Prototype of a tool for automatic generation of commit messages for Java applications

Luis Fernando Cortés Coy

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingenieria de Sistemas e Industrial
Bogotá, Colombia
2014

Prototype of a tool for automatic generation of commit messages for Java applications

Luis Fernando Cortés Coy

A thesis submitted in partial fulfillment of the requirements for the degree of:
Maestría en ingeniería - Ingeniería de sistemas y computación

Advisor:
Jairo Aponte, Ph.D
Co-Advisor:
Mario Linares-Vásquez, Ph.D candidate

Research line
Software Engineering
Research group
ColSWE: Software Engineering Research Group

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingenieria de Sistemas e Industrial
Bogotá, Colombia
2014

Dedication

This work is dedicated to my parents, brothers, grandparents, who with their love and teachings have allowed me to get big wins. To my love Gina Vargas who encouraged and advised me until the end.

Acknowledgments

I am grateful to my advisors Mario Linares and Jairo Aponte for their support and help that I allowed reaching the dissertation goal. Their knowledge, advise and ideas were major factor to perform this thesis.

I express my grateful to the members of ColSWE research group from Universidad Nacional de Colombia and SEMERU research group from The College of William and Mary for their time and feedback given to complete this work successfully.

To my parents, brothers, friends and my love Gina Vargas thanks for helping me unconditionally, for their patience and advise.

Abstract

Title in English

Prototype of a tool for automatic generation of commit messages for Java applications.

Título en español

Prototipo de una herramienta para la generación automática de recomendaciones de comentarios de commit en aplicaciones Java.

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. However, accurate and complete commit messages summarizing software changes are important to support a number of development and maintenance tasks. This thesis presents 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. This work presents the evaluation of *ChangeScribe* in an evaluative 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.

Resumen

Aunque los sistemas de control de versiones le permiten a los desarrolladores de software describir y explicar las razones por la cuales modificaron el código fuente utilizando un mensaje en el commit, en la práctica estos mensajes son muy cortos o incluso vacíos. De hecho, en recientes estudios de 23K+ de proyectos Java se ha encontrado que el 10% de los mensajes son descriptivos y alrededor del 66% de estos contienen pocas palabras comparado con el tamaño promedio de una oración escrita en el idioma inglés. Sin embargo, resumir los cambios en el software de una manera precisa y completa es muy importante para

apoyar las tareas que se realizan en el desarrollo y mantenimiento de un software. Este trabajo presenta *ChangeScribe* un prototipo para generar mensajes de commit usando lenguaje natural y teniendo en cuenta el estereotipo del commit, el tipo de cambio (rename de un archivo, cambios a archivos de propiedades, etc), y también el conjunto de impacto de los cambios realizados. De otro lado, presenta la evaluación de *ChangeScribe* en un estudio de usuarios que involucró 23 desarrolladores de software que analizaron los mensajes de commit generados automáticamente por *ChangeScribe* y los mensajes de commit escritos por los desarrolladores originales de seis sistemas open source. Los resultados demuestran que los mensajes generados de forma automática por *ChangeScribe* son preferidos en cerca del 62% de los casos en commits largos, y en cerca de 54% de los casos en commits cortos (pocas modificaciones).

Keywords
Commit message, software summarization, code changes

Palabras clave
Mensaje de commit, Software summarization, cambios al codigo fuente

Contents

List of Figures

List of Tables

1. Introduction

1.1. Background and Justification

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 [1] 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+ software systems by Dyer *et al.* [2] it has been shown that 14% of the commit messages were virtually empty, 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 [3–5]. 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 [3, 4]. In fact, these daily activities become one of the causes for ignoring or forgetting implementation details behind the changes by commit time [5]. 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, commit messages still remain as an important source of information, knowledge, and docu-

[1] `http://subversion.apache.org/`
[2] Website of Git SCM `http://git-scm.com/`

mentation that developers rely on while addressing software maintenance tasks [4,6,7]. The main objective of a commit message is to provide information about the *what* and the *why* as related to software changes [8]. The *what* refers to the changes implemented during the incremental change while the *why* describes the motivation and context behind the changes. Although the *what* 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 [8], 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 [8], [9], and line-based differencing [10]. In addition, a recent approach by Rastkar and Murphy [11] used a multi-document summarization technique to describe the motivation behind software changes.

1.2. Problem Definition

Software applications are modified constantly, and changes are stored in Version Control Systems (VCS). Systems as Subversion, Mercurial, or Git are historical repositories of software [6], and these allow different actions during the development process. The commit is one of those actions because the commit is the integration of the source code changes into a version control system. Moreover, regardless to the VCS used by the development team, VCS allow developers to write a message with a textual description of the source code modifications.

The software development community uses the information of a commit in different ways. For instance, to understand the parts of the system changed and the reasons for these modifications. Also, this information allows software developers to share the modifications and synchronize their version of the source code with the modifications made by other developers. As well, this information is raw data useful for research in software engineering, specifically in software evolution, software maintenance and mining software repositories, in order to understand and improve the development of systems from the history of artifacts generated during this process. For instance, Sunghun *et al.* [12] used commit annotations together with the descriptions of the Bug Tracking system Bugzilla[3] to predict whether a

[3]Bug tracker or Bug Tracking System is an information system for recording and tracking of bugs and supports the software developer with the quality assurance of the software projects.

change can introduce bugs in the system. Bachmann *et al.* [13] also used software repositories such as SVN and Bugzilla to create a tool that allowed to link a bug report with its own correction in the source code. The process involves the analysis of the information contained in the bug report and the commit log. Also, if the descriptions of the changes and the bugs have high quality, i.e., the information of changes is useful to understand the modifications applied to the source code. Alonso [14] shows a method to determine the expertise of a developer in a software project using text mining techniques on CVS logs.

Commit messages should contain descriptive information about the changes to the source code, and this information should describe which parts of the source code changed, and the reason for these for each one of the artifacts involved in the commit. According to D'Ambros *et al.* [4] in the Eclipse project, 20% of the commits do not have a comment about the changes and also the Vuze project has the same problem with around of 5300 of 13000 commits. Therefore, based on this numbers is reasonable to use tools such as Unix diff [15] to extract the textual differences between versions of source code, and thus to make a general model of the changes. This result can broaden the context within which changes were done. However, this can not change the commit comment written by software developer and the diff result can be too long confusing for the developer [8].

Studies as [13] [12] [14] [16] assumes that the commit messages of the commits contains a detailed description about the changes applied to a set of files. Usually, this description does not provides information about the *what* and the *why* as related to software changes, and both researchers and software developers are forced to work with information that is not useful because the commit message does not descriptive or even sometimes this is empty. For instance, in the repository of *Mozilla* project was found the following commit message *"Bug 494847 - Kill MTBF, r = dbaron"*, similarly in the repository of *Netbeans* project *"The same issue as in # 41049"*. Other example, can be found in the *Macports-dev* project a developer commit changes with the following description *"Going forward, could you I ask you to be more descriptive in your commit messages? Ideally you should state what you've changed and also why (unless it's obvious) ...."*[4].

Then, empty or non-descriptive commit messages hinder the evolution and maintenance tasks with regard to the understanding of previous changes increasing the complexity to maintain the software, the time to do a modification, and also increasing project costs. Therefore, worthwhile to propose a solution that help developers to generate automatically commit messages describing the WHAT and the WHY of the commit.

[4] `http://lists.macosforge.org/pipermail/macports-dev/2009-June/008881.html` (Verified may 24, 2014)

1.3. Contributions

This thesis presents a novel approach, coined as *ChangeScribe*, that automatically generates commit messages for a given change-set between two adjacent versions of a Java system and describes the *what* and *why* of a change in natural language by indicating commit stereotype [17], type of changes (e.g., files rename, changes done only to properties files) and the impact set of the changes. While *ChangeScribe* integrates some previously published techniques it also offers a new way of summarizing code changes by taking into account the impact set of changes being committed; the impact-value threshold is defined by the developer and allows to filter the content with change-set is large. *ChangeScribe* has been instantiated to work with software applications written in Java and hosted using Git as an underlying VCS. In general, this thesis makes the following contributions:

- An approach and tool, *ChangeScribe*, for automatic generation of commit messages for Java applications hosted in Git repositories;
- An empirical study with 23 developers comparing *ChangeScribe*'s commit messages with those written by the original open source developers; and
- An open source Eclipse plug-in that implements the proposed approach and is publicly available[5].

