On Automatically Generating Commit Messages via Summarization of Source Code Changes

Luis Fernando Cortés-Coy[1], Mario Linares-Vásquez[2], Jairo Aponte[1], Denys Poshyvanyk[2]
[1] Universidad Nacional de Colombia, Bogotá, Colombia
[2]The College of William and Mary, Williamsburg, VA, USA
lfcortesco@unal.edu.co, mlinarev@cs.wm.edu, jhapontem@unal.edu.co, denys@cs.wm.edu

***Abstract*—Although version control systems allow developers to describe and explain the rationale behind code changes in commit messages, the state of practice indicates that most of the time such commit messages are either very short or even empty. In fact, in a recent study of 23K+ Java projects it has been found that only 10% of the messages are descriptive and over 66% of those messages contained fewer words as compared to a typical English sentence (i.e., 15-20 words). However, accurate and complete commit messages summarizing software changes are important to support a number of development and maintenance tasks. In this paper we present an approach, coined as *ChangeScribe*, which is designed to generate commit messages automatically from change sets. *ChangeScribe* generates natural language commit messages by taking into account commit stereotype, the type of changes (e.g., files rename, changes done only to property files), as well as the impact set of the underlying changes. We evaluated *ChangeScribe* in a survey involving 23 developers in which the participants analyzed automatically generated commit messages from real changes and compared them with commit messages written by the original developers of six open source systems. The results demonstrate that automatically generated messages by *ChangeScribe* are preferred in about 62% of the cases for large commits, and about 54% for small commits.**

***Keywords*—Commit message, summarization, code changes**

I. Introduction

Changes to software systems are stored in version control systems (VCS) such as Subversion[1] and Git[2] and are partially documented in commit messages (a.k.a., commit notes, commit comments, or commit logs). The main purpose behind commit messages is to describe the changes and help encoding rationale behind those changes. These commit messages, especially if they are correct and complete, are essential to program comprehension and software evolution in general since they help developers understand and validate changes, locate and re(assign) bug reports, and trace changes to other software artifacts.

However, the state of the practice on using and writing commit messages by actual developers seriously discords with theory. In fact, the study by Maalej and Happel [21] analyzed more than 600K+ commit messages and personal work descriptions demonstrating that 10% of the messages were removed because they were empty, had very short strings (fewer than two words) or lacked any semantical sense. Also, in another study of 23K+ projects by Dyer *et al.* [8] it has been shown that 14% of the commit messages were empty (i.e., zero words), 66% of the messages contained fewer words than a typical English sentence (i.e., 15 - 20 words) and only 10% of analyzed messages were descriptive.

One possible explanation behind this dissonance between theory and practice has been recently explored in several studies [26], [4], [20]. In particular, it has been observed that the number and nature of daily activities by software developers, including a large number of interruptions, can influence their attention to modified code [26], [4]. In fact, these daily activities become one of the causes for ignoring or forgetting implementation details behind the changes by commit time [20]. Moreover, identifying and remembering the exact set of changes done during a commit can be hard and expensive for non-trivial large changes spanning across multiple code packages, classes, methods, configuration files, database schemas, and other artifacts [4].

Regardless of exact reasons or excuses for the vast majority of unusable commit messages, them still remain an important source of information, knowledge, and documentation that developers rely on while addressing software maintenance tasks [14], [4], [11]. The main objective of a commit message is to provide information about the *what* and the *why* as related to software changes [2]. The *what* refers to the changes implemented during the incremental change while the *why* describes the motivation and context behind the changes. Although the details about the changes and changed code units can be generated automatically and accurately with line-based differencing tools, these tools do not provide enough context to understand the *why* behind the changes. Moreover, according to Buse and Weimer [2], raw diffs are not always enough as a summary for some of the *what* questions about the change, because raw diffs only report textual differences between two versions of the files, which is often long and confusing, and does not provide developers with answers to many high-level questions.

Previous approaches tried to augment some of the *what* and *why* aspects of commit messages by automatically enhancing them using visualization [4], code summarization [2], [28], and line-based differencing [3]. In addition, a recent approach by Rastkar and Murphy [31] used a multi-document summarization technique to describe the motivation behind software changes. Building on top of the previous work, in this paper we present a novel approach, coined as *ChangeScribe*, that helps developers to write descriptive commit messages. *ChangeScribe* automatically generates editable commit messages for a given change-set and describes the *what* and *why* of a change in natural language by indicating commit stereotype [6], type of changes (e.g., files rename, changes done only to properties files) and the impact set of the changes. While *ChangeScribe* integrates and extends some previously published techniques

[1]http://subversion.apache.org/
[2]http://git-scm.com/

it also offers a new way of summarizing code changes by taking into account the impact set of changes being committed. *ChangeScribe* has been instantiated to work with software applications written in Java and hosted using Git.

We evaluated *ChangeScribe* using a survey involving 23 developers, in which real commit messages from 50 commits of six open source projects (i.e., Elastic Search, Spring Social, JFreeChart, Apache Solr, Apache Felix, and Retrofit) were compared against the commit messages generated by *ChangeScribe*. The results of the user study demonstrate that 84% of the generated commit messages do not miss essential information required to understand the changes, 25% of them are concise, and in 39% of the cases the generated messages are easy to read and understand. In addition, the participants preferred *ChangeScribe*'s commit messages to those written by the original developers in 62% of cases for large commits and in 54% of the cases for small commits.

In general, this paper makes the following contributions: (i) an approach and tool, *ChangeScribe*, for automatic generation of descriptive commit messages that can reduce the amount of meaningless messages written by developers; (ii) an empirical study with 23 developers comparing *ChangeScribe*'s commit messages with those written by the original open source developers; and (iii) an open source Eclipse plug-in that implements the proposed approach and is publicly available[3].

II. Related Work

ChangeScribe is mainly related to (i) other approaches for augmenting the context provided by differencing tools, (ii) techniques for generating natural language descriptions for software artifacts, and (iii) previous studies on the characteristics of commit messages. The differences between *ChangeScribe* and the related work are listed in Table I.

A. Describing and Augmenting Context of Code Changes

Jackson and Ladd [18] introduced the *Semantic Diff* tool, which detects differences between two versions of a procedure, and then summarizes the semantic differences by using program analysis techniques. Other approaches that improve line-based differencing tools are *LDiff* by Canfora *et al.* [3] and *iDiff* by Nguyen *et al.* [27]. Parnin *et al*. [28] proposed an approach for analyzing differences between program versions at bytecode statement level; for describing the changes, type information and fully qualified source code locations of the changes (in the source entity and the entities impacted by the change) are presented. *ChangeScribe* also relies on line-based differencing, however it augments the context of the changes with a natural language description that includes the commit stereotype, change descriptions, and impact set.

