Using Stereotypes to Help Characterize Commits

Natalia Dragan
Computer Science
Kent State University
Kent, Ohio, USA
ndragan@cs.kent.edu

Michael L. Collard
Computer Science
The University of Akron
Akron, Ohio, USA
collard@uakron.edu

Maen Hammad
Software Engineering
Hashemite University
Zarqa, Jordan
mhammad@hu.edu.jo

Jonathan I. Maletic
Computer Science
Kent State University
Kent, Ohio, USA
jmaletic@kent.edu

***Abstract*—Individual commits to a version control system are automatically characterized based on the stereotypes of added and deleted methods. The stereotype of each method is automatically reverse engineered using a previously defined taxonomy. Method stereotypes reflect intrinsic atomic behavior of a method and its role in the class. The stereotypes of the added and deleted methods form a descriptor of the change embodied by a given commit. These descriptors are then used to categorize commits, into types, based on the impact of the changes to a class (or classes). The goal is to gain a better understanding of the design changes to a system over its history and provide a means for documenting the commit.**

Keywords-method stereotypes, commit types, reverse engineering, redocumenation

I. INTRODUCTION

Version control systems, such as *Subversion*, *CVS*, *Git*, *MS Visual SourceSafe*, and *Mercurial*, are standard tools to help manage changes to artifacts during the development and maintenance of software systems. As changes to the system are made, a new version is saved as a commit and stored by the version control system. This new version can be compared to previous versions (using tools such as *diff*) to determine what changed. These changes may be quite simple, such as fixing a spelling error in a comment, or quite complex, such as adding a new feature to the system.

Error correction (i.e., bug fixing) normally involves only small changes, whereas adding new features or altering the design of a system typically requires the addition and/or removal of classes or methods. This latter class of changes often has broader implications to developers, testing plans, and project management. Here we focus on changes (more specifically commits) that alter the design of a system. Furthermore, we want to better understand the types of different design changes taking place in a given commit and across the evolution of a system. The work presented here proposes a means to categorize commits that impact the design of a software system.

Knowing and understanding what types of changes are occurring in a given commit is very valuable to developers, testers, and managers. For example, if we know a commit changes the behavior of a given class, then that class would need to be re-tested and additional test cases may need to be developed or integrated into the test suite. This would also give some notification to a developer that code using this class may be impacted. A manager could use such information to assess the cost of a given change and assess the risks of different deployment options. That is, if a particular change impacts a module or class that has historically been error prone, the risk assessment may be too great to deploy that change.

In an ideal environment, good development practice would annotate a commit with an accurate description about what is being changed. However, in reality this is rarely done or is inaccurate and/or incomplete. Therefore, we feel that an automated approach to augment commit messages would be valuable. Additional knowledge and understanding can be derived from the source code and commit, and explicitly documented to help address this problem. To accomplish this, we must first develop a set of commit-categories (or types) that are meaningful to developers and assist them in understanding what maintenance activities are taking place in a commit. Commits can be categorized with data present in the version control system or directly measured from the commit, e.g., LOC, author, etc., and can also be categorized based on analysis of commit messages via Natural Language Processing [1], [2], or information retrieval techniques [3].

Our goal is to develop an efficient approach that provides simple, yet fairly accurate, heuristics to the developers as to the overall characteristics of a given commit in the context of how it impacts the behavior or structure of classes. To accomplish this, we build on our previous work, which reverse-engineers method stereotypes from source code. The stereotype information of methods added or deleted in a commit is used to construct a categorization of commit types. Then, we define an automated approach to derive the commit type and label commits with this meta-data in a pilot study.

The next section (II) contains an overview of previous work [4, 5] on method stereotypes. We also define the idea of a commit signature that forms the input for automatic identification of commit types. In section III we present a categorization of commits. Section IV describes our approach to reverse engineering commit types from existing C++ code. Section V describes a pilot study using the approach. This is followed by related work and conclusions.

II. DEFINING COMMIT SIGNATURES

A commit details the changes to a software system and may represent major design changes or may just be minor edits or comment improvements. Here we provide a mechanism to automatically identify the different types of commits that impact the design of a system. Our approach of defining commit types is based on method stereotypes [4] and how the changes impact different types of methods. Method stereotypes are generalizations that reflect some

intrinsic or atomic behavior of a method and indicate the method's role and responsibilities within a class. With stereotype information of the methods in a commit, we can enrich the context of existing versioning systems with additional semantics of method and class level changes.