1.4. Thesis Organization

The remainder of this thesis provides the following content: *Chapter 2* reviews related work in the field of software summarization, specifically on approaches that improve the description of source code changes using visualization or natural language descriptions. Also, on commit characterization. *Chapter 3* presents *ChangeScribe*, our approach for automatic generation of commit messages for Java projects hosted on Git. *Chapter 4* presents the evaluative survey applied to 23 software developers. *Chaper 5* shows the *ChangeScribe* architecture and some implementation details. Finally, some conclusions and future work are discussed in chapter *Chapter 6*.

1.5. Bibliographical Notes

This section reports that parts of this thesis have been previously published in collaboration with other researchers in the 14th IEEE International Working Conference on Source

[5] `http://www.cs.wm.edu/semeru/data/ICSME14-ChangeScribe`

Code Analysis and Manipulation (SCAM 2014), which was held in Victoria, Canada from September 28th to 28th. These parts have been used with the permission of the co-authors.

2. Related work

Automatic generation of commit messages 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 **2-1**.

Table 2-1.: Approaches for generating descriptions of source code changes and software artifacts. The table lists the description type, artifacts (**C**ode **C**hanges, **S**tatement, **C**lass, **M**ethod, **B**ug **R**eport,**C**ode **F**ragment, **C**rosscutting **C**once**R**n), and techniques (**I**nformation **R**etrieval, **P**rogram **A**nalysis, **S**oftware **V**isualization, **N**atural **L**anguage **P**rocessing, **S**tereotypes **I**dentification, **U**nsupervised **L**earning, Supervised **L**earning, **I**mpact **A**nalysis)

Approach	Type	Artifact	Technique
Semantic Diff [18]	Abstract summary	CC	PA
Ldiff [10]	Line-diff	CC	PA
iDiff [19]	Line-diff	CC	PA
Parnin *et al.* [9]	Abstract summary	CC	PA
DeltaDoc [8]	Abstract summary	CC	PA
Rastkar and Murphy [11]	Extractive summary	CC	IR
Commit 2.0 [4]	Visual	CC	SV
Haiduc *et al.* [20]	Extractive summary	C+M	IR
Hill *et al.* [21]	Word sequences	S	NLP
Sridhara *et al.* [22]	Abstract Summary	M	NLP
Rastkar *et al.* [23, 24]	Abstract summary	CCR	PA+NLP
JSummarizer [16]	Abstract Summary	C	NLP+SI
Lotufo *et al.* [25]	Extractive summary	BR	UL
Rastkar *et al.* [26]	Extractive summary	BR	SL
Ying and Robillard [27]	Extractive summary	CF	SL
McBurney and McMillan [28]	Abstract Summary	M	NLP+IR
ChangeScribe	Abstract summary	CC	NLP+SI+IA

2.1. Describing and Augmenting Context of Code Changes

Jackson and Ladd [18] introduced *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.* [10] and *iDiff* by Nguyen *et al.* [19]. Parnin *et al.* [9] proposed an approach for analysing differences between program versions at byte code 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 [8] 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 change-set is very large (i.e. many files or methods), it describes each method separately ignoring possible dependencies of those methods. *DeltaDoc* takes as input two revisions of a software and generates a commit message based only on modified methods ignoring added and removed methods. As the first step *DeltaDoc* computes the conditions under which a path is executed. Then, as the second step the documentation is generated describing the effects of the modifications on behaviour of the software. As the third step, the documentation is summarized to reduce the size because the message can be as long as Unix diff. The summarization process loses context information because this does not preserve the source code semantics. The last step, the content is filtered where only the relevant statements such (i.e. return and throw statements) are retained and other as assignments to local variables are not. The process implemented by *DeltaDoc* is depicted in the Figure **2-1**.

Recently, Rastkar and Murphy [11] proposed a multi-document summarization technique for describing the motivation behind a change. The main concept of this approach is based on the extraction and generation of information of related documents to the source code such as feature requests, emails and bug reports. In the first step of the summarization process a set of relevant documents is constructed. Once the documents are selected, they identify the most important sentences to form the summary. Finally, they filter the content using a proposed ranking based on eight relevant features, documents adjacency, sentence frequency, sentence similarity, sentence similarity to the document title, sentence position in the document, and the sentence length, among others.

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. *Commit 2.0* only considers structural changes as additions, deletions of instance variables, the change annotations are sent to software repository as usual, but the visualizations are sent to a blog service called Posterous [1]. In *Commit 2.0*, the Java packages are represented as rectangles, and the classes are viewed as rectangles within corresponding package. In the rectangle representing classes, the width is proportional to the number of attributes, and the height to the number of methods, but the size of packages means nothing. This approach enables to software developer a different way to view and comment the changes, but the problem of reducing the effort and time is not resolved. The Figure **2-2** shows *Commit 2.0 output*.

As compared to the approaches above, *DeltaDoc* contains information about the *what* of the change, in the Rastkar and Murphy approach the importance is for the *why* of the change. In our approach, the commit messages generated contain more information on the *what* about the changes including information on dependencies and do not require using artifacts of multiple types. While *ChangeScribe* only generates a textual description, however, in the future work, visualization like the one in *Commit 2.0* can be integrated into our proposed approach.

2.2. 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.* [20] 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) and leading summaries. The most relevant result obtained with this approach was that the summaries formed with leading terms of classes and methods and terms extracted with VSM are the most pertinent for software developers. Also, for the summarization techniques based on text retrieval, VSM obtained higher scores for relevant information evaluation because VSM favours terms with high frequency in the documents. Regarding to summary size the evaluators prefer summaries with 10 relevant terms and not with 5 relevant terms because the first ones contains more relevant information than shorten alternative.

Hill *et al.* [21] 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

[1] Website of Posterous https://posterous.com/ (Verified may 29 of 2014)

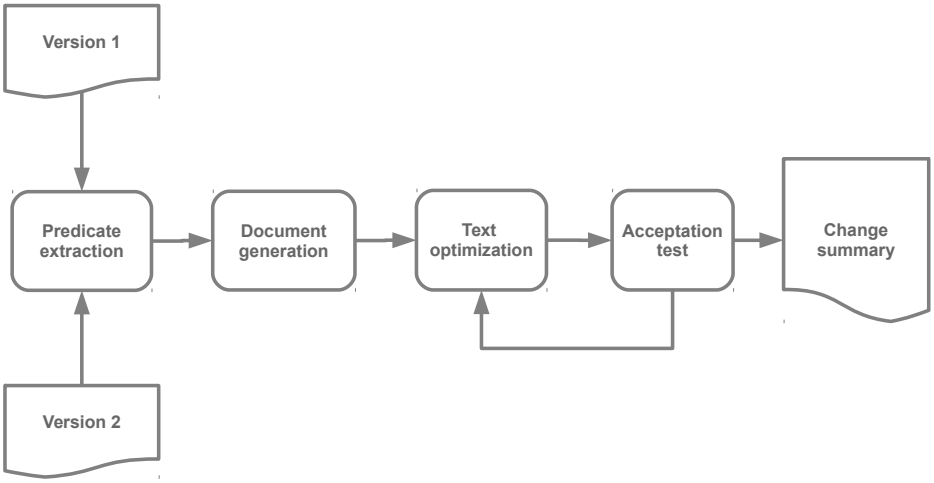


Figure 2-1.: Architectural view of DeltaDoc

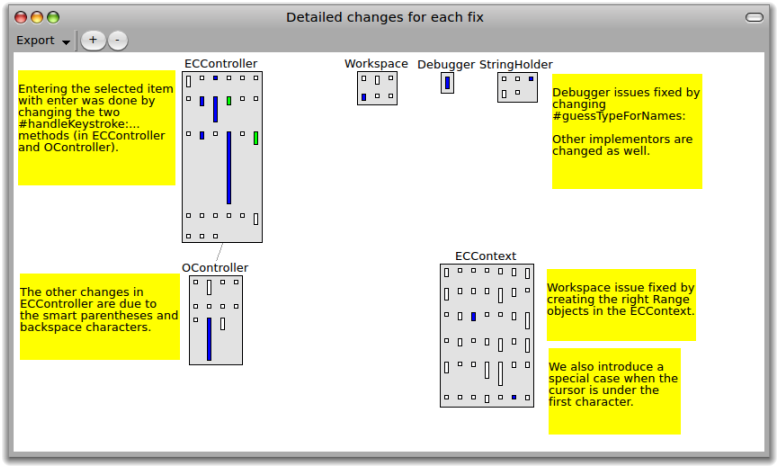


Figure 2-2.: Example of Commit 2.0 output [4]

query. In this approach proposes an algorithm to extract information of source code identifier using verb, noun and prepositional phrases. The process implemented by Hill is composed of four steps: (i) splitting source code identifiers. (ii) computing the treatment (verb, noun or prepositional phrase) to all fields and methods of a class. (iii) identifying verb, direct and indirect objects, and preposition. (iv) inferring arguments to generate additional phrases.

Sridhara *et al* [22] proposed generating 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 behaviour. This approach assumes that no comment of classes and methods are present because this documentation frequently is outdated. The automatic summary generation is composed of three components: (i) the content selection to be included in the summary. (ii) natural language generation to describe the content of the Java type. (iii) combining the generated sentences. This work uses a novel approach that capture the conceptual knowledge of the software developers using linguistic information of the source code and the language semantic, this model is know as Software Word Usage Model (SWUM).

Rastkar *et al.* [23, 24] described cross-cutting 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. The proposed approach is composed by the following steps: (i) extract structural and natural language facts from the source code represented as an ontology (ii) apply a set of heuristics to the previous ontology to find patterns and salient code elements (iii) generate sentences using the previous information (patterns, salient code elements, information from the ontology) and using defined templates. The Figure **2-3** shows an example of generated summary.

Moreno *et al.* [16] 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 [29] description including the class responsibilities, and (iii) class behaviour description. The summary generation process is composed of the following steps: (i) method and class stereotype identification, (ii) content selection based on two proposed heuristics (using the stereotypes and the access level of Java types), and (iii) generation of a readable text describing the class using the JavaDoc format. An example of a generated summary with this approach is showed in Table **2-2**.

Ying and Robillard [27] used machine learning techniques for summarizing Java code fragments. McBurney and McMillan [28] generate summaries of Java methods by including local information (keywords in the method) and contextual information (keywords in the most important referenced methods). Other artifacts such as bug reports have been summarized by

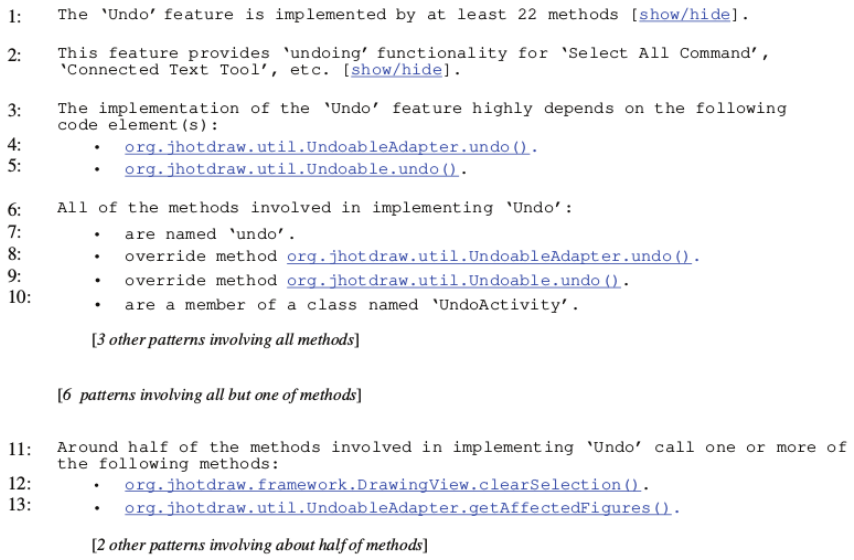


Figure 2-3.: Example of output of Rastkar approach [24]

Table 2-2.: Summary of a Java class generated by [16]

An AbstractPlayer extension for m player handlers. This entity class consists mostly of mutators to the m player handler's state. It allows managing: - mute; - volume; and - next with no gap. Also allows: - finishing m player handler; - handling next; - playing audio file f; - stopping m player handler; - playing m player handler; and - handling previous.

using machine learning techniques [25,26]. For instance, Lotufo *et al.* [25] used unsupervised-

learning methods, meanwhile Rastkar *et al.* [26] used supervised-learning approaches that are suitable for summarizing conversational data (e.g., email and forum discussions).

2.3. Empirical Studies on Characterizing Commit Messages

Few studies have mined software repositories aimed at characterizing commit messages. Alali *et al.* [30] analysed 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 order to characterize the commits with respect to the number of files and lines the results show that 75% of commits are small in all cases (lines and files). The largest commits do not happen frequently, for instance when the license is updated in each file or modifying a large class. In this approach, a change is considered small when the developer modifies between 0 and 5 lines of the source code or when modifies only 1 file. In addition, Dyer *et al.* [2] 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 [31] and [32], did not analyse the characteristics of commit messages, but used commit messages to categorize the commits in terms of the change type. For example, Hindle *et al.* [31] proposed a commit classification for large commits based on the commit intention, for example if the commit fix a bug, add a module, modify the legal information, among others. Similarly, Hattori *et al.* [32] proposed a classification based on the commit size and content of the commit messages. The proposed classification based on the size is showed in the Table **2-3**. This classification is used in this work as reference to study the performance of *ChangeScribe* for different commit size.

Table 2-3.: Commit types proposed by Hattori [32]

Commit type	Size (number of files)
Tiny	1 to 5
Small	6 to 25
Medium	25 to 125
Large	up to 125

3. The approach: Generating commit messages for Java applications

ChangeScribe was conceived as an approach for helping developers to generate commit messages automatically. Therefore, *ChangeScribe* is integrated with the JGit plug-in[1], 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 and include issue-ids to link the commit to a feature/issue request. 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 in the change set, the impact value is measured by the number of its methods impacted by the change (i.e., class added, class removed, class modified) 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 **3-1**) 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

[1] Website of JGit project `http://www.eclipse.org/jgit/`

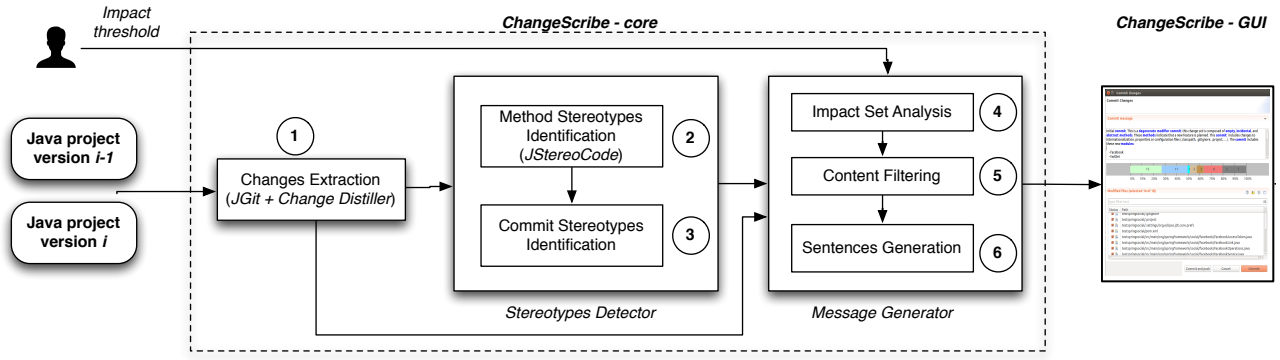


Figure 3-1.: Architectural view of ChangeScribe

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.