Buse and Weimer [2] designed an automatic technique, *DeltaDoc*, to describe source code modifications using symbolic execution and summarization techniques. *DeltaDoc* generates textual descriptions of the changes, but when the changeset is very large (i.e. many files or methods), it describes each method separately ignoring possible dependencies of those methods. Recently, Rastkar and Murphy [31] proposed a multi-document summarization technique for describing the motivation behind a change. As compared to the approaches above, the commit messages generated by *ChangeScribe* contain more information on the *what* about the changes including information on dependencies and do not require using artifacts of multiple types.

[3]http://www.cs.wm.edu/semeru/data/SCAM14-ChangeScribe

Table I. Approaches for generating descriptions of source code changes and software artifacts. The table lists the description type, artifacts (Code Changes, Statement, Class, Method, Bug Report,Code Fragment, Crosscutting ConceRn), and techniques (Information Retrieval, Program Analysis, Software Visualization, Natural Language Processing, Stereotypes Identification, Unsupervised Learning, Supervised Learning, Impact Analysis)

Approach	Type	Artifact	Technique
Semantic Diff [18]	Abstract summary	CC	PA
Ldiff [3]	Line-diff	CC	PA
iDiff [27]	Line-diff	CC	PA
Parnin *et al*. [28]	Abstract summary	CC	PA
DeltaDoc [2]	Abstract summary	CC	PA
Rastkar and Murphy [31]	Extractive summary	CC	IR
Commit 2.0 [4]	Visual	CC	SV
Haiduc *et al.* [13]	Extractive summary	C+M	IR
Hill *et al.* [16]	Word sequences	S	NLP
Sridhara *et al*. [33]	Abstract Summary	M	NLP
Rastkar *et al.* [29], [30]	Abstract summary	CCR	PA+NLP
JSummarizer [23]	Abstract Summary	C	NLP+SI
Lotufo *et al.* [19]	Extractive summary	BR	UL
Rastkar *et al.* [32]	Extractive summary	BR	SL
Ying and Robillard [34]	Extractive summary	CF	SL
McBurney and McMillan [22]	Abstract Summary	M	NLP+IR
ChangeScribe	Abstract summary	CC	NLP+SI+IA

The code context of source code changes can be also augmented using visualizing tools. For instance, D'Ambros *et al.* [4] proposed *Commit 2.0*, a tool for augmenting commit logs with a visual context of the changes. *Commit 2.0* provides a visualization of the changes at different granularity levels, and allows developers to annotate the visualization. While *ChangeScribe* only generates a textual description, however, in the future work, a visualization like the one in *Commit 2.0* can be integrated into our proposed approach.

B. Natural Language Descriptions of Software Artifacts

Summarizing software artifacts is an active research topic in software maintenance and most of the existing techniques work mainly on source code artifacts. Haiduc *et al.* [13] proposed an approach for summarizing methods and classes as collections of the most representative terms from the source code; the terms were extracted and selected using different Information Retrieval techniques (e.g., VSM and LSI). Hill *et al.* [16] used natural language processing (NLP) techniques for generating natural language phrases (i.e., sequences of words) from source code units that are relevant to a query. Sridhara *et al*. [33] generate natural language comments for Java methods using summarization techniques; the method's comments are generated from elements in the method's signature and body, which are identified as relevant to the method behavior. Rastkar *et al.* [29], [30] described crosscutting concerns and how they were implemented in a system; the summaries contained sentences describing salient code elements (i.e., relevant to the concern), and the sentences were generated using structural and natural language information that is extracted from the source code. Moreno *et al*. [23] proposed a technique for generating summaries of Java classes in JavaDoc format composed of three parts: (i) general description explaining the objects represented by the class, (ii) class stereotype [24] description including the class responsibilities, and (iii) class behavior

description. Ying and Robillard [34] used machine learning techniques for summarizing Java code fragments. McBurney and McMillan [22] generate summaries of Java methods by including local information (keywords in the method) and contextual information (keywords in the most important referenced methods). *ChangeScribe* uses code summarization techniques based on NLP, similarly to [16], [29], [30], [23], for generating commit messages.

Other artifacts such as bug reports have been summarized by using machine learning techniques [19], [32]. For instance, Lotufo *et al.* [19] used unsupervised-learning methods, meanwhile Rastkar *et al.* [32] used supervised-learning approaches that are suitable for summarizing conversational data (e.g., email and forum discussions).

C. Empirical Studies on Characterizing Commit Messages

Few studies have mined software repositories aimed at characterizing commit messages. Alali *et al.* [1] analyzed distributions of terms in commit messages of nine open source systems. The results suggest that vocabulary terms such as fix, add, test, bug, patch are in the top ten list of most frequently used terms; the combinations file-fix, fix-use, add-bug, remove-test, and file-update are the most frequent sets. In addition, Dyer *et al.* [8] found that 14% of the commit messages from 23k+ Java projects from SourceForge are empty; only 10% of the messages are descriptive; and over 66% of the messages contain only one to fifteen words. Other studies such as [17] and [15], did not analyze the characteristics of commit messages, but used commit messages to categorize the commits in terms of the change type.

III. Generating commit messages with ChangeScribe

ChangeScribe was conceived as an approach for helping developers to generate descriptive commit messages automatically. Therefore, *ChangeScribe* is integrated with the JGit plugin[4], and the message generation process is triggered when a developer decides to commit a set of changes to the repository. Then, the commit message (automatically generated by *ChangeScribe*) is presented in an editable text area to allow developers to add rationale, include issue-ids to link the commit to a feature/issue request, and modify the message if it is required. Currently, *ChangeScribe* does not link change sets to issue tracking systems because we wanted to provide a general approach able to work when no issue trackers are available. However, future extensions of *ChangeScribe* will include the feature for linking commits to features/issues.

Our approach is aimed at summarizing code changes between two adjacent versions of a system; in addition, the messages include commit stereotypes and sentences that could help describe the motivations behind the changes. However, augmenting the context provided by diff-line based descriptions also has a drawback: large commits can generate large descriptions. We take care of this limitation by allowing developers to select the length of the message. Yet we did not base it on the number of lines or characters, because truncating the description can impact semantics of the message. Instead, we defined an impact set-based metric to show only modified classes with an impact set above certain threshold. For each class C_i in the change set (i.e., new, removed, or modified classes), the impact value is measured by the number of methods outside the class impacted by the changes to C_i (i.e., methods referencing C_i) over the total number of methods in the commit. The threshold is defined by the developer during the commit process, and allows her to control the length of the message without truncating it arbitrarily.

