Which Method-Stereotype Changes are Indicators of Code Smells?

Michael J. Decker
Department of Computer Science
Bowling Green State University
Bowling Green, OH, USA
mdecke@bgsu.edu

Christian D. Newman
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
cnewman@se.rit.edu

Natalia Dragan
Departmnet of Management and Information Systems
Kent State University, Kent, OH, USA
ndragan@kent.edu

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

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, OH, USA
jmaletic@kent.edu

Nicholas A. Kraft
ABB Corborate Research
Raleigh, NC, USA
nicholas.a.kraft@us.abb.com

***Abstract*— A study of how method roles evolve during the lifetime of a software system is presented. Evolution is examined by analyzing when the stereotype of a method changes. Stereotypes provide a high-level categorization of a method's behavior and role, and also provide insight into how a method interacts with its environment and carries out tasks. The study covers 50 open-source systems and 6 closed-source systems. Results show that method behavior with respect to stereotype is highly stable and constant over time. Overall, out of all the history examined, only about 10% of changes to methods result in a change in their stereotype. Examples of methods that change stereotype are further examined. A select number of these types of changes are indicators of code smells.**

Keywords—method stereotypes, software change, software evolution, code smells, empirical

I. Introduction

A typical, non-trivial, software system is in a constant state of evolution over its lifetime [1]. At points in this lifetime, its design may degrade or be broken by the evolutionary change. This eventually requires developers to spend time redesigning parts (or the entirety) of the system. Code smells [2][3] are one indicator of poor design or design degradation. Thus, there is research examining ways to automatically identify certain code smells with the goal of warning users when a system's design is potentially degrading [4][5][6].

The work presented here takes a similar approach. That is, we are trying to understand what types of changes to a system are potentially hazardous to the design. Here, we are specifically interested in how individual methods change over time. While a change to an individual method will not typically degrade a system's design, changes to large sets of methods can. Hence, we empirically investigate and propose a relatively simple abstraction and lightweight approach that indicates when a change is a potential problem.

The granularity of change we are examining is at the level of a method. The abstraction we are using is the idea of method stereotypes [7][8]. Method stereotypes represent a rich abstraction of the role and behavior of a method within its class and the system as a whole. A number of researchers have leveraged stereotypes for various applications [9][10][11][12][13][14]. Two simple examples of stereotypes are accessor (e.g. getter) and mutator (e.g. setter). Accessor methods allow for information to be read from an object while preventing modification of the object. Mutator methods allow one to modify the state of an object. In the next section, we present a complete set of stereotypes and definitions.

In the work presented here, we are interested in how the stereotype of a particular method changes during the lifetime of a system. In particular, we want to determine if a method's stereotype changes often or rarely. What types of changes in stereotype are most common? Are some types of these changes hazardous? Or innocuous? Our goal is to determine what is the norm, with regards to changes in stereotype, so we may let developers and project managers know when something is abnormal, and thus potentially a problem.

To this end, we have undertaken an empirical study of the complete version history of 50 open-source C++ systems. This provides us with almost 1.4 million changes to methods. We also separately examined 6 closed-source systems with nearly 54K changes to methods. Each of the methods that change are examined over time and what happened to their (automatically computed) stereotype is recorded. Our first goal is to examine the stability of method stereotypes during evolution. That is, do method stereotypes change as methods change?

Our second goal is to determine what types of changes from one stereotype to another stereotype are potentially hazardous. Intuitively, a method should not change drastically from its original intent. If there is a drastic change in its role and/or behavior this may be an indication of a problem. However, clearly systems do evolve and the role of methods must evolve in some manner. There may also be changes that fix poor designs and these types of changes may be reflected in the stereotype of a method. We seek to answer the following research questions:

- **RQ1:** Is a method's stereotype stable as it evolves?
- **RQ2:** What types of method-stereotype changes occur?
- **RQ3:** Which method-stereotype changes are benign or suspicious?

To answer these questions, we perform three separate types of analysis. First, we analyze stereotype changes in terms of the change in behavior they represent and the consequences they have on the source code. Then, we perform an empirical investigation on 50 open-source software systems. Finally, we finish with a manual investigation of method-stereotype changes. We also examine 6 closed-source systems to see if the results are similar to the open-source code. The contributions of this work are as follows:

- An empirical study that demonstrates that the stereotype of a method is very stable as it evolves within a system.
- A set of defined stereotype transitions and their potential impact on a system's design.
- A manual evaluation of potentially problematic stereotype transitions.