3.1. Change Extraction

We extract the change set between two adjacent versions of a Java project by using the JGit[2] 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.* [33]; *Change Distiller* extracts fine-grained source code changes based on a customized tree differencing algorithm.

3.2. Method and Commit Stereotype Identification

A method stereotype describes method intents and its responsibilities within the class [34]. 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 of getting and setting attributes of an object; collaborational methods define the communication between objects of an application. We used *JStereoCode* implementation proposed by Moreno *et al.* [29] to identify method stereotypes by analysing

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

abstract syntax trees and using the rules proposed by Dragan *et al.* [34]. Table 3.2 lists method stereotypes identified by *JStereoCode* tool.

Table 3-1.: Method stereotypes identified by *JStereoCode* [29]

Category		Method stereotype	Description
Structural	Accessor	Get	Returns a local field directly
		Predicate	Returns a Boolean value that is not a local field
		Property	Returns information about local fields
		Void accessor	Returns information about local fields through the parameters
	Mutator	Set	Changes only one local field
		Command	Changes more than one local fields
		Non-void command	Command whose return type is not void or Boolean
Creational		Constructor	Invoked when creating an object
		Destructor	Performs any necessary clean-ups before the object is destroyed
		Copy constructor	Creates a new object as a copy of the existing one
		Factory	Instantiates an object and returns it
Collaborational		Collaborator	Connects one object with other type of objects
		Controller	Provides control logic by invoking only external methods
		Local controller	Provides control logic by invoking only local methods
Degenerate		Abstract	Has no body
		Empty	Has no statements
		Incidental	Any other case

Method stereotypes [34] of added, removed, or modified methods are used to compute the commit stereotype [17]. According to Dragan *et al.* [17] a commit is characterized by aggregating the responsibilities of added and removed methods. Therefore, commit stereotypes can provide information about the intention of a change because it provides information about the types of design changes were performed to the software system in a commit. *ChangeScribe* uses the method stereotypes to identify commit's intent using rules proposed by Dragan *et al.* [17] (See Table **3-2**), which consider only added/removed methods; we included also the stereotypes of modified methods to characterize the commit.

Table 3-2.: Commit types proposed by Dragan *et al.* [17]

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

The commit signature proposed by Dragan *et al.* [17] refers to the distribution of method stereotypes that are added and removed methods, considering this signature as a graphic

representation of the commit stereotype. Based on the above, the commit signature describes the structural complexity of a source code change. For instance, in the Spring Social change [3], the commit stereotype for this change-set is *object creation modifier commit*, it allows developer to understand that those modifications to the software system gains more creational features because the modified types contains many factory methods (2 constructor method, 5 factory method and 1 controller method), the Figure **3-2** depicts the commit signature.

In *ChangeScribe* for each change-set the commit stereotype is used in the generated commit message and the commit signature is a visual element for augmenting the context of the source code changes when the developer is doing the commit, this signature is not available after doing the commit of the changes.

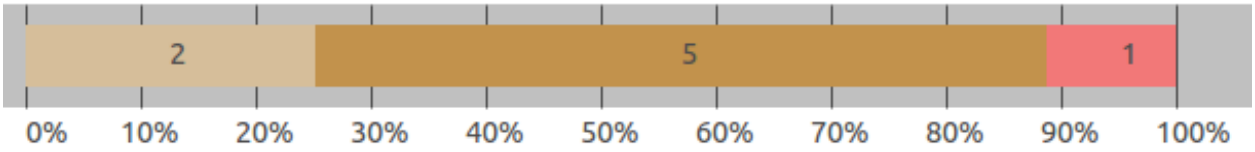


Figure 3-2.: Example of the commit signature of spring social commit

3.3. 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 change in the commit. 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 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 **3-3** lists two examples of commit message when the developer disables the filter and uses an impact threshold of 17%.

3.4. Generating Commit Messages

The message is composed of three elements: (i) tag (only for non-initial commits), (ii) general description, (iii) detailed description of the changes.

[3] `http://goo.gl/xyp3cS`

3.4.1. 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 this change-set 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 below:

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*[4] 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>)

[4] goo.gl/XzSxbu

For example, one commit in *Apache Solr*[5] 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).*

3.4.2. 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.* [21]. *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.* [35]. 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 [...].*

[5] `goo.gl/uokJfW`

Table 3-3.: Example of commit messages generated with two different values of the impact threshold

ChangeScribe message without filter
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 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

When the Java type is modified, *ChangeScribe* generates phrases for all changes at statement level. The *Change Distiller* tool [33] 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 **3-4**. 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>*.

Table 3-4.: 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)

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 uses the text template below:

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 **3-5** shows a complete commit message for a change set in *Spring Social* Java project[6]. Table **3-6** shows the complete commit message for another change set[7], which 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*. Other examples of commit messages generated with *ChangeScribe* are listed in Appendix A.

Table 3-5.: Example of ChangeScribe's commit message, which includes details of references to added/deleted classes

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

[6] `http://goo.gl/a1q8Xh`
[7] `http://goo.gl/ch4PZK`

Table 3-6.: Example of ChangeScribe's commit message, which includes details of 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

3.5. Availability

ChangeScribe is built as an Eclipse plug-in and released under Eclipse Public License, EPL [8]. The most recent version of the plug-in is available to download here[9]. *ChangeScribe* can be installed on Windows, Linux and Mac. Before installing ChangeScribe you should: install Java 7+ and download Eclipse Juno 4.2+. The plug-in provides the following features: (i) Automatic generation of commit messages. (ii) A visualization of the method's stereotype distribution in the commit, namely commit signature by Dragan *et al.* [17]. (iii) An help button describing commits/methods stereotypes. (iv) *ChangeScribe* is integrated to the commit mechanism: it allows to select the classes to be committed, and has a button for committing the code and the message to the Git repository. (v) It has a settings window to enable the stereotypes visualization, set the name of the committer/author, and set the threshold for the impact filter when the commit is large (contains several files, for instance when change-set is an initial commit). The Figure **3-3** depicts the plug-in user interface.

The help button shows a window with the explanation of the commit/method stereotypes. This guide explains each commit stereotype and shows the color used to represent in the commit signature, as can be seen in the Figure **3-4**-a. In the case of method stereotypes, the help guide shows a table with all descriptions of the method stereotypes.

The describe button is responsible to launch the *ChangeScribe* message generator using the

[8]Eclipse Public License definition: `http://www.eclipse.org/legal/epl-v10.html` (Verified May 24 of 2014)

[9]Website of ChangeScribe update site `http://www.cs.wm.edu/semeru/changescribe/update/`