The process for generating commit messages (Figure 1) using *ChangeScribe* takes as input two adjacent versions (i.e., V_{i-1} and V_i) of a Java project versioned in Git. The process includes the following steps: ① extraction of source code changes for added, removed or modified types (e.g. class or interface); ② detection of method responsibilities within a class using method stereotypes; ③ characterization of the change set using commit stereotypes; ④ estimation of the impact set for the changes in the commit; ⑤ selection of the content (i.e., filtering) based on the impact-value threshold defined by the developer; and ⑥ generation of change descriptions for each modified type that exceed the impact-value threshold defined by the developer, and the general description for the commit. In the following sections, we describe the details for each one of these steps.

A. Change Extraction

We extract the change set between two adjacent versions of a Java project by using the JGit[5] library for Eclipse. For each element of the change set we identify the change type (i.e., addition, deletion or modification) and the renamed files. If a Java type (class or interface) is updated, then we identify source code changes using the *Change Distiller* tool implemented by Fluri *et al.* [10]; *Change Distiller* extracts fine-grained source code changes based on a customized tree differencing algorithm.

B. Methods and Commit Stereotype Identification

A method stereotype describes method intents and its responsibilities within the class [7]. Those responsibilities/intents can be categorized as structural, behavioral, creational, and collaborational. For instance, a creational method creates and destroys objects; structural methods are responsible for getting and setting attributes of an object; collaborational methods define the communication between objects of an application. Method stereotypes [7] of added, removed, or modified methods are used to compute the commit stereotype [6]. According to Dragan *et al.* [6] a commit is characterized by aggregating the responsibilities of added and removed methods. Therefore, commit stereotypes can provide additional information about the intention of a change.

We used *JStereoCode* implementation proposed by Moreno *et al.* [24] to identify method stereotypes by analyzing abstract syntax trees and using the rules proposed by Dragan *et al.* [7]. *ChangeScribe* uses the method stereotypes to identify commit's intent using rules proposed by Dragan *et al.* [6] (See Table II), which consider only added/removed methods; we included also the stereotypes of modified methods to characterize the commit.

[4]http://www.eclipse.org/jgit/

[5]Implementation of Git SCM in Java. http://wiki.eclipse.org/JGit/

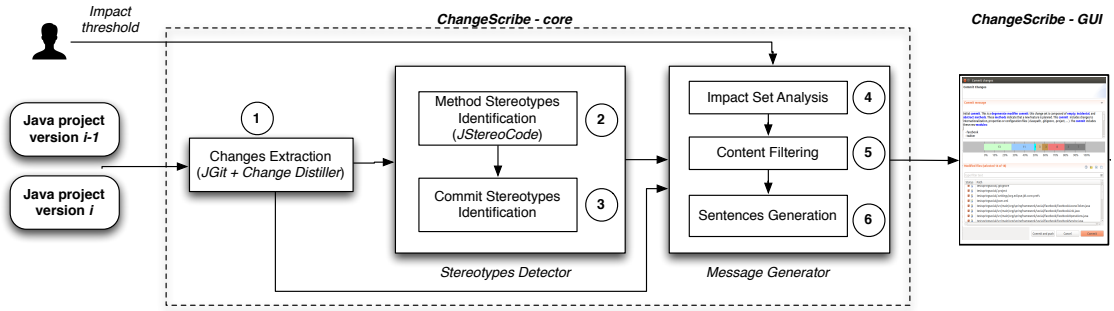

Figure 1. Architectural view of ChangeScribe

Table II. COMMIT TYPES PROPOSED BY DRAGAN *et al.* [6]

Commit type	Description	Rule
Structure modifier	Only the simple accessor and mutator, get and set, are present.	|get| + |set| ≠ 0 |methods| - (|get| + |set|) = 0
State Access modifier	Consists mostly of accessors	|accessors| > 2/3 . |methods|
State update modifier	Consists mostly of mutators	|mutators| > 2/3 . |methods|
Behavior modifier	Consists mostly of command and non-void-command methods	|non-void command| + |command| > 2/3 . |methods|
Object creation modifier	Consists mostly of factory methods	|factory| > 2/3 . |methods|
Relationship modifier	More collaborators than non-collaborators. Not all the methods are factory methods. Low number of controller methods.	|collaborators| > |non-collaborators| |factory| < 1/2 . |methods| |controller| < 1/3 . |methods|
Control modifier	Many control features Controller is present	|controller| + |factory| > 2/3 . |methods|
Large modifier	Categories of stereotypes (accessor with mutator) and (factory with controller) have to participate in distributions not in small proportions Controller or factory have to be present Number of methods in a commit is high	|accessors| + |mutators| > 1/5 . |methods| |factory| > 1/10 . |methods| ⋁ |controller| > 1/10 . |methods| |accessors| <= 1/2.|methods| ⋁ |mutators| <= 1/2 . |methods| |factory|≠0 ⋁ |controller|≠0 |methods| > average + stdev
Lazy modifier	Has to contain get/set methods It might have a large number of degenerate methods Occurrence of other stereotypes is low	|get| + |set| ≠ 0 |methods| - (|get| + |set| - |degenerate|) <= 1/3 . |methods| |degenerate| > 1/3 . |methods|
Degenerate modifier	Has at least one degenerate method	|degenerate| > 1
Small modifier	Number of methods in a class is less than 3	|methods| < 3

C. Impact Set Analysis and Content Selection

Once the commit stereotypes are identified, we filtered the content to be included in the commit message. For each class in the change set, *ChangeScribe* computes the impact value measured as the relative number of methods impacted by a class in the commit (i.e., new, removed, or modified classes). For example, the number of methods invoking a new class over the total of methods in the change set. Then, a class is included in the commit message if its impact-value is greater than or equal to the impact threshold defined by the software developer (the rationale here is to include only classes that have more impact in the change set). Table III lists two examples of commit message when the developer disables the filter (i.e., threshold equals to zero) and uses an impact threshold of 17%.