We start by defining terms that are used in this work. A *method is in a commit* if the method is added or deleted as part of the commit. A *commit signature* is the frequency distribution of stereotypes of added/deleted methods occurring in a commit and is used to identify a commit type. The commit signature provides information about what types of design changes are actually occurring in a commit. A *design change* is defined as the addition or deletion of a class, a method, or a relationship in the corresponding UML class diagram [6].

The aggregates for commit signature identification, method stereotypes, were first introduced in [4] and we refer the reader there for complete details and examples. The taxonomy of method stereotypes (Table 1) is organized by the main role of a method, while simultaneously emphasizing its creational, structural, behavioral, and collaborational aspects with respect to a class's design. We now describe how the method stereotypes are used for defining the commit signatures.

TABLE I. TAXONOMY OF METHOD STEREOTYPES.

Stereotype Category		Stereotype	Description
Structural *Accessor*		get	Returns a data member.
	Behavioral	predicate	Returns Boolean value.
		property	Returns information about data members.
		void-accessor	Returns information through a parameter.
Structural *Mutator*		set	Sets a data member.
	Behavioral	command	Performs a complex change to the object's state.
		non-void-command	
Creational		factory	Creates and/or destroys objects.
Collaborational		collaborator	Works with objects.
		controller	Changes an external object's state (not *this*).
Degenerate		incidental	Does not read/change an object's state.
		empty	Has no statements.

A. Commit Signature

The idea of a *system signature* was introduced in [5] where the distribution of stereotypes for one open source system was examined. That study demonstrated that distributions of method stereotypes are potentially good indicators of system design. Here we apply a similar concept to commits to better understand design changes.

As a distribution of method stereotypes for the methods that are added/deleted in a commit, a *commit signature* provides us with a heuristic of the structural complexity of the changes occurring in a commit. From the commit signature we can infer information of system changes and specifically whether the system gains more structural, behavioral, collaborational, or control features.

A signature is formed by first determining which methods are in each commit (i.e., those methods added or deleted) and then automatically reverse engineering the stereotype for each of these methods. The sum of the stereotypes in the commit is calculated. The method stereotype counts are shown as a bar chart (see examples in Table II, numbers in the rectangles) ordered by method stereotype categories: accessors, mutators, creational, and collaborational. The color scheme is the following: accessors and mutators are shown in different nuances of green and blue respectively, factory – in tan, collaborational – in rose and turquoise, degenerate – in grey. In the bar chart the method stereotype is given a grey shadow effect if the method is also a collaborator (e.g., get collaborator).

Note, we use a lightweight approach and ignore changes to existing methods. We feel little additional information will be added by its inclusion. Our main argument for this is because we are particularly interested in changes that impact the system's design. Small changes to the body of existing methods often reflect error corrections (bug fixes) and are less likely to impact the design ([7], [6]). Clearly, additional investigation is necessary to fully understand the impact of such changes and to completely support our argument.

III. COMMIT CATEGORIZATION

The categorization of commits based on our empirical examination of the evolution history of open source systems is presented in this section. The process of commit categorization is influenced by our previous work on uncovering patterns of design from a single-version system at the different levels of abstraction: method [4], class [5], and system [8]. This foundation of identifying stereotypes at the method, class, and system level allowed us to hypothesize that those patterns of design, in the form of method stereotype distributions for a single-version system, also exist in multiple-version systems and could characterize design changes over the evolution history. A software system evolves through the changes in structural, behavioral, creational, and collaborational characteristics that are implemented in methods. Each method in a commit has specific responsibilities within the class and we characterize a commit by aggregating the responsibilities of the methods added/deleted in the change. The commit types are defined from the distributions of method stereotype. A given commit may take on more than one of these types. The list of commit types is shown visually in Table II with specific examples from the open-source systems Kate, KSpread, and QuantLib. We now individually explain each commit type.

A *Structure Modifier* commit is responsible for changes related to data storage and only contains methods that perform simple access and modification to the data. It consists only of get and set methods.

A *State Access Modifier* commit consists of methods that provide a client with information and does not change any data members. It consists almost entirely of accessor methods. For example, commit #582964 of Kate has 8

accessor methods out of the 9 methods changed in the commit.

A *State Update Modifier* commit provides changes related to updates of an object's state and consists mainly of mutator methods. These methods often implement complex behavior and may involve objects of different classes.

A *Behavior Modifier* commit is a special case of the State Update Modifier where the main characteristic is to execute complex *internal* behavioral changes within an object. It mainly consists of command and non-void-command methods. The largest part of the logic for the class's behavior is implemented in these methods.

An *Object Creation Modifier* commit is responsible for changes related to creation of objects and has mostly factory methods. Commit #496123 of Kate is an example where 75% of the added/deleted methods are factory methods.