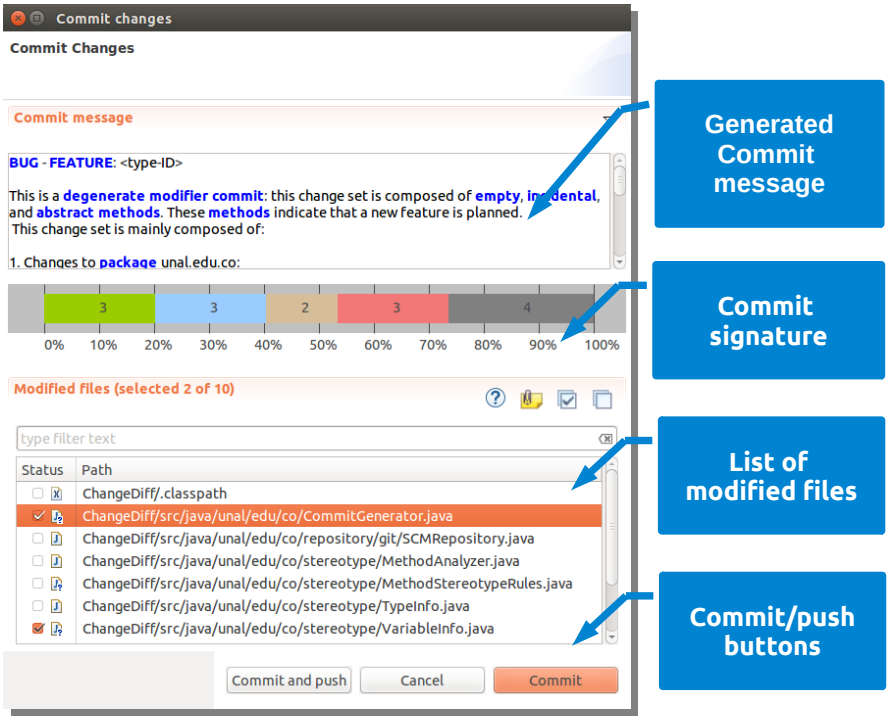
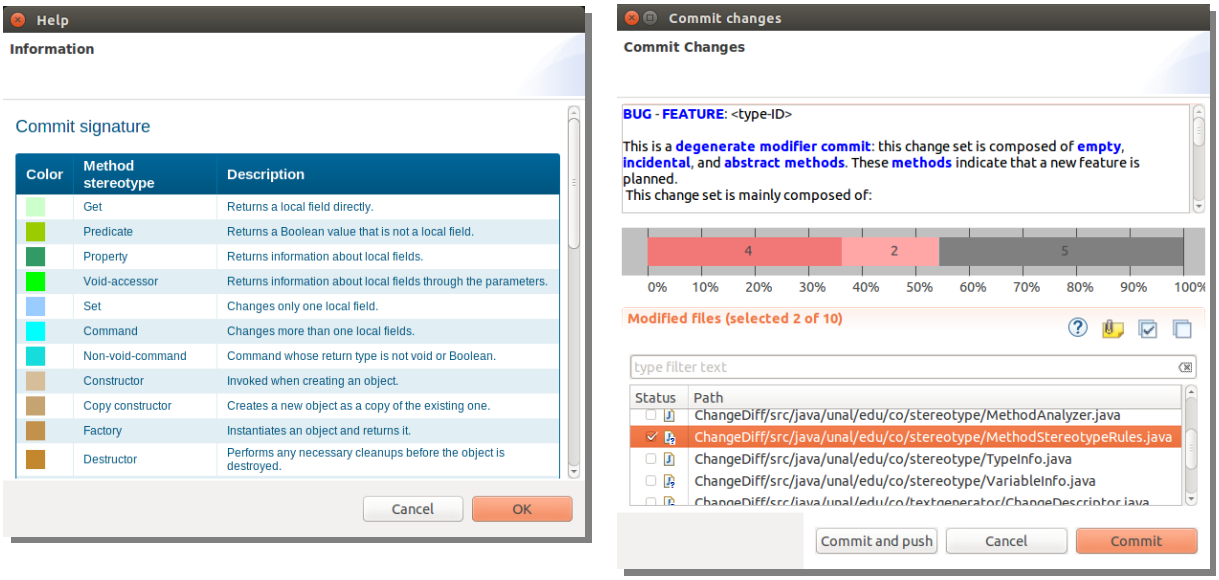


Figure 3-3.: Visualization of *ChangeScribe* plug-in

files selected in the list of modified files. Also, to configure *ChangeScribe* plug-in, such as define the committer/author user name, activate/deactivate the impact set-based filtering and to define the impact threshold. This window is depicted in the Figure **3-4**-b.



a. Help window

b. ChangeScribe preferences window

Figure 3-4.: The two windows in the *ChangeScribe* Plugin: help window, a) lists stereotypes (method and commit) and their descriptions; Preferences window, b) allows developer to set variables such as the impact threshold, and the author name.

4. Evaluative survey: Evaluating generated commit messages

We conducted a evaluative survey to assess our approach. The *goal* of this survey is to assess the quality of commit messages generated by *ChangeScribe*. The *context* is 50 commits from 6 open source projects (i.e., Elastic Search, Spring Social, JFreeChart, Apache Solr, Apache Felix, and Retrofit) hosted at GitHub (see Table **4-1**), and written in Java. The commit messages were selected randomly while manually looking for a diverse set of messages, including those representing initial commits, refactoring, large commits, short commits, and commits with pseudo-messages. In terms of size-categories defined by Hattori and Lanza [32], 4 commits are tiny, 10 are small, 17 are medium, and 19 are large. Also, our decision to use Java projects in the survey 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.

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 [22] [16]. Therefore, the *quality focus* of the survey 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.

4.1. Research Questions

In the context of our survey, 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?*

Table 4-1.: Java Projects hosted at GitHub and used in the survey. 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

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 [16]. 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 **4-2** 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 4-2.: 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. The message 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 [36] with $\alpha = 0.05$. We also computed the Cliff's delta d effect size [37] to measure the magnitude of the difference. We followed the guidelines in [37] 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, our decision in terms of statistical test was to use non-parametric methods (Mann-Whitney test, and Cliff's delta).

4.2. 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[1]. 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 **4-1** 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 **4-2**. 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 **4-1**);

- 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 **4-2**, 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 **4-2**) 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 **4-2**, we included questions suggested by Feigenspan *et al.* [38] to measure programming experience of the participants.

[1]Website of Qualtrics `http://qualtrics.com`

4.3. Replication Package

All the experimental materials used in our survey and *ChangeScribe* (Eclipse plugin) are publicly available at: `http://www.cs.wm.edu/semeru/data/ICSME14-ChangeScribe`. In particular we provide: (i) the links to the commits used in the survey, (ii) real and *ChangeScribe* commit messages, and (iii) anonymized survey's results.

Figure 4-1.: Example of code changes describing a commit. The changes are presented using a diff-based style similarly to GitHub

4.4. 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 survey involved 23 developers who evaluated 107 instances of the commits. Thus, the importance 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 survey 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

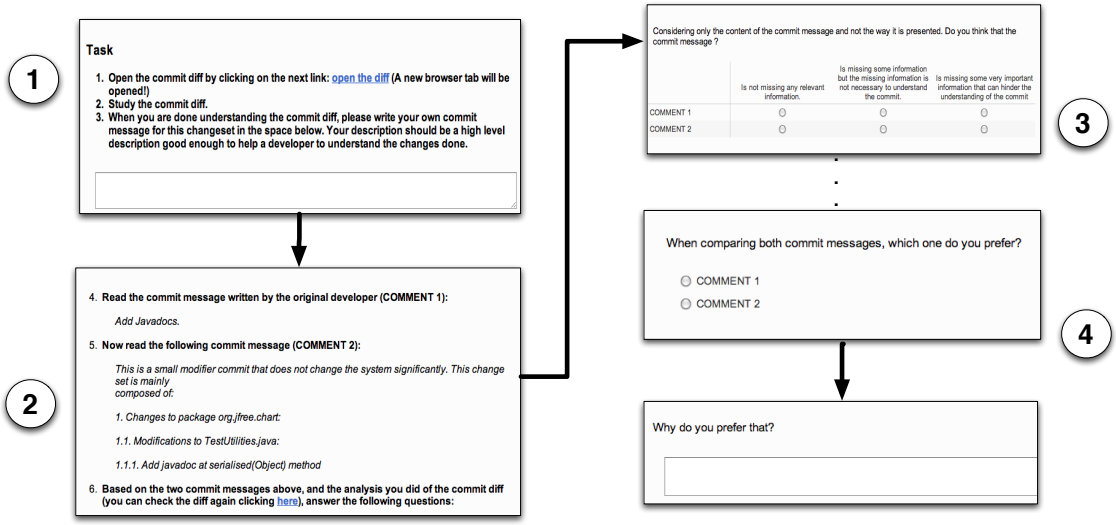


Figure 4-2.: Example of a set of questions for a particular commit

made sure that our participants had significant experience in software development and had minimal or no experience with the systems from the survey. 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.

4.5. Results

23 participants completed the evaluative survey in which they analyzed 50 commits and provided 119 evaluations. In each evaluation the participant analyzed the changes in the source code; wrote their own commit message; evaluated both the commit message written by original open source developer and the automatic commit message generated by our approach; and finally, the 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. Worth noting that the evaluator was quite familiar with each change set included in the survey, 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 that the message is presented facilitates understanding of the changes done.