The paper is organized as follows. In Section II, we discuss method stereotypes in detail. In Section III we discuss our motivation for investigating stereotype stability. In Section IV, we discuss the stability of method stereotypes and introduce a taxonomy of method-stereotype changes. In Section V, we discuss how we collect data for an empirical and manual investigation. In Section VI, we present the results of an empirical and manual investigation of the method stereotypes and relate it to the taxonomy. Finally, in Sections VII and VIII, the threats to validity and conclusion are provided.

II. Method Stereotypes

We now provide a brief introduction to method stereotypes. The work of Dragan et al. [7] [15][16] introduces a taxonomy for method stereotypes, which we show in TABLE I. A method stereotype concisely represents behavioral aspects of a method. Method stereotypes are separated into five broad categories: *Structural Accessors* – query the state of an object on which it is called; *Structural Mutators* – modify the state of an object on which it is called; *Creational* – create/provide new objects; *Collaborational* – work on objects pertaining to classes other than itself; and *Degenerate* – no use of the object's state and often no statements.

The individual stereotypes indicate a refinement of the broad behavior described by the category. As an example, stereotypes in the general category *Structural Accessor* query an object's state, while the stereotype *predicate* more specifically returns a computed Boolean value. This Boolean characterizes some information about the state of the object on which the predicate method is called.

Methods may be labeled with one or more stereotypes. That is, methods may have a single stereotype from any category and may also have additional stereotypes from other categories. For example, a *get collaborator* is a *get* (accessor) method that uses an object of another class (e.g., return type). In addition, while considered an anti-pattern, a method can both query an object's state (*Structural Accessor*) and modify that state as well (*Structural Mutator*).

A freely available tool (see https://github.com/srcML/stereocode), *StereoCode*, analyzes and re-documents C++ source code (using srcML [17]) with the stereotype information for each method. *StereoCode* reports a superset of the stereotypes defined in TABLE I. For instance, a *get* method is required to be *const* and thus cannot be both a Structural Accessor and Structural Mutator. However, the tool will also identify *non_const_get* methods which, unlike a *get* method, can additionally mutate an object's state (i.e., can have an additional stereotype from the Structural Mutator category). An alternative tool for stereotype identification [8] that works only for Java programs is also available but appears to be less robust than *StereoCode*, hence the use of *StereoCode*.

TABLE I. Method stereotype taxonomy. Each Stereotype Category is listed with its set of Stereotypes.

Stereotype Category	Stereotype	Description
Structural Accessor	get	Returns a data member.
	predicate	Returns Boolean value that is not a data member.
	property	Returns info about data members.
	void-accessor	Returns information via a parameter.
Structural Mutator	set	Sets a data member.
	command	Performs a complex change to the object's state.
	non-void-command	
Creational	constructor, copy-constructor, destructor, factory	Creates and/or destroys objects.
Collaborational	collaborator	Works with objects (parameter, local or return value).
	controller	Changes only an external object's state (not *this*).
Degenerate	incidental	Does not read/change the object's state. No calls to other class methods.
	stateless	Does not read/change the object's state. One call to other class methods.
	empty	Has no statements.

III. Motivation and Related Work

Method stereotypes have been shown to be a useful and powerful abstraction of a method's role and behavior. Stereotypes have been used for numerous applications, including generating natural-language summaries [9][18], feature location [11], detecting potential design flaws [12], categorizing identifiers [13], and generating commit messages [14]. However, there has been no work examining the overall behavior of stereotypes and how they change as a system evolves. The closest to this work is Dragan et al. [16], which examines only added and deleted methods to categorize commits. Additionally, Dragan [19] also examined the distribution of method stereotypes in a few releases (~20) of two systems and found the distribution of method stereotypes to be fairly consistent for one system and unstable for the other.

The primary advantage to method stereotypes is that they summarize the role/responsibility of a method. Understanding how stereotypes evolve lays a stronger theoretical groundwork for current and any future research relying on stereotypes. It also provides a basis on top of which research can build improvements and additions to the stereotype model. To be specific, one way of measuring how well a model fits the data it is measuring is to analyze how well the model indicates normal behavior and how well it can indicate abnormal behavior. In this case, stereotypes are the model and methods are the data.

The goal is to determine if changes in stereotype indicate a benign or suspicious change to the stereotype's method. Showing that stereotypes do not typically change in response to method changes lends more credibility to stereotypes as a good model of the innate qualities of methods if we assume a method's behavior does not typically change drastically as a system evolves. Under this assumption, we then need to additionally show that when a stereotype changes due to a change to a method, this change indicates some sort of important activity. That is, if lack of change in stereotype indicates normal behavior then a stereotype change indicates abnormal behavior. This abnormal behavior may indicate a poorly designed method or inappropriate/incorrect changes to a method.

The result of this work contributes the following: 1) data on how method stereotypes evolve, which will support future research, 2) a basis to improve stereotypes as a model of methods and as their role in the context of a system, and 3) a preliminary study of how stereotypes may be used to alert developers of code smells and suspicious changes in general. Towards this, we first explore the question of whether stereotypes generally change when changes are made to methods (**RQ1**). In order to answer this, we mine the software repositories (complete history) of over 50 open-source and closed source systems (Section V and VI). We then use this in combination with a taxonomy (presented in Section IV) to investigate methods whose stereotypes do change, in order to see the type of changes that occur (**RQ2**) and whether those changes are benign or suspicious (**RQ3**). The overall goal is to understand consequences of changes in method stereotypes.

As a goal of the presented work is to evaluate if changes in method stereotype can be used to indicate the introduction of code smells, previous work on code smells and their detection is relevant. Mantyla et al. [20] present a taxonomy that categorizes similar bad smells and presents findings from an empirical study of using smells to detect software quality. Rani and Chhabra [21] present an empirical study on the distribution of select code smells over different versions of software projects. Tufano et al. [2][22] investigate the change history of 200 open-source projects to identify when code smells are introduced and if they are ever removed. The major difference between these works and ours is that they tend to target very specific forms of code and design smells, while our work focuses on the behavior of the methods (as represented by method stereotypes) themselves.

In [23], the use of change history was explored to detect code smells. Similar to this work, they use structural changes over time to identify a set of code smells defined within the paper. The primary difference between their work and ours is that they correlate specific types of changes and change patterns to the smell categories provided in their work whereas we do not do code-level change patterns, but method-behavior level.

Additional code smell detectors include: JDeodorant [24][25][26][27], Moha et al. [5] Sahin et al. [28], and Mansoor et al. [29]. These detectors have been used (such as in [21] and [2][22]) or can be used to investigate the evolution of code smells. The work provided here provides a complimentary and novel view based on the behavior represented by method stereotypes. Additional work on the mining software repositories includes commit categorization [30][31][32], bug prediction [12], and topic analysis [33].

IV. Evolution of a Method's Stereotype

As stereotypes describe a method's behavior and role at a high level, we are able to use changes in a method's stereotype to theorize about the consequences of such changes. We define a *stereotype transition* as a change in a method's stereotype caused by changes made to the method. Stereotype transitions can be indicators of design improvement or degradation. We use suspicious for a transition that indicates degradation and benign otherwise (improvement or neutral). For example, a change that causes a C++ method to transition from being a *non_const_get* to a *get* is an example of a benign transition, as it forbids modification of object state by the callee and allows *const* objects to use the method. Thus, it improves design by restricting undefined behavior and makes the system easier to maintain. If the opposite occurs, a transition in stereotype from a *get* method to a *non_const_get* method, we have an example of a suspicious change that indicates a degradation in the design of the system (i.e., *code smell*). Note, the addition of a *non_const_get* method is sometimes necessary in C++ and therefore valid. However, a change which replaces a *const* accessor (which is necessary for *const* objects) with a *non_const_get* method indicates that development is loosening restrictions on previously forbidden behavior, in addition to limiting the objects on which the method can be invoked. This type of change must be highlighted and reviewed, hence we term it *suspicious*.

TABLE II. contains our taxonomy of stereotype transitions. This taxonomy is not meant to be complete, but to highlight important transitions that have well-defined consequences. For each entry in the taxonomy there is a name/type of that classification of transitions, what types of stereotypes are involved in the transition, a description of the transition type, and lastly notes about the implications and significance of such transition and whether such transitions are generally benign or suspicious. Transitions that are not part of the taxonomy such as other changes within the same category are considered benign. This provides a partial answer to **RQ2,** and it provides a start for answering **RQ3**. We investigate this further in a manual investigation described in a following section.

The main goal of this study is to investigate how often stereotypes change (i.e., **RQ1**), and if they change, the frequency and patterns of changes, i.e., how often and what do they change to. To do this, the entire history of multiple systems is analyzed, and data collected on changes (or lack thereof) to method stereotypes. As the transition categories provided in TABLE II. are primarily conjectural, an investigation into the changes that induce a change in method stereotype is investigated manually (Section VI.C).

V. Data Collection

To investigate changes in method stereotypes, fifty open-source software systems, given in TABLE III. were selected for study. The subject systems are selected using the following criteria: 1) C++ is the primary language, 2) a minimum of 500 commits, 3) representative ranges of project size in terms of number of commits, and 4) representative sample of domains. The number of revisions (center column) are for each system and are non-merges that contain a modified C++ file.

TABLE II. TAXONOMY OF IMPORTANT STEREOTYPE TRANSITIONS. EACH TRANSITION TYPE HAS A STEREOTYPE CATEGORY, A DESCRIPTION WHICH INCLUDES THE METHOD STEREOTYPES THAT ARE PART OF THE TRANSITION, AND THE IMPLICATIONS OF THE TRANSITIONS TO THE DESIGN OF THE SYSTEM.

Transition Type	Stereotype Category	Description	Design Implications
Move to/from Unclassified	**Unclassified**	Method transitioned to/from Unclassified	• Methods that cannot be classified lack a clear abstraction • To unclassified (suspicious) • From unclassified (benign)
Move to/from Non_const_get	**Structural Accessor**	Method transitioned between non_const_get and another Accessor	• From Accessor to non_const_get breaks ability use on constants/degrades design (suspicious) • From non_const_get to Accessor increases information hiding. Method probably should have always been Accessor (benign)
Add Collaborational	**Collaborational**	A method adds a Collaborational stereotype or transitions from another category to Collaborational	• Addition of Collaborational indicates increased dependency to other object(s) (benign, but suspicious in large numbers)
Remove Collaborational		A method removes Collaborational or transitions from a Collaborational to another category	• Removal of Collaborational indicates decreased dependency to other object(s) (benign)
Add Degenerate	**Degenerate**	A method adds a Degenerate stereotype or transitions from another category to a Degenerate	• Method's functionality has been diminished • Indicates method does not have enough responsibility and consider removal (suspicous)
Remove Degenerate		A method removes a Degenerate stereotype or transitions from a Degenerate stereotype to one of another category	• Degenerate methods do not provide enough functionality • Removal is generally a positive (benign) • Indicates addition of more functionality or increased complexity
Cross Stereotype Boundaries	**Multiple Categories**	A change from Structural Accessor, Structural Mutator, and Creational to a different one of those categories	• Massive change to function behavior • Change is suspicious and should be investigated
Add/Remove/ Replace Categories		Method that has multiple stereotypes from at least two of the following: Structural Accessor, Structural Mutator, and Creational	• Method has too much responsibility • Presence is possible code smell • Transition that adds additional method stereotypes indicates degrade in design (suspicous) • Transition that removes stereotypes indicates a design improvement (benign) • When categories are replaced with others indicates poor design (suspicous)

The entire commit history of all systems is investigated providing a total of 445,255 revisions. The right-column in TABLE III. contains the location of repository. When the repository location is not a fully qualified URL, the repository comes from GitHub and has the following complete location: https://github.com/{Location}.git. The data collection process consists of the following parts:

1. Identification of all revisions and the files changed in those revisions.
2. Application of a syntactic differencer to the original and modified version of all changed C++ files to generate a fine-grained change log of each revision.
3. Analysis of the change logs to locate changed methods.
4. Collection of the original and modified stereotype of each changed method.

More specifically, data is collected for each system in the following manner. First, a local clone of the repository is made. Next, the syntactic difference of every modified C++ file for each revision of the default branch is computed (parts 1 and 2) using srcDiff [34]. In order to generate the change logs for each revision, a Python program generates a list of each revision and the modified C++ files contained within each of those revisions (part 1). To generate this list, the Python program uses the command git-log. With the correct options, git-log reports all modified files and their associated revisions. Since we are interested in changes to existing methods, added and deleted files are ignored since they can only contain new or removed methods. For similar reasons, copied and renamed files are also ignored. In short, we only use file reported by git-log as modified. In addition, git-log is set to not report merge revisions. As git-log reports modified files irrespective of contents, when processing the report provided by git-log, the Python program uses file extensions to identify C++ files and filter out non-C++ files. The typical file extensions for C++ are used (e.g., .cpp, hpp) with .h (a typical C-language extension that is still used in C++) treated as C++ code. Then, the Python program runs srcDiff on all the identified modified C++ files for each revision and generates a single change log for each revision (part 2).

Once the change log is created for each revision, the data can be examined for stereotype changes (parts 3 and 4). To examine, the change logs, a separate Python program we developed is used to parse the change log, identify and collect the signatures of changed methods (part 3), and then apply *StereoCode* to compute the original and modified method stereotypes for the changed method (part 4). A method is identified as changed and the signature collected if 1) it contains a change (added, deleted, or modified) to the text of the method, 2) the text changed is not whitespace or part of a comment, and 3) the method itself is not inserted or deleted.

Whitespace and comments (i.e., non-source code) changes are ignored, as they are non-functional/stylistic changes. In addition to the signature, for each changed method, the line number, class, and file containing the method are also collected. These additional attributes are collected to generate a unique ID. After information for each changed method is collected, the original and modified source code versions for that revision are converted to srcML using tools provided at srcML.org.

TABLE III. SELECTED SOