [Test], [TestMethod], [TestCase] - these annotations are used by NUnit [5] and Microsoft unit testing frameworks [4]. 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$_2$**).

C. Results

The survey questions and the results from 212 developers are summarized in Table I and the developer demographics information can be found in our online appendix [6].

RQ$_1$: Our primary interest in answering **RQ$_1$** is to understand whether developers comment unit test methods - as preceding comments or as comments inside the method. To this end, we first asked **Q$_1$** 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."

$\mathbf{Q}_3$ 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 Q_1 and Q_2, 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 $\mathbf{RQ}_1$. Although 47.17% of the developers indicated that they document unit test cases in comments, we observed that 96.44% of the projects lacked preceding comments and 85.98% lacked inner comments to document the unit tests. We also observed that 27.93% projects contained test cases despite 53.77% of developers indicating that they "fairly often" or "always" write test cases (83.01% if we consider the response "sometimes").

$\mathbf{RQ}_2$: 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 $\mathbf{Q}_3$. 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 [44]. 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 $\mathbf{Q}_4$ 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 $\mathbf{Q}_3$ 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 $\mathbf{Q}_4$ (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 $\mathbf{Q}_3$ and $\mathbf{Q}_4$, since we observe far fewer updates to unit test methods than expected from the developer survey.

Summary for $\mathbf{RQ}_2$. 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.

$\mathbf{RQ}_3$: 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 $\mathbf{Q}_5$. 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 $\mathbf{Q}_6$ indicated unit test cases should be simple or should follow the "Arrange, Act, Assert" paradigm [12], which also aids in understandability.

Summary for RQ$_3$. 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.

D. 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 [35]. 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 [6] that suggests that we have a diverse sample of open-source and industrial C# developers.

III. Approach

Based on the findings from the study (Section II), 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 that are examined through assertions in unit tests [30]. 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 IV) showed that identifying and highlighting focal methods would help developers better understand respective unit test cases (see Table VI).

```
public AddWordsSeveralTimes(){                                          1
   int listLength = 20;                                                 2
   int coocurrenceCount = 3;                                            3
   var words =                                                          4
       GenerateRandomWordList(listLength);
   for(int i = 0; i < coocurrenceCount; 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
```

Fig. 1. `CoOccurrenceMatrixTests.AddWordsSeveralTimes` unit test method of the `Sando` system

```
public ExpandMoreLetters(){                                             1
  var queries =                                                         2
      expander.GetExpandedQueries("abfdsafafdc");
  Assert.IsNotNull(queries);                                            3
  queries =                                                             4
      expander.GetExpandedQueries("bcfdasfdsad");
  Assert.IsNotNull(queries);                                            5
  queries =                                                             6
      expander.GetExpandedQueries("defdasfdsaf");
  Assert.IsNotNull(queries);                                            7
}                                                                       8
```

Fig. 2. `AcronymExpanderTests.ExpandMoreLetters` unit test method of the `Sando` system

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` [8] system (Fig. 1). 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.

TABLE II
TAXONOMY OF METHOD STEREOTYPES PROPOSED BY DRAGAN ET AL.[23] WITH OUR PROPOSED MODIFICATIONS

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)

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. 2) 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.

A. UnitTestScribe Architecture

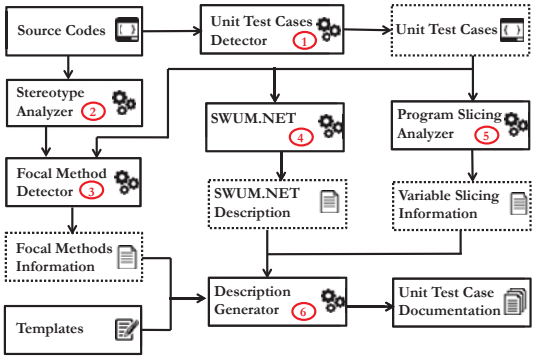

Fig. 3. UnitTestScribe Architecture. The solid arrows denote the flow of data. Numbers denote the sequence of operations.