4.5.1. 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 **4-3**). In general, this result 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 ($d = -0.9386784$).

Table 4-3.: 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 essential information	40	16

4.5.2. 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 **4-4**). Only in 25% of the cases the generated commit message does 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 survey 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.

The evaluated commit messages were classified by commit size using the taxonomy proposed by Hattori and Lanza [32], 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 favoured 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"*.

Table 4-4.: 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 some unnecessary information	5	27

4.5.3. RQ_3: Expressiveness

In our interpretation, this property was positively evaluated by the participants although the original commit messages got better scores (Table **4-5**). 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.

Table 4-5.: 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

4.5.4. 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 analysing the reasons why respondents preferred the original message, we found that most of the times they argue that the original message 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 automatic commit message.

On the other hand, they preferred the automatic commit message mainly because generated message is more explanatory and covers more extensively 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.

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. 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. 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"*.

5. Architecture of ChangeScribe

We integrated *ChangeScribe* into a well known IDE used in open source and commercial environments, namely Eclipse IDE[1]. We exploit the advantages of Java Development Tools (JDT)[2] as Abstract Syntax Tree features, search references, workspace features, among others. *ChangeScribe* built for Java applications only hosted on Git (see the screen-shot in Figure **5-1**).

ChangeScribe provides a dialog for the commit operation; The *ChangeScribe* menu option is available only on the Eclipse Package View for Java projects. The dialog is build using Eclipse UI API's (SWT[3] and JFace[4]). Also, *ChangeScribe* provides a dialog integrated with the Eclipse preferences to configure the plugin options such as the impact threshold, the author and commiter user names, and so on.

The *ChangeScribe* plug-in is composed of 7 components: (i) The user interface responsible for displaying the modified files, the commit stereotype, and the generated commit message, (ii) The Git component to interact with the source repository where the Java project is hosted, (iii) The source code differencing component to analyse the source code and extract the differences, (iv) The stereotype identification component to compute the method and commit stereotypes to describe the commit intention, (v) The summary generator component responsible for generating the commit message; this component is based on the following one, (vi) The phrases/sentences generator component to generate natural language sentences using the approach of [21], (vii) The impact analysis component to search for dependences of each modified Java type to compute a threshold that reflects the importance of the class in the change-set; this impact value is used to filter the content of the commit message. Figure **5-2** depicts the components of *ChangeScribe*. In this section, we describe the *ChangeScribe* architectural elements.

[1]Website of Eclipse IDE `http://www.eclipse.org`
[2]Website of Eclipse JDT `http://www.eclipse.org/jdt/`
[3]Website of SWT project `http://www.eclipse.org/swt/`
[4]Website of JFace project `http://wiki.eclipse.org/JFace`

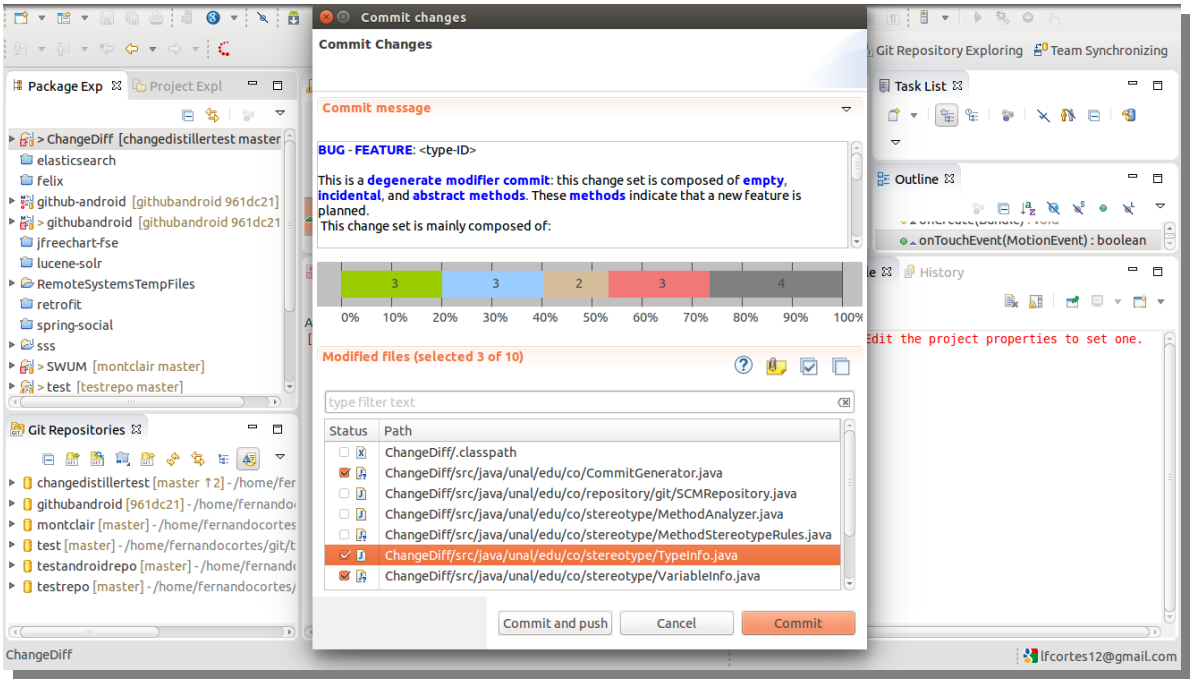


Figure 5-1.: ChangeScribe in action

5.1. Git as data source

For this thesis we used information obtained from Git repositories. Git is an open source version control system that allows to have multiple local branches and tags, also Git is very fast and optimized to take up little space on your hard disk. Git is a distributed system, which means that each developer obtains an entire copy of the repository with the *clone* command, and also this allows developers to work with any work-flows such as *Subversion style*, *integration manager*, among others. For instance, in the *integration manager* work-flow a single person, the *integrator role* is responsible to do commit to the *blessed repository*, and then, software developers clone that repository and they do push their own independent repositories, and ask to the integrator to pull in their changes.

Git stores detailed information for each commit in a repository. This information includes: date of the commit, list of modified files, author of the changes, commiter, and a message that can be used to describe the modifications applied to the source code. We extract

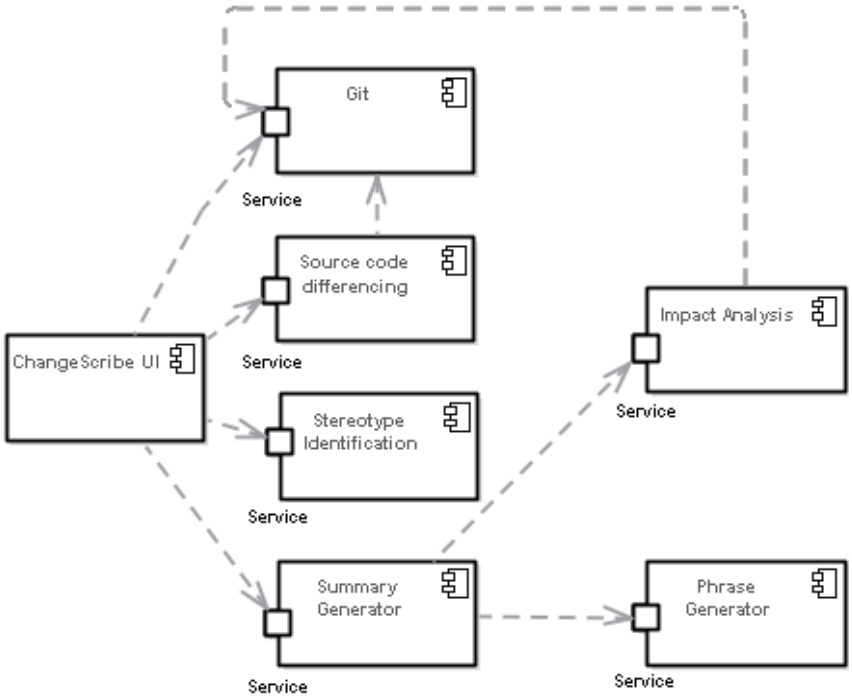


Figure 5-2.: Component diagram of ChangeScribe

the source code changes using a Java Git implementation called JGit[5]. This Java library allows to implement repository access routines (i.e. commit, push, pull, merge, rebase, and so on), several network protocols (i.e. HTTP, HTTPS, SSH), rename identification algorithm, and core version control algorithms. This main class to perform this process is *SCMRepository* because this allows extract a list of modified classes and also get the Git repository status. Each modified file is modelled by *ChangedFile* class. This class contains the main information of a modified file such as the file name, file path, a list of modified methods, the type of change (add, delete, update). The *TypeChange* enumeration lists the types of possible changes. Figure **5-3** depicts the classes used to interact with the source code repository.

[5]Website of JGit project `http://www.eclipse.org/jgit/`

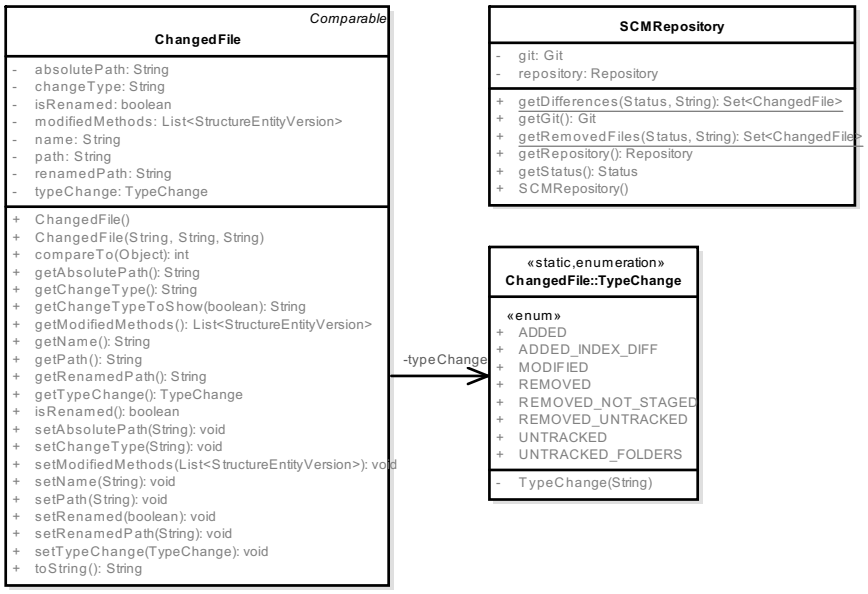


Figure 5-3.: Class diagram of Git component

5.2. Source code differencing

Once the list of modified files is identified *ChangeScribe* computes the source code differences for each file. This process is implemented using *Change Distiller*, a plug-in proposed by Fluri *et al.* [33]. *Change Distiller* like *ChangeScribe* uses Java Development Tool (JDT). *Change Distiller* receives as input two versions of a Java type (Class or interface) and for each version this builds an intermediate abstract syntax tree using the AST Visitor from JDT Eclipse API. Finally, the differencing algorithm is applied to the proposed AST generating a classified list (based on a set of change types) of basic edit tree operations.

5.3. Commit stereotype identification

This architectural element was built to extract the commit intention. Then, In this step is necessary to compute the method stereotypes. We use *JStereCode* Tool implemented by Moreno *et al.* [29] to identify method stereotypes by analysing abstract syntax trees and

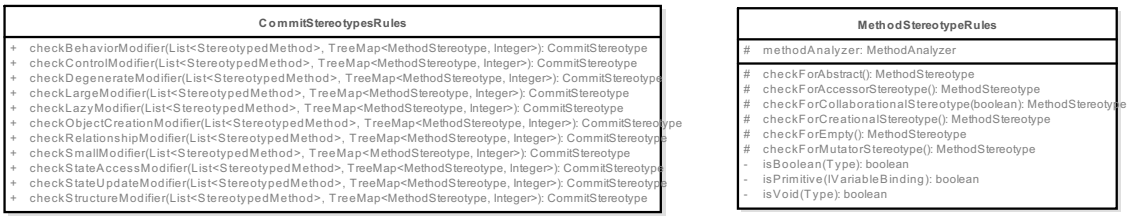


Figure 5-4.: Class diagram of commit stereotype component

using the rules proposed by Dragan *et al.* [34]. This tool uses AST Visitor from JDT Eclipse API similar to Change Distiller and performs an AST walker analysis. Once the method stereotypes are computed, we implement the rules proposed by Dragan *et al.* [34] in order to compute the commit stereotype. *MethodStereotypeRules* class is responsible for compute the method stereotype for all methods of a class. The *CommitStereotypeRules* class is responsible for check the commit stereotype, for instance, if the commit is behaviour modifier, control modifier, among others. Figure **5-4** shows the classes responsible for computing the commit stereotype.

5.4. Text generation

Regarding the text generation, we implement the algorithm proposed by Hill *et al.* [21]. In this process is necessary to split the source code identifiers to obtain a list of words. Then for each word, we assign parts of speech for each one of them using the approach proposed by Toutanova *et al.* [39]. For the content filter, we implement the impact algorithm using the search engine of JDT API. This API allows to search for references and declarations of any class, interface, field, method, and so on. The *Phrase* interface models a sentence of natural language and this interface is implemented by three classes *VerbPhrase* class, *NounPhrase* class, *ParameterPhrase* class. These classes implement the rules proposed by Dragan *et al.* [17]. The Figure **5-5** shows the class diagram of the phrases generation component.

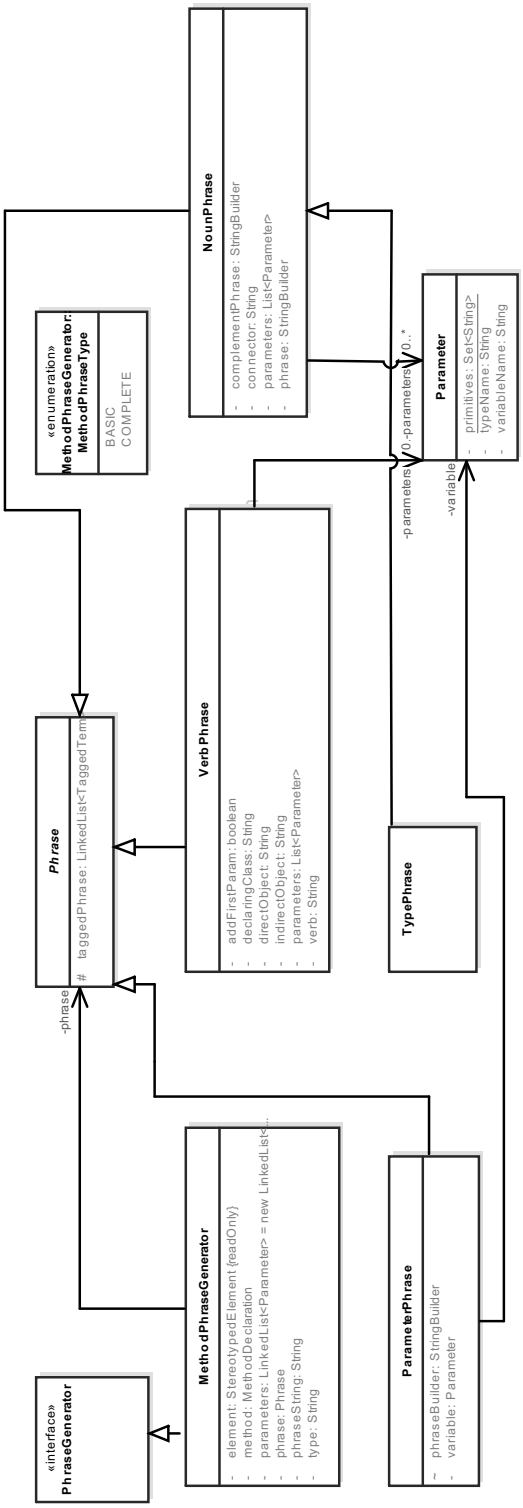


Figure 5-5.: Class diagram of phrase generator component

6. Conclusions

This thesis presents an approach for generating automatic commit messages based on the code changes included in a change set. *ChangeScribe* extracts and analyses 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 an evaluative 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 survey 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 survey participants preferred *ChangeScribe*'s commit messages to the ones written by the original developers. 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.