Table III. EXAMPLE OF COMMIT MESSAGES GENERATED WITH TWO DIFFERENT VALUES OF THE IMPACT THRESHOLD

ChangeScribe message without filter (Impact-value threshold = 0%)
BUG - FEATURE: <type-ID> This is a degenerate modifier commit: this change set is composed of empty, incidental, and abstract methods. These methods indicate that a new feature is planned. This change set is mainly composed of: 1. Changes to package org.springframework.social.connect.web: 1.1. Modifications to ConnectController.java: 1.1.1. Add try statement at oauth1Callback(String,NativeWebRequest) method 1.1.2. Add catch clause at oauth1Callback(String,NativeWebRequest) method 1.1.3. Add method invocation to method warn of logger object at oauth1Callback(String,NativeWebRequest) method 1.2. Modifications to ConnectControllerTest.java: 1.2.1. Modify method invocation mockMvc at oauth1Callback() method 1.2.2. Add a functionality to oauth 1 callback exception while fetching access token 2. Changes to package org.springframework.social.connect.web.test: 2.1. Add a ConnectionRepository implementation for stub connection repository. It allows to: Find all connections; Find connections; Find connections to users; Get connection; Get primary connection; Find primary connection; Add connection; Update connection; Remove connections; Remove connection Referenced by: ConnectControllerTest class

ChangeScribe with filter (Impact-value threshold = 17%)
BUG - FEATURE: <type-ID> This is a degenerate modifier commit: this change set is composed of empty, incidental, and abstract methods. These methods indicate that a new feature is planned. This change set is mainly composed of: 1. Changes to package org.springframework.social.connect.web: 1.1. Modifications to ConnectController.java: 1.1.1. Add try statement at oauth1Callback(String,NativeWebRequest) method 1.1.2. Add catch clause at oauth1Callback(String,NativeWebRequest) method 1.1.3. Add method invocation to method warn of logger object at oauth1Callback(String,NativeWebRequest) method

D. Generating Commit Messages

The message is composed of three elements: (i) tag describing whether the commit fixes a bug or implements a new feature (only for non-initial commits), (ii) general description, (iii) detailed description of the changes.

1) General Description: The general description characterizes the change set providing a general overview of the commit. It has the following parts: (i) a phrase describing whether it is an initial commit, (ii) a phrase describing commit's intent, (iii) a phrase describing class renaming, (iv) a sentence listing the new modules, (iv) a sentence indicating whether the commit includes changes to properties or internationalization files.

The commit intent is described using the commit stereotype, and the corresponding sentence is generated using the template:

> This is a <commit stereotype> : <commit stereotype description>

For example, if the change set consists mostly of factory

methods, the sentence will be "*This is a degenerate modifier commit: this change set is composed of empty, incidental, and abstract methods. These methods indicate that a new feature is planned*".

When new modules are added, we add a sentence using the following template:

> The commit includes these new modules: <module 1>, <module 2>, ..., <module n>

We consider a module as a functional unit that groups code units with the same responsibilities (i.e., a package). For example, one commit for the *Spring Social*[6] includes two new packages and this change is described by *ChangeScribe* as follows: *The commit includes these new modules: facebook, twitter*.

ChangeScribe also describes other relevant changes such as class renames by using the following sentence: *"This commit renames some files"*. In addition, when the change set includes changes to property or internationalization files, the general description includes a sentence generated with the following template:

> This commit includes changes to internationalization, property or configuration files (<file 1>, <file 2>, ... , <file n>)

For example, one commit in *Apache Solr*[7] modifies several property, configuration and internationalization files, and *ChangeScribe* describes the change as follows: *This commit includes changes to internationalization, property or configuration files (CHANGES.txt, schema-complex-phrase.xml, solrconfig-query-parser-init.xml)*

2) Detailed Description: This part of the commit message describes the changes made to each Java type (class or interface) that exceed the impact threshold defined by a developer, and the changes are organized according to packages. According to the change type, if it was an addition or deletion, our approach describes the class' goal and its relationships with other objects. Moreover, if an existing file is modified, we describe the changes for each inserted, modified and deleted code snippet.

For each class added or removed, we describe the responsibilities by extracting information from source code identifiers based on the approach by Hill *et al.* [16]. *ChangeScribe* generates noun, verb or prepositional phrases using method identifiers. For example, for the constructor with the signature `public CloudGateway(Settings, ClusterName, CloudBlobStoreService)`, *ChangeScribe* will generate the sentence *"Instantiate cloud gateway with settings, cluster name and cloud blob store service"*; for the method of the class `CloudGateway` with signature `void doStart()`, *ChangeScribe* will generate the sentence: *"Start cloud gateway"*.

ChangeScribe generates sentences for class signatures (a.k.a., class declaration) using the class stereotypes proposed by Moreno *et al.* [25]. The following template is used to generate sentences for class signatures:

> <change type> <class stereotype> <represented object>. It allows: <methods description>

For example, for the `ConstructorCodeAdapter` class responsible for data encapsulation, the sentence generated is *"Add an entity class for constructor code adapter"*; and with the class declaration `public class TwitterService implements TwitterOperations`, *ChangeScribe* generates the sentence *Add a TwitterOperations implementation for twitter service. It allows [...]*.

When the Java type is modified, *ChangeScribe* generates phrases for all changes at statement level. The *Change Distiller* tool [10] generates a list of classified changes based on the operation type (insertion, deletion or modification) and the changes to the abstract syntax tree. This information is used by *ChangeScribe* for generating sentences and describing the modified types using the text templates listed in Table IV. For example, when a new method is added, the sentence generated is *Add an additional functionality to <Object>*. But, if the method is removed, the resulting sentence is *remove functionality to <Object>*. In addition, we included context information such as the visibility, or whether the method is unused: *remove an unused functionality from <Object>*.

For each added, removed or modified type (i.e., class), a sentence is added to describe the impact of the change in two ways: (i) references to the type in the change set, and (ii) co-lateral changes triggered when a method was added to or removed from an existing class. The first case use this text template:

> Referenced by: <class name 1> class, <class name 2> class, ... , <class name n> class

For the second case, we use the text template (<operation>:= added | deleted):

> The <operation> methods triggered changes at <class name 1>, <class name 2>, ... , <class name n>

The complete commit message is created by concatenating the general description and detailed description. Table V shows a complete commit message for a change set in *Spring Social* Java project[8]. The commit updates a class with a new constructor method; this change triggered modifications to other classes (the *OAuth2ProviderSignInAccount* class) and this case is documented by *ChangeScribe*.

IV. Design of the Study

We conducted a survey to evaluate our approach. The *goal* of this study is to assess the quality of commit messages generated by *ChangeScribe*. The *context* is 50 commits from six open source projects (i.e., Elastic Search, Spring Social, JFreeChart, Apache Solr, Apache Felix, and Retrofit) hosted at GitHub, and written in Java. The commits were selected

[6] goo.gl/XzSxbu
[7] goo.gl/uokJfW
[8] http://goo.gl/a1q8Xh

Table IV. CHANGESCRIBE TEMPLATES FOR DESCRIPTIONS OF MODIFIED TYPES

Change type	Template (T) and example (E)
Add/remove functionality	T: <operation> <context information> functionality to <functionality name> E: Remove an unused functionality to rescore search source builder
Class rename	T: Rename type <old class name> with <new class name> E: Rename type InternalSettingsPerparerTests with InternalSettingsPreparerTests
Method rename	T: Rename <old method name> with <new method name> E: Rename buckets method with getBuckets
Object state rename	T: Rename <old name> object attribute with <new name> E: Rename rescore method with addRescore
Add/remove/update variable declaration	T: <operation> variable declaration statement at <method name> E: Add variable declaration statement at backgroundInvoke(Method,Object[]) method
Add/remove object state	T: <operation> (object state) <attribute name> attribute E: Add (Object state) entries attribute
Change attribute type	T: Change attribute type <old type> with <new type> E: Change attribute type RescoreBuilder with List<RescoreBuilder>
Update parent class	T: <operation> parent class <old parent class name> with <new parent class name> E: Modify the parent class DirectoryReader with FilterDirectoryReader
Add/remove parent class	T: <operation> parent class <parent class name> E: Remove parent class DirectoryReader
Update parent interface	T: Modify parent interface <parent interface name> with <new parent interface name> E: Remove parent class DirectoryReader
Add/remove parent interface	T: <operation> parent interface <parent interface name> E: Remove parent class DirectoryReader
Add/remove/update Javadoc	T: <operation> javadoc at <class/method name> class/interface/method E: Modify javadoc at Histogram interface E: Modify javadoc at rescore() method
Decrease/increase accessibility of attributes and methods	T: Decrease/Increase accessibility of <old accessibility> to <new accessibility> at <attribute or method name> attribute/method E: Decrease accessibility of protected to private at method getName()
Parameter type change	T: Type's <parameter name> change of <old type> with <new type> at <method name> method E: Type's size change of String with Long at setSize(Long size)

Table V. EXAMPLE OF *ChangeScribe*'S COMMIT MESSAGE LISTING IMPACT SET DETAILS (CLASSES IMPACTED BY A METHOD ADDITION/DELETION)

BUG - FEATURE: <type-ID>
This is a small modifier commit that does not change the system significantly.
This change set is mainly composed of:
1. Changes to package org.springframework.social.oauth2:
1.1. Modifications to AccessGrant.java:
1.1.1. Add a constructor method
The added/removed methods triggered changes to OAuth2ProviderSignInAccount class

2. Changes to package org.springframework.social.web.signin:
2.1. Modifications to OAuth2ProviderSignInAccount.java:
2.1.1. Modify arguments list when calling connect method at connect(Serializable) method

randomly while manually looking for a diverse set of commit sizes and messages, including those representing initial commits, refactorings, large commits, short commits, and commits with pseudo-messages. In terms of size-categories defined by Hattori and Lanza [15], four commits are tiny, ten are small, 17 are medium, and 19 are large. Also, our decision to use Java projects in the study is based on the fact that some of the elements in our automatically generated commit messages are built using previous techniques designed for Java projects. In addition, the projects that we selected are fairly active and mature software systems that have been used in the case studies before.

Table VI. JAVA PROJECTS HOSTED AT GITHUB AND USED IN THE STUDY. THE TABLE LISTS THE SYSTEM DESCRIPTION, TOTAL OF COMMITS AT GITHUB, NUMBER OF DEVELOPERS, AND COMMITS ANALYZED

Project	Description	Commits@GH	#Devs.	Analyzed
Elastic Search	Distributed restful search engine	7474	159	5
Spring social	Library for connecting applications with SaaS providers such as Facebook and Twitter.	1559	12	10
JFreeChart	Java chart library for professional quality charts.	323	7	10
Apache Solr	Open source enterprise search platform	10K	16	10
Apache Felix	Open source implementation of OSGI specification	10K	11	10
Retrofit	Type-safe REST client for Android and Java	666	447	5

Since *ChangeScribe* uses code summarization techniques for generating commit messages, we decided to use an evaluation framework, which was previously used for assessing automatically generated code summaries [33][23]. Therefore, the *quality focus* of the study is on the evaluation provided by real developers regarding the *content adequacy*, *conciseness*, and *expressiveness*. In addition, we wanted to understand other attributes that are important for useful commit messages as perceived by developers.

A. Research Questions

In the context of our study, we defined the following research questions:

- *RQ_1: Does the content adequacy of commit messages generated by* ChangeScribe *outperform real commit messages?*
- *RQ_2: Does the conciseness of commit messages generated by* ChangeScribe *outperform real commit messages?*
- *RQ_3: Does the expressiveness of commit messages generated by* ChangeScribe *outperform real commit messages?*
- *RQ_4: What are the attributes that describe commit messages preferred by developers?*

The first three research questions (i.e. RQ_1-RQ_3) aim at comparing real commit messages to messages generated by *ChangeScribe*, based on the three properties: content adequacy, conciseness, and expressiveness. Meanwhile, the purpose of the last research question (RQ_4) is to identify developers' preferences in terms of other attributes/properties of commit messages. For RQ_1-RQ_3, we evaluated the quality of a property in a commit message by using a 3-points Likert Scale similarly to [23]. For RQ_4, we asked the participants to select the message that they preferred (i.e., original developer's or the one by *ChangeScribe*) and write specific rationale for the choice. Table VII lists the questions that we used to evaluate each one of the research questions.

To validate the results for each property are statistically significant, when comparing the rankings of the original message *vs ChangeScribe*'s message, we used the Mann-Whitney

Table VII. SURVEY QUESTIONS AIMED AT EVALUATING MESSAGE PROPERTIES AND COLLECTING PARTICIPANT PREFERENCES

Property (RQ)	Question	Possible Answers
Content adequacy (RQ_1)	Considering only the content of the commit message and not the way it is presented, do you think that the commit message?	1) Is not missing any relevant information. 2) Is missing some information but the missing information is not necessary to understand the commit. 3) Is missing some very important information that can hinder the understanding of the commit
Conciseness (RQ_2)	Considering only the content of the commit message and not the way it is presented, do you think that the commit message?	1) Has no unnecessary information 2) Has some unnecessary information 3) Has a lot of unnecessary information
Expressiveness (RQ_3)	Considering only the content of the commit message and not the way it is presented, do you think that the commit message?	1) It is easy to read and understand 2) Is somewhat readable and understandable 3) Is hard to read and understand
Preferences (RQ_4)	When comparing both commit messages, which one do you prefer?	1) COMMENT 1 2) COMMENT 2
Preferences (RQ_4)	Why do you prefer that?	Open question

test [5] with $\alpha = 0.05$. We also computed the Cliff's delta d effect size [12] to measure the magnitude of the difference. We followed the guidelines in [12] to interpret the effect size values: negligible for $|d| < 0.147$, small for $0.147 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$ and large for $|d| \geq 0.474$.). Because we are not assuming population normality and homogeneous variances, we used non-parametric methods (Mann-Whitney test, and Cliff's delta).

B. Data Collection Process

In order to evaluate the quality of the commit messages as perceived by developers, we designed an online survey using the Qualtrics tool[9]. We asked survey participants (i.e., Java developers) to evaluate commit messages written by original developers and generated by *ChangeScribe*. For the analysis, we provided the set of changes in the commit and displayed those using GitHub's diff style. Figure 2 depicts an example of changes presented for one of the questions in the survey. We designed the survey using the following guidelines:

- The commit messages should be anonymized while presenting them to developers in order to avoid participants' bias towards any specific source. Therefore, in the survey we identified the messages as COMMENT 1 (i.e., real message) and COMMENT 2 (i.e., *ChangeScribe*) – Figure 3. In addition, instead of using links to GitHub for showing the commits, we collected the diffs and presented the changes outside of GitHub without any reference to the commits' ids or real messages (see Figure 2);
- The participants should understand the code changes before evaluating the quality of the messages. In this case, each set of questions for a particular commit started with an initial step (Figure 3, step 1), which asked a participant to provide her own commit message;
- The survey should not take more than 60 minutes to reduce the drop-out rate, and to avoid getting quick answers because of the duration of the survey. We estimated that the four steps (Figure 3) for evaluating a commit and the corresponding messages (i.e., real and *ChangeScribe*) would be done in maximum 12 minutes. Therefore, we asked participants to evaluate five commits each.

In addition to the questions in Table VII, we included questions suggested by Feigenspan *et al.* [9] to measure programming experience of the participants. The results for the demographic questions are in our online appendix.

[9]http://qualtrics.com

Figure 2. Example of code changes describing a commit. The changes are presented using a diff-based style similarly to GitHub

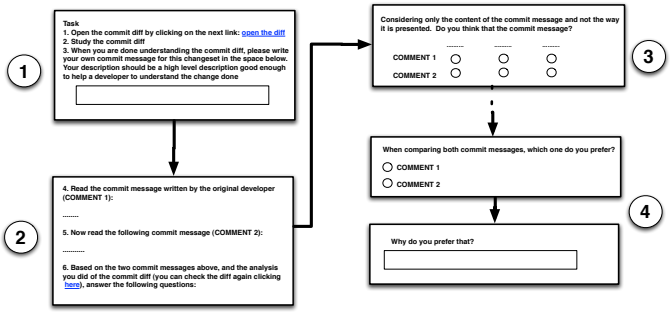

Figure 3. Example of a set of questions for a particular commit

C. Replication Package

All the experimental materials used in our study and *ChangeScribe* (Eclipse plugin) are publicly available at: http://www.cs.wm.edu/semeru/data/SCAM14-ChangeScribe. In particular we provide: (i) the links to the commits used in the study, (ii) real and *ChangeScribe* commit messages, (iii) anonymized survey's results, and (iv) instructions for installing *ChangeScribe* as an Eclipse plugin.

V. RESULTS ANALYSIS

23 participants completed the survey in which they provided 119 evaluations of the commits. In each evaluation the participant analyzed the changes in the source code; wrote their own commit message; evaluated both the commit message written by the original open source developer and the automatic commit message generated by our approach; and finally, the

Table VIII. CONTENT ADEQUACY EVALUATION OF THE ORIGINAL AND AUTOMATIC COMMIT MESSAGES

Response	Original commit messages (% ratings)	Automatic commit messages (% ratings)
Not missing any information	21	60
Missing some no essential information	38	24
Missing some essential information	40	16

Table IX. CONCISENESS EVALUATION OF THE ORIGINAL AND AUTOMATIC COMMIT MESSAGES

Response	Original commit messages (% ratings)	Automatic commit messages (% ratings)
Has no unnecessary information	86	25
Has some unnecessary information	9	48
Has a lot of unnecessary information	5	27

participant made a decision about which of the two messages she would prefer. Based on the information gathered about their background we found that all of the participants rated their knowledge of control version systems as satisfactory, good or very good, 21 of them (91%) most of the times or always wrote commit messages when contributing to a software project, only one of them had less than four years of programming experience, and 18 of them (78%) had industry experience as developers. Regarding academic degrees, ten participants were bachelors, ten were master students, and three were PhD students or had PhD degrees.

As the first step of the analysis, one of the authors evaluated the *content adequacy* of the commit messages created by the participants in order to determine whether each respondent understood the shown changes. It is worth noting that the evaluator was quite familiar with each change set included in the study, and thus, he was competent to judge this property of these commit messages. The result of this evaluation showed that 10% of the commit messages generated by the participants (12 commit messages out of the 119) did not contain correct information, and therefore, indicated a poor understanding of the changes done. We decided to discard these 12 evaluations, since understanding the changes is essential for conducting reliable and accurate assessment of the original and automatic commit messages. Thus, in the end we kept 107 evaluations.

As mentioned above, the participants were asked to evaluate both the commit messages generated by *ChangeScribe* and the commit messages written by the original developers. The properties evaluated were: *content adequacy, conciseness, and expressiveness*. *Content adequacy* judges whether the message contains all important information about the changes done. *Conciseness* assesses whether a commit message is clear and succinct or, in other words, if it does not contain superfluous and unneeded information. *Expressiveness* evaluates if a commit message is easy to read and if the way it is presented facilitates understanding of the changes done.

A. RQ_1: Content Adequacy

We consider this property as the most important one since commit messages that contain all essential information about the changes done may ease a number of maintenance tasks. The results show that only in 16% of the cases our approach generated commit messages that missed essential information. Conversely, the original commit messages miss essential information in 40% of the cases (Table VIII). In general, this result

Table X. EXPRESSIVENESS EVALUATION OF THE ORIGINAL AND AUTOMATIC COMMIT MESSAGES

Response	Original commit messages (% ratings)	Automatic commit messages (% ratings)
Is easy to read and understand	71	39
Is somewhat readable and understandable	19	44
Is hard to read and understand	10	17

indicates that the approach achieves a significant improvement in terms of relevant information needed to properly explain the changes done by the committer, and thus, its use might substantially alleviate a well-known maintenance issue. On the other hand, the results show that our approach is able to generate a commit message that includes all essential information of the changes done in 60% of the cases, while the messages written by the developers only reach this degree of completeness in 21% of the cases. From this point of view, the improvement achieved by *ChangeScribe* is also significant. In terms of statistical significance of the results, the difference is significant ($p - value = 1.543E - 08$) between the content adequacy rankings of the original messages and the messages by *ChangeScribe*; and the magnitude of the difference is large ($d = -0.9386784$).

B. RQ_2: Conciseness

The automatic commit messages generated by the tool contain a lot of superfluous and unneeded information in 27% of the cases (Table IX). Only in 25% of the cases the generated commit messages do not have any unnecessary information, while the messages written by the original developers reach this level of conciseness in 86% of the cases. These percentages indicate that, regarding this property, there is a wide margin for improvement. In terms of statistical significance, the difference is significant ($p - value < 2.2E - 16$) between the conciseness rankings of the original messages and the messages by *ChangeScribe*; and the magnitude of the difference is large ($d = 0.662866$).

Due to the format and the information included by default in the automatic commit message, this message is always longer than the original one. We found that the average length of the original commit messages is five lines, while the length of the commit message generated by our approach has 43 lines, on average. Overall, these results indicate that there is a trade-off between *content adequacy* and *conciseness*. That is why our tool allows developers to set up a threshold that controls how much information will be included in the commit message. For this study we fine-tuned this threshold having in mind that content adequacy is more important than conciseness. However, we are aware that the excess of non-essential information in the generated commit message could potentially adversely affect developers' productivity and also decrease the degree of acceptance of the tool.

C. RQ_3: Expressiveness

In our interpretation, this property was positively evaluated by the participants although the original commit messages got better scores (Table X). For instance, 17% of the automated messages were rated as hard to read and understand, while only 10% of the original commit messages got this score. At the other end of the scale is where the difference is more notorious and there is more room for improvement. There, the results

show that while the original commit messages are easy to read and understand in 71% of the cases, the automatic commit messages get this rating only in 39% of the cases. We found the difference is statistically significant ($p-value = 1.728E-05$) between the conciseness rankings of the original messages and the messages by *ChangeScribe*; and the magnitude of the difference is medium ($d = 0.3572579$). This indicates that, overall, readability and understandability of the automatic messages are acceptable.

D. RQ_4: Which messages did participants prefer? Why?

As a final question of each evaluation, we asked respondents which commit message they preferred and why. In 51% of the cases the participants preferred the commit messages generated by *ChangeScribe*.

When analyzing the reasons why respondents preferred the original message, we found that most of the times they argue that it is simpler, or shorter, or has enough information to infer the general idea and get a high level understanding of the purpose of the change. For instance, one of the participants noted: "*Even though it is not complete and misses information, it includes the reason for the commit which will allow you to understand the multiple changes that the commit includes*". In some cases, they argued that the automatic commit message explains the change step by step including details and technical information that are not truly relevant to describe the changes done at a high level. In this regard, another respondent pointed out: "*The changes made do not justify the use of a message as complex and detailed as Comment 2. Also, Comment 2 presents a large amount of unnecessary information*". Comment 2 refers to the generated message.

On the other hand, they preferred the automatic commit message mainly because it is more explanatory and more extensively covers the changes done. One of the participants noted: *"Comment 1 is easy to read, and hard to understand for someone that does not have the necessary background. / Comment 2 is very lengthy, but easy to understand, even for someone that may not be very familiar with the software. / / I would prefer to see the second comment a bit shorter ..."*. Here again Comment 2 refers to the automatic commit message while Comment 1 makes reference to the original one.

The evaluated commit messages were classified by commit size using the taxonomy proposed by Hattori and Lanza [15], but due to the size of our set of commits, instead of having four categories (tiny, small, medium, and large), we divided the set in two categories, namely small and large commits. Thus, our set has 37 large and 13 small commits. For large commits, the results show that in 62% of the cases our approach was preferred by the participants. Therefore, *ChangeScribe* clearly outperforms the original commit message when the change set includes many different changes that often require detailed and longer explanations. For small commits, the automatic commit message was preferred in 7 of the 13 cases. Those who favored the original commit messages considered that *ChangeScribe* includes unnecessary information. For instance, one of the participants noted: *"The amount of extra information provided by comment 2 just adds noise to the real purpose of commenting"*.

In summary, the participants' responses indicate that *ChangeScribe*'s messages are more detailed and longer than the original ones, so that they are able to convey more (relevant) information about the changes, in particular for large commits. That is why the *content adequacy* is the property with the highest scores. However, for being longer and wordy, these messages tend to include unnecessary information mostly for small commits. This would explain why the *conciseness* feature obtained the lowest scores. In this regard, one of the participants explained why the automatic commit messages should be preferred: *"Even when some unnecessary information is included, it is always better to have unnecessary info that you can filter rather than not having necessary information that you may need"*.

VI. Threats to Validity

This section describes the main threats to validity that can potentially affect our results and conclusions. First of all, the empirical evaluation was limited to 50 change sets from six open source systems only. The study involved 23 developers who evaluated 107 instances of the commits. Thus, it is important to notice that several variables could affect the effectiveness of the approach such as the quality of the commit messages written by the original developers and the quality of the commits itself, the problem domain, and the background of the study participants and their familiarity with the systems. In order to minimize these threats we made sure to randomly sample commit messages representing different categories. Also, we made sure that our participants had significant experience in software development and had minimal or no experience with the systems from the study. However, we realize that a more comprehensive assessment is needed in order to generalize the results.

In order to reduce the internal validity threats and maximize the reliability of the results of evaluation, we confirmed that (i) participants had adequate knowledge of version control systems, (ii) they had the habit of writing commit messages as part of their working routines, and also, (iii) the messages that they wrote reflect appropriate understanding of the changes included in each commit in evaluation. Furthermore, in all the cases, the evaluated commit messages were presented to participants anonymously to reduce bias, and the changes were presented outside GiHub to avoid references to the original commit messages. However, some learning effect may have occurred while the subjects judged the commit messages since after the first evaluation, they knew the content and format of the questions, and also, they might had been able to infer which of the two was the original commit message.

VII. Conclusion and Future Work

This paper presents an approach for generating automatic commit messages based on the code changes included in a change set. *ChangeScribe* extracts and analyzes the differences between two versions of the source code, and also, performs a commit characterization based on the stereotypes of methods modified, added and removed. The outcome is a commit message that provides an overview of the changes and classifies and describes in detail each of the changes made by a developer in the source code. Furthermore, we conducted a survey in which 23 developers performing 107 evaluations of 50 commit messages from six open source systems and equivalent number of commit messages generated

by *ChangeScribe*. According to the case study results, 84% of the generated commit messages do not miss essential information required to understand the changes, 25% of them are concise, and in 39% of the cases the generated message is easy to read and understand. The results also demonstrate that while the original commit messages miss some very important information that can hinder understanding of the changes *what* and *why* in 40% of the cases, *ChangeScribe*'s commit messages have been rated to have this deficiency in just 16% of the cases. Finally, in 51% of the cases the study participants preferred *ChangeScribe*'s commit messages to the ones written by the original developers. When considering only large commits, *ChangeScribe*'s commit messages are preferred in 62% of the cases; and when considering only small commits *ChangeScribe*'s commit messages are preferred in 54% of the cases. All in all, the evaluation indicates that *ChangeScribe* can be useful as an online assistant to aid developers in writing commit messages or to automatically generate commit messages when they do not exist or their quality is low. Moreover, the length of the message is an important attribute that should be controlled by the developer to avoid unnecessary information without truncating the message. *ChangeScribe* provides developers with a filter based on the changes impact, that reduce the size of the message without truncating the descriptions.

The evaluation also provided us with useful tips for the future work. First of all, we observed that, according to the participants, the generated messages must be shorter and more succinct. We plan on studying how we can improve these properties without affecting content adequacy. In the future we are also planning on using an improved version of the tool in a study that can help us assess the impact of our approach on real development practices in longitudinal study (including the usage of the impact set-based filter provided by *ChangeScribe*). In this context, the tool could generate an initial version of the commit message and the developer would make only minor modifications, before committing the changes.

VIII. Acknowledgements

We would like to thank developers from software development companies in Colombia, researchers from Italy and graduate students from the College of William and Mary for their help in answering the survey. This work is supported in part by the NSF CCF-1016868, NSF CCF-1218129, and NSF-1253837 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] A. Alali, H. Kagdi, and J. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC'08*, pages 182–191, 2008.

[2] R. Buse and W. Weimer. Automatically documenting program changes. In *ASE'10*, pages 33–42, 2010.

[3] G. Canfora, L. Cerulo, and M. D. Penta. Ldiff: An enhanced line differencing tool. In *ICSE'09*, pages 595 –598, 2009.

[4] M. D'Ambros, M. Lanza, and R. Robbes. Commit 2.0. In *Workshop on Web 2.0 for Software Engineering (Web2SE '10)*, pages 14–19, 2010.

[5] S. D.J. *Handbook of Parametric and Nonparametric Statistical Procedures (fourth edition)*. Chapman & All, 2007.

[6] N. Dragan, M. Collard, M. Hammad, and J. Maletic. Using stereotypes to help characterize commits. In *ICSM'11*, pages 520–523, 2011.

[7] N. Dragan, M. Collard, and J. Maletic. Reverse engineering method stereotypes. In *ICSM'06*, pages 24–34, 2006.

[8] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE'13*, pages 422–431, 2013.

[9] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring programming experience. In *ICPC'12*, pages 73–82, 2012.

[10] B. Fluri, M. Wursch, M. Pinzger, and H. Gall. Change distilling:tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725 –743, 2007.

[11] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *IWPSE 2005*, pages 113–122, 2005.

[12] R. Grissom and J. Kim. *Effect sizes for research: Univariate and multivariate applications*. Taylor and Francis, New York, NY, 2012.

[13] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *WCRE'13*, pages 35–44, 2010.

[14] A. Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance (FoSM'08)*, pages 48–57, 2008.

[15] L. P. Hattori and M. Lanza. On the nature of commits. In *ASE'08*, pages 63–71, 2008.

[16] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *ICSE'09*, pages 232–242, 2009.

[17] A. Hindle, D. German, , and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR'08*, pages 99–108, 2008.

[18] D. Jackson and D. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *ICSM'94*, pages 243–252, 1994.

[19] R. Lotufo, Z. Malik, and K. Czarnecki. Modelling the hurried bug report reading process to summarize bug reports. In *ICSM'12*, pages 430–439, 2012.

[20] W. Maalej and H. Happel. From work to word: How do software developers describe their work? In *MSR'09*, pages 121–130, 2009.

[21] W. Maalej and H. Happel. Can development work describe itself? In *MSR'10*, pages 191–200, 2010.

[22] P. W. McBurney and C. McMillan. Automatic documentation generation via source code summarization of method context. In *ICPC'14*, page to appear, 2014.

[23] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC'13*, pages 23–32, 2013.

[24] L. Moreno and A. Marcus. Jstereocode: automatically identifying method and class stereotypes in java code. In *ASE'12*, pages 358–361, 2012.

[25] L. Moreno, A. Marcus, L. Pollock, and K. Vijay-Shanker. Jsummarizer: An automatic generator of natural language summaries for java classes. *ICPC'13 - formal tool demonstration*, pages 230–232, 2013.

[26] G. Murphy. Attacking information overload in software development. In *VL/HCC'09*, page 4, 2009.

[27] H. A. Nguyen, T. T. Nguyen, H. V. Nguyen, and T. N. Nguyen. idiff: Interaction-based program differencing tool. In *ASE'11*, pages 575–575, 2011.

[28] C. Parnin and C. Gorg. Improving change descriptions with change contexts. In *MSR'08*, pages 51–60, 2008.

[29] S. Rastkar. Summarizing software concerns. In *ICSE'10*, pages 527–528, 2010.

[30] S. Rastkar, G. Murphy, and A. Bradley. Generating natural language summaries for crosscutting source code concerns. In *ICSM'11*, pages 103–112, 2011.

[31] S. Rastkar and G. C. Murphy. Why did this code change? In *ICSE'13*, pages 1193–1196, 2013.

[32] S. Rastkar, G. C. Murphy, and G. Murray. Automatic summarization of bug reports. *IEEE Trans. Software Eng*, 40(4):366–380, 2014.

[33] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *ASE'10*, pages 43–52, 2010.

[34] A. T. Ying and M. P. Robillard. Code fragment summarization. In *ESEC/FSE'13*, 2013.