The architecture of `UnitTestScribe` is depicted in Fig. 3. The starting point of `UnitTestScribe` is the source code of the system, including source code of the unit tests. `UnitTestScribe` analyzes the source code to identify all the unit test cases ①. Then, `UnitTestScribe` performs data-flow analysis to identify stereotypes at method level [23] 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` [11, 31] 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 [34] 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`.

B. Unit Test Detector

Our implementation focuses on systems that utilize NUnit [5] and Microsoft unit testing frameworks [4] 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).

C. Method Stereotype Analyzer

Method stereotypes are labels/categories that indicate the intent and the role of a method in a class [23], e.g., *getter*, *setter*, *collaborator*. We modified the rules proposed by Dragan *et al.* [23] for `C++` to have the corresponding stereotypes for C#. The Method Stereotype Analyzer in `UnitTestCribe` analyzes data flows provided by `SrcML.NET` [10], and then detects the stereotypes with the rules listed in Table II. 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.

D. 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 [30]. 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. [30]. 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++`,

TABLE III
A SUBSET OF PLACEHOLDER TEMPLATES WITH EXAMPLES

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⟩}.	**`globalScope.IsGlobal is true. globalScope`** is obtained from variable **`xml`** through slicing path **`xml >>> globalScope`**.
⟨Path⟩	{⟨Variable⟩ >>> }	**`xml >>> xmlElement >>> globalScope >>> actual`**

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
```

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).

E. 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 [32, 57]. In addition, programmers do not arbitrarily select names and tend to choose descriptive and meaningful names for code units [42]. UnitTestScribe relies on the SWUM approach by Hill et al. [32], in particular the SWUM.NET tool implemented by ABB in C# [11], to extract natural language phrases that are used in composing general descriptions for unit test methods.

TABLE IV
LEAF LEVEL PLACEHOLDERS

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

F. 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 [34]. 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.

G. 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;
- ⟨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 III . The placeholders ⟨...⟩ in the templates mark tokens to be replaced (the placeholders are described in Table IV) by the Description Generator. We provide a complete list of templates, placeholders, and report examples in our online appendix [6]. A description for the method in Fig. 1 generated by UnitTestScribe is shown in Fig. 4. 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.

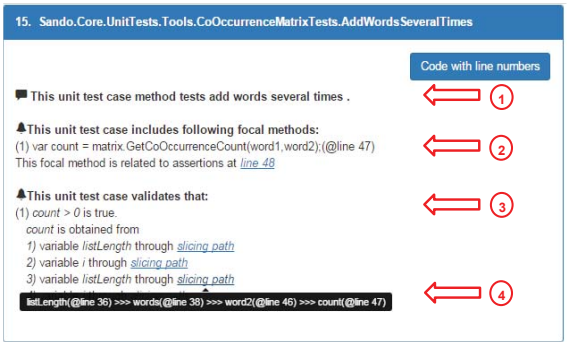

Fig. 4. An example of UnitTestScribe Description for `Sando`'s method `CoOccurrenceMatrixTests.AddWordsSeveralTimes`

IV. Empirical Study

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 [19, 47, 57]. 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.

A. 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 [10] used by ABB Corporate Research for program transformation and source code analysis; 2) the `Sando` [8] system developed by ABB Corporate Research, which is a Visual Studio Extension for searching `C`, `C++`, and `C#` projects; 3) `Glimpse` [2], which is a open-source diagnostics platform for inspecting web requests; and 4) the `Google-api-dotnet` library [3] 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 V. 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 CPU2.4GHZ 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

TABLE V
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).

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

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 II.

B. Research Questions

The RQs aimed at evaluating the three quality attributes in the evaluation framework [19, 47, 57] (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:

RQ$_4$ *How **complete** are the unit test case descriptions generated by UnitTestScribe?*

RQ$_5$ *How **concise** are the unit test case descriptions generated by UnitTestScribe?*

RQ$_6$ *How **expressive** are the unit test case descriptions generated by UnitTestScribe?*

RQ$_7$ *How important are focal methods and program slicing for understanding unit test cases?*

RQ$_8$ *How well can UnitTestScribe help developers understand unit test cases?*

C. 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 [7]. The

TABLE VI
STUDY QUESTIONS AND ANSWERS.

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**
• The description does not miss any important information	33(47.14%)	132(69.47%)
• The description misses some important information to understand the unit test case	28(40.00%)	50(26.32%)
• 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**
• The description contains no redundant/useless information	36(51.43%)	100(52.63%)
• The description contains some redundant/useless information	25(35.71%)	77(40.53%)
• The description contains a lot of redundant/useless 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**
• The description is easy to read and understand	43(61.43%)	114(60.00%)
• The description is somewhat readable and understandable	16(22.86%)	53(27.89%)
• 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**
• Yes	7(100%)	17(89%)
• No	0(0%)	2(11%)
Preferences: Identifying of slicing path would help developers to understand the unit test case	**Group 1**	**Group 2**
• Yes	6(86%)	13(68%)
• No	1(14%)	6(32%)
Preferences: Are our generated description useful for understanding the unit test cases in the system?	**Group 1**	**Group 2**
• Yes	4(57%)	17(89%)
• No	3(43%)	2(11%)

survey included (i) demographic background questions, and (ii) questions aimed at answering the RQs (Table VI 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.

D. 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.

V. 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 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.

$\mathbf{RQ}_4$ - $\mathbf{RQ}_6$ focus on three quality attributes: *completeness*, *conciseness*, and *expressiveness*. For completeness, we examined whether the descriptions of `UnitTestScribe` contain all important information ($\mathbf{RQ}_4$). For conciseness, we evaluated whether the descriptions of `UnitTestScribe` contain redundant information ($\mathbf{RQ}_5$). For expressiveness, the focus was whether the descriptions of `UnitTestScribe` are easy to read ($\mathbf{RQ}_6$). 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 $\mathbf{RQ}_7$ and $\mathbf{RQ}_8$ based on the results shown in the preferences criteria in Table VI. Generated descriptions and anonymized study results from open-source developers are publicly available at our online appendix [6].

A. 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.

B. Completeness (RQ_4)

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 IV-D.

Summary for RQ$_4$. 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.

C. Conciseness (RQ_5)

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 RQ$_5$. Overall, the results support our claim that our designed templates for the `UnitTestScribe` generate descriptions with less redundant information.

D. Expressiveness (RQ_6)

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 RQ$_6$. Overall, the results support that `UnitTestScribe` descriptions are easy to read and understand.

TABLE VII
"WHAT SE TASKS WOULD YOU USE *UnitTestScribe* DESCRIPTIONS FOR?"

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)

E. User Preferences (RQ_7 - RQ_8)

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."

Summary for RQ$_7$ and RQ$_8$. 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.

F. 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 VII. 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.

VI. Related Works

A. Approaches and studies on unit test cases

Kamimura and Murphy [36] 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 [61] 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. [53] presented an approach for automatically recommending test code examples when programmers make changes in the code. Panichella et al. [52] presented an approach for automatically generating test case summaries for JUnit test cases. Runeson [56] 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 [39, 40], prioritization [21, 55], automatic test case generation [20, 26, 29], test templates [65], data generation [41, 45]. 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.

B. Studies on automatically summarizing software artifacts

There are several related techniques for automatically summarizing and documenting different software artifacts. Sridhara et al. [57] presented an approach for automatically generating summary comments for Java methods. They demonstrated how to identify important lines of code, depending on various characteristics of methods, and convert them into NL phrases. McBurney and McMillan [46] presented a novel approach for method summarization by considering contextual information. As far as class level granularity is concerned, Moreno et al. [47, 50] focused on documenting content and responsibilities of the Java classes. Their descriptions are based on superclass, stereotypes of the class, and behavior of the blocks. Another group of the studies focused on summarizing differences between system versions [18, 19, 33, 37, 38, 43, 48]. Linares-Vásquez et al. [19, 43] implemented a tool, namely `ChangeScribe`, to automatically summarize commit messages between two arbitrary program versions. Moreno et al. [48] introduced `ARENA` for generating release nodes. Buse and Weimer [18] used symbolic execution for synthesizing documentation for program changes. Automatic summarization techniques have also been applied to exceptions [17], bug reports [54], developer discussions [51, 58, 60], loops [59] and code examples [62, 63].

C. Studies on classifying stereotypes

A program entity (method or class) stereotype reflects a high level description of the role of the program entity [23, 25]. Dragan et al. [23] first conducted an in-depth study of stereotypes at method level. They presented a well-defined taxonomy of method stereotypes. Then, Dragan et al. [25] 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. [22] presented commit level stereotypes based on the types of the changing methods/classes in the commits. Moreno and Marcus [49] 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. [24] showed that method stereotypes could be an indicator of a system's design. Moreno et al. [47, 50] utilized class stereotypes to summarize the responsibilities of classes. Linares-Vásquez et al. [19, 43] relied on commit stereotypes for generating commit messages. Abid et al. [13] 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.

VII. 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%).

Acknowledgment

We would like to acknowledge the researchers from ABB Corporate Research Center, Vinay Augustine and Patrick Francis, for their contributions to `SrcML.NET` and `SWUM.NET`. We would like to thank the anonymous reviewers for their insightful comments that helped us to significantly improve this paper. We also thank all the students, developers, and researchers who responded to our survey. This work is supported in part by the NSF CCF-1218129 and CNS-1510239 grants.

REFERENCES

[1] "GitHub API. https://developer.github.com/v3/. Last accessed: 2015/01/15."

[2] "Glimpse. https://github.com/Glimpse/Glimpse."

[3] "Google-api-dotnet. https://github.com/google/google-api-dotnet-client."

[4] "Msdn. https://msdn.microsoft.com/."

[5] "Nunit. http://nunit.org/."

[6] "Online appendix. http://www.cs.wm.edu/semeru/data/ICST16-UnitTestScribe/."

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

[8] "Sando. https://github.com/abb-iss/Sando."

[9] "Srcml. http://www.srcml.org/."

[10] "Srcml.net. https://github.com/abb-iss/SrcML.NET."

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

[12] "Unit Test Basics. https://msdn.microsoft.com/en-us/library/hh694602.aspx#BKMK_Writing_your_tests. Last accessed: 2015/10/15."

[13] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using stereotypes in the automatic generation of natural language summaries for c++ methods," in *Proc. ICSME*. IEEE, 2015, pp. 561–565.

[14] K. Beck, *Test Driven Development: By Example*, 1st ed. Addison-Wesley Professional, 2002.

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

[16] K. Beck and E. Gamma, "Test infected: Programmers love writing tests," *Java Report*, vol. 3, no. 7, pp. 37–50, 1998.

[17] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008, pp. 273–282.

[18] ——, "Automatically documenting program changes," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 33–42.

[19] L. F. Cortés-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 275–284.

[20] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE*, 2015.

[21] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*. IEEE, 2013, pp. 302–311.

[22] N. Dragan, M. L. Collard, M. Hammad, J. Maletic *et al.*, "Using stereotypes to help characterize commits," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE, 2011, pp. 520–523.

[23] N. Dragan, M. L. Collard, J. Maletic *et al.*, "Reverse engineering method stereotypes," in *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE, 2006, pp. 24–34.

[24] ——, "Using method stereotype distribution as a signature descriptor for software systems," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 567–570.

[25] ——, "Automatic identification of class stereotypes," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 2010, pp. 1–10.

[26] F. Ensan, E. Bagheri, and D. Gašević, "Evolutionary search-based test generation for software product line feature models," in *Advanced Information Systems Engineering*. Springer, 2012, pp. 613–628.

[27] B. Fluri, M. Wursch, and H. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *WCRE'07*, 2007, pp. 70–79.

[28] B. Fluri, M. Würsch, E. Giger, and H. C. Gall, "Analyzing the co-evolution of comments and source code," *Software Quality Journal*, vol. 17, no. 4, pp. 367–394, 2009.

[29] G. Fraser and A. 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*. ACM, 2011, pp. 416–419.

[30] M. Ghafari, C. Ghezzi, and K. Rubinov, "Automatically identifying focal methods under test in unit test cases," in *SCAM'15*, 2015, p. 10 pages.

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

[32] E. Hill, L. 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*. IEEE Computer Society, 2009, pp. 232–242.

[33] D. Jackson, D. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications," in *Software Maintenance, 1994. Proceedings., International Conference on*. IEEE, 1994, pp. 243–252.

[34] R. Jhala and R. Majumdar, "Path slicing," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 38–47.

[35] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 92–101. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597074

[36] M. Kamimura and G. C. Murphy, "Towards generating human-oriented summaries of unit test cases," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 215–218.

[37] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 309–319.

[38] M. Kim, D. Notkin, D. Grossman, and G. Wilson Jr, "Identifying and summarizing systematic code changes via rule inference," *Software Engineering, IEEE Transactions on*, vol. 39, no. 1, pp. 45–62, 2013.

[39] Y. Lei and J. H. Andrews, "Minimization of randomized unit test cases," in *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium*

on. IEEE, 2005, pp. 10–pp.

[40] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer, "Efficient unit test case minimization," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 417–420.

[41] B. Li, M. Grechanik, and D. Poshyvanyk, "Sanitizing and minimizing databases for software application test outsourcing," in *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014, pp. 233–242.

[42] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th annual psychology of programming workshop*, 2006.

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

[44] M. Linares-Vásquez, B. Li, C. Vendome, and D. Poshyvanyk, "How do developers document database usages in source code?" in *ASE'15 - New Ideas Track*, 2015, pp. 36–41.

[45] R. Malhotra and M. Garg, "An adequacy based test data generation technique using genetic algorithms," *Journal of information processing systems*, vol. 7, no. 2, pp. 363–384, 2011.

[46] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 279–290.

[47] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 23–32.

[48] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "Automatic generation of release notes," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 484–495.

[49] L. Moreno and A. Marcus, "Jstereocode: automatically identifying method and class stereotypes in java code," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 358–361.

[50] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Jsummarizer: An automatic generator of natural language summaries for java classes," in *Program Comprehension (ICPC), 2013 IEEE 21st International Conference on*. IEEE, 2013, pp. 230–232.

[51] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, "Mining source code descriptions from developer communications," in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 63–72.

[52] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," *PeerJ PrePrints 3:e1833 https://dx.doi.org/10.7287/peerj.preprints.1467v2*, 2015.

[53] R. Pham, Y. Stoliar, and K. Schneider, "Automatically recommending test code examples to inexperienced developers," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 890–893.

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

[55] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

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

[57] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for java methods," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 43–52.

[58] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora, "Codes: mining source code descriptions from developers discussions," in *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014, pp. 106–109.

[59] X. Wang, L. Pollock, and K. Vijay-Shanker, "Developing a model of loop actions by mining loop characteristics from a large code corpus," in *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Sep 2015.

[60] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *ASE 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 562–567.

[61] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 52–63.

[62] A. T. Ying and M. P. Robillard, "Code fragment summarization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 655–658.

[63] ——, "Selection and presentation practices for code example summarization," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 460–471.

[64] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 220–229.

[65] B. Zhang, E. Hill, and J. Clause, "Automatically generating test templates from the test names," 2015.