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. 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.

Currently, *ChangeScribe* is working as an Eclipse plug-in allowing developers to generate the commit messages. But, when is necessary generate the commit message of multiple change-set, for instance in Mining Software Repositories, *ChangeScribe* not allow this feature

because this run within Eclipse IDE. In this context, is necessary implement a command line version and an Application Programming Interface (API) to enable *ChangeScribe* for large scale studies of Mining Software Repositories, program comprehension and software evolution and maintenance. Also, another feature that can be implemented is the SVN or Mercurial support. This feature will allow greater and easier *ChangeScribe* acceptance by developers.

A. Examples of commit messages generated with ChangeScribe

This appendix shows examples of commit messages generated with *ChangeScribe* plug-in of several open source projects.

A.1. Generated commit message of Elastic Search project

This change set is available at: `http://goo.gl/1c0s1l`.

> BUG - FEATURE: <type-ID>
> This commit renames some files. This change set is mainly composed of:
>
> 1. Changes to package org.elasticsearch.node.internal:
>
> 1.1. Rename type InternalSettingsPerparerTests with InternalSettingsPreparerTests

A.2. Generated commit message of Spring Social project

This change set is available at: `http://goo.gl/5Igx1s`.

> Initial commit. 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 commit includes changes to internationalization, properties or configuration files (.classpath, .gitignore, .project, ...). The commit includes these new modules:
>
> - facebook

- twitter

This change set is mainly composed of:

1. Changes to package org.springframework.social.twitter:

1.1. Add a data class for tweet. It allows to:

Get text;
Set text;
Get created at;
Set created at created at date;
Get tweet from user;
Get set from user;
Get id;
Set id;
Get profile image url;
Set profile image url;
Get tweet to user id;
Set tweet to user id;
Get tweet from user id;
Get set from user id;
Get language code;
Set language code;
Get source;
Set source

Referenced by:
SearchResults class
TwitterService class

A.3. Generated commit message of JFreeChart project

This change set is available at: `http://goo.gl/M1ILNF`.

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.jfree.chart:

1.1. Modifications to TestUtilities.java:

1.1.1. Add javadoc at serialised(Object) method

A.4. Generated commit message of JFreeChart project

This change set is available at: `http://goo.gl/StXeJS`.

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.jfree.chart.plot:

1.1. Modifications to RingPlot.java:

1.1.1. Modify variable declaration extendedSeparator

2. Changes to package org.jfree.chart.util:

2.1. Modifications to LineUtilities.java:

2.1.1. Add a functionality to extend line

The added/removed methods triggered changes to RingPlot class

A.5. Generated commit message of Apache Lucene/Solr project

This change set is available at: `http://goo.gl/LOcTWh`.

BUG - FEATURE: <type-ID>
This is a small modifier commit that does not change the system significantly.This com-

mit includes changes to internationalization, properties or configuration files (CHANGES.txt). This change set is mainly composed of:

1. Changes to package org.apache.lucene.facet:

1.1. Modifications to MultiFacets.java:

1.1.1. Add variable declaration statement at getAllDims(int) method
1.1.2. Add line comment at getAllDims(int) method
1.1.3. Add foreach statement at getAllDims(int) method
1.1.4. Add if statement at getAllDims(int) method
1.1.5. Add return statement at getAllDims(int) method
1.1.6. Add method invocation to method add of results object at getAllDims(int) method
1.1.7. Remove line comment at getAllDims(int) method
1.1.8. Remove throw statement of UnsupportedOperationException exception

A.6. Generated commit message of Apache Lucene/Solr project

This change set is available at: `http://goo.gl/IV6aWm`.

BUG - FEATURE: <type-ID>
This is a state update modifier commit: this change set is composed only of mutator methods, and these methods provide changes related to updates of an object's state. This change set is mainly composed of:

1. Changes to package org.apache.solr.common.cloud:

1.1. Modifications to ClusterState.java:

1.1.1. Remove an unused functionality to get shard

A.7. Generated commit message of Apache Felix project

This change set is available at: `http://goo.gl/6NfXeg`.

BUG - FEATURE: <type-ID>
This is a behaviour modifier commit: this change set is composed of command and non-void-command methods, and these methods execute complex internal behavioural changes within an object. This commit includes changes to internationalization, properties or configuration files (pom.xml, metadata.xml). This change set is mainly composed of:

1. Changes to package org.apache.felix.ipojo.test.scenarios.component:

1.1. Modifications to ReconfigurableSimpleType.java:

1.1.1. Add a functionality to prop reconfigurable simple type
1.1.2. Add (Object state) controller attribute

2. Changes to package org.apache.felix.ipojo.test.scenarios.factories:

2.1. Modifications to ReconfigurationTest.java:

2.1.1. Rename testRevalidationOnREconfiguration method with testRevalidationOnReconfiguration
2.1.2. Add a functionality to set reconfiguration test
2.1.3. Add a functionality to test revalidation on reconfiguration using config admin and controller
2.1.4. Add a functionality to tear reconfiguration test
2.1.5. Add a functionality to test revalidation on reconfiguration with controller
2.1.6. Add (Object state) admin attribute

3. Changes to package org.apache.felix.ipojo.test.scenarios.util:

3.1. Modifications to Utils.java:

3.1.1. Add a functionality to wait utils for service

A.8. Generated commit message of Retrofit project

This change set is available at: `http://goo.gl/U0TaQt`.

BUG - FEATURE: <type-ID>
This is a large modifier commit: this is a commit with many methods and combines multiple roles. This change set is mainly composed of:

1. Changes to package retrofit.core:

1.1. Modifications to Callback.java:

1.1.1. Add a functionality to client error
1.1.2. Remove an unused functionality to client error

1.2. Add a data class for client message. It allows to:

Instantiate client message;
Get title;
Get message;
Get button label

2. Changes to package retrofit.http:

2.1. Modifications to CallbackResponseHandler.java:

2.1.1. Modify arguments list when calling clientError method at handleResponse(HttpResponse) method
2.1.2. Add a private functionality to parse client message

2.2. Modifications to UiCallback.java:

2.2.1. Add a functionality to client error
2.2.2. Remove an unused functionality to client error

Bibliography

[1] 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.

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

[3] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *ICPC'13*, pages 23–32, 2013.

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

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

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

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

[8] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE'13*, pages 422–431, 2013.

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

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

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

[12] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122, 2005.

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

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

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

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

[17] Sunghun Kim, E. J. Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? 34(2):181–196, 2008.

[18] Adrian Bachmann and Abraham Bernstein. Data retrieval, processing and linking for software process data analysis. Technical report, University of Zurich, Department of Informatics, 2009.

[19] Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from cvs. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 125–128. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-024-1.

[20] Ieee standard for information technology- portable operating system interface (posix) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1 –3826, dec. 2008.

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

[22] Hoan A. Nguyen, Tung T. Nguyen, Hung V. Nguyen, and Tien N. Nguyen. idiff: Interaction-based program differencing tool. In *ASE'11*, pages 575–575, 2011.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] Sarah Rastkar, Gail C. Murphy, and Gabriel Murray. Automatic summarization of bug reports. *IEEE TSE*, to appear, 2013.

[29] Annie T.T. Ying and Martin P. Robillard. Code fragment summarization. In *ESEC/FSE'13*, 2013.

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

[31] 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.

[32] 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.

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

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

[35] 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.

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

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

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

[39] Kristina Toutanova and Christopher D. Manning. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *Proceedings of the 2000 Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora: Held in Conjunction with the 38th Annual Meeting of the Association for Computational Linguistics - Volume 13*, EMNLP '00, pages 63–70. Association for Computational Linguistics, Stroudsburg, PA, USA, 2000.