A *Relationships Modifier* commit adds or deletes methods that implement generalization, dependency and association relationships by performing calls on parameter or local variable objects. These changes, performed by a commit consisting of many collaborational methods, represent modifications of relationships between classes. Alternatively, this type of commit could be a State Access Modifier when the main purpose of its methods is to get data from a model (when it mainly consists of accessors) or Behavior Modifier when the main purpose of its methods is to update data (when it mainly consists of mutators). Commit #496124 of Kate is an example of a Relationships Modifier commit where more than 75% of added/deleted methods have a stereotype collaborator.

A *Control Modifier* commit provides changes in the *external* behavior of the participating class, i.e., processes data of the class's external objects. It consists mostly of controller methods that implement external class's behavior.

A *Large Modifier* commit contains a large number of responsibilities. This is a commit with a high impact on design. "Large number" can be characterized using metrics such as number of methods, number of classes, LOC, etc. However, those types of metrics do not directly reflect the different semantics of changes. We consider a commit a Large Modifier commit if it has both many methods and combines multiple roles, such as State Access Modifier, Behavior Modifier, Relationships Modifier, and Control Modifier.

A *Lazy Modifier* commit is a very trivial commit that does "too little". The Lazy Modifier commit might occur in the context of a new or planned feature that is not yet completed. This is a commit with a minimal impact on design. Similarly, "too little" can be interpreted using different metrics. But we consider a commit as Lazy Modifier if it has get/set methods and a low percentage of other methods. The commit is also considered Lazy Modifier if it has a large number of degenerate methods.

A *Degenerate Modifier* commit includes a degenerate, incidental, or empty method. If a commit contains even one degenerate method it means that adding a new feature is planned. As a maintainer we would like to know when exactly in the evolution history this will occur and how this method is changed (if at all).

A *Small Modifier* commit has only one or two methods and does not change the system significantly.

With a definition of commit types based on the commit signature, we can automatically reverse engineer the commit type. To do so we perform the following steps:

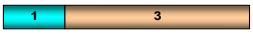
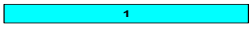
Recover design changes from the code changes of commit by the *srcTracer* tool [6].

Extract added/deleted methods per commit from the design changes.

Identify commit signature for the extracted methods with the *StereoCode* tool [4].

Identify a commit type by applying rules on the commit signature.

TABLE II. EXAMPLES OF COMMIT TYPES WITH THEIR ASSOCIATED SIGNATURES IN SYSTEMS KATE, KSPREAD AND QUANTLIB.

Commit Type	Signature
Structure Modifier (#502478 Kate)	
State Access Modifier (#582964 Kate)	
State Update Modifier (#593810 Kate)	
Behavior Modifier (#493147 Kate)	
Object Creation Modifier (#496123 Kate)	
Relationships Modifier (#496124 Kate)	
Control Modifier (#6375 QuantLib)	
Large Modifier (#605471 KSpread)	
Lazy Modifier (#859282 Kate)	
Degenerate Modifier (#715531 Kate)	
Small Modifier (#525142 Kate)	

A tool *StereoCommit* was developed to automatically identify the commit types. The commit signatures are fed into StereoCommit to assign types to a commit. The rules for identification of commit types are influenced by the rules on automatic identification of patterns of design at the class level for a single-version system [8].

IV. PILOT STUDY

We now apply our approach of commit categorization by introducing a *commit label*. Previously, a commit labeling concept was described in [9], however it is limited to listing the exact design changes of commits (i.e., names of methods and classes added/deleted, and type of relationships changed). Here, we label each commit with the commit types and its signature (the method stereotypes distribution).

Three years of the evolutionary histories of four C++ open source projects (the editor *Kate*, the spreadsheet

KSpread, the finance library *QuantLib*, and the GUI library *wxWidgets*) with 18120 commits were analyzed. We examined the following questions. Do the commit types we defined exist in the evolution histories of real systems? How well do the commit categories cover actual commits? What is the distribution of commit types?

The data showed that the majority of the commits (96.5% to 99.5%) fit into at least one of the commit types and all commit types occur in all of these systems. Based on the distribution of the commit types we observed some similarities and differences between the systems. The data showed that the frequency and distribution of commit types across a system reflected an implementation of particular design decisions, underlying architecture, and good/bad design changes. We also obtained an initial result concerning the correlation between the commit type and maintenance type (e.g., bug fix, feature addition, refactoring). However, further investigation is required to formulate any type of a broader conclusion. Demonstrating that a mapping exists between a given commit type(s) and a maintenance type(s) would be concrete evidence for the usefulness of the approach.

V. Related work

Automatic classification of large changes in software systems into various categories of maintenance tasks using machine learning techniques is given in [2]. Hattori and Lanza [10] propose commit classification with respect to the size that is based on the number of files. Additionally, they classify commits by the types of development and maintenance activities based on the content of the comments. D'Ambros et al. [11] present an approach to visualize changes at different levels and allow a user to comment the commit. Evolution of the object-oriented software system at a coarse-grained level is analyzed in [12]. Design patterns at the class-level are investigated in [13] to find common patterns across projects or releases. Analysis of changes at the method-level is performed in [14].

Our work is distinguished by identifying key characteristics of commits such as changes to the class structure, class behavior, changes related to the communication, creation and control of other objects, and type of access to class's data members. We do not study the internal evolutionary patterns of methods or classes; instead we match the semantic information about a group of added/deleted methods to a set of classes.

VI. Conclusions and Future Work

A means to categorize the changes occurring in a commit is proposed. The categorization is based on the stereotype of the methods (in C++) that are added or deleted in the commit. The category names help to characterize the type of a change occurring in a given commit. The intent is to assist the software developer in understanding the extent and impact of the change on a system and to ease the communication between developers.

The commit categories are derived from empirically examining the distributions of method stereotype changes in open source systems. That is, the categories were an emergent artifact from the data of the systems examined and reflect the variety of changes we observed in the context of method stereotypes. In support of this particular commit categorization, we demonstrated that it was complete enough to label the majority of the commits in four systems (over three years of history for each). While we do not claim this validates the correctness or completeness of our commit categorization scheme, it does give us a high level of confidence that it would be useful to assist in better understanding the types of changes occurring. We have yet to demonstrate that labeling commits with this categorization actually improves the understanding of the changes occurring, and we are currently designing an experimental study to test the hypothesis and usefulness of the approach.

References

[1] L. P. Hattori and M. Lanza, "On the Nature of Commits," in 4th International ERCIM Workshop on Software Evolution and Evolvabillity (EVOL'08), 2008, pp. 63 - 71.

[2] A. J. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic Classification of Large Changes into Maintenance Categories," in IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada 2009.

[3] H. Kagdi and D. Poshyvanyk, "Who Can Help Me with this Change Request?," in IEEE 17th International Conference on Program Comprehension (ICPC '09) Vancouver, BC 2009, pp. 273-277.

[4] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," in 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, 2006, pp. 24-34.

[5] N. Dragan, M. L. Collard, and J. I. Maletic, "Using Method Stereotype Distribution as a Signature Descriptor for Software Systems," in IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada 2009, pp. 567-570.

[6] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically Identifying Changes that Impact Code-to-Design Traceability," in 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada, 2009, pp. 20-29.

[7] S. Raghavan, R. Rohana, A. Podgurski, and V. Augustine, "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases," in 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, 2004, pp. 188-197.

[8] N. Dragan, M. L. Collard, and J. I. Maletic, "Atomatic Identification of Class Stereotypes," in 26th IEEE International Conference on Software Maintenance (ICSM'10), Timisoara, Romania 2010, p. to appear.

[9] M. Hammad, M. L. Collard, and J. I. Maletic, "Automatically Identifying Changes that Impact Code-to-Design Traceability During Evolution " Journal of Software Quality vol. 18, to appear, accepted for publication April, 2010 2010.

[10] M. Lanza and L. Hattori, "On the Nature of Commits," in 4th International ERCIM Workshop on Software Evolution and Evolvabillity, 2008, pp. 63 - 71.

[11] M. D'Ambros, M. L. Lanza, and R. Robbes, "Commit 2.0," in 1-st International Workshop on Web 2.0 for Software Engineering (Web2SE' 2010), 2010, pp. 14-19.

[12] X. Dong and M. W. Godfrey, "Identifying Architectural Change Patterns in Object-Oriented Systems," in IEEE International Conference on Program Comprehension Amsterdam, The Netherlands, 2008, pp. 33-42.

[13] S. Kim, K. Pan, and E. J. J. Whitehead, "Micro Pattern Evolution," in International Workshop on Mining Software Repositories (MSR '06) Shanghai, China, 2006.

[14] S. Kim, E. J. J. Whitehead, and J. Bevan, "Properties of Signature Change Patterns," in 22nd IEEE International Conference on Software Maintenance (ICSM'06) Philadelphia, Pennsylvania USA, 2006, pp. 4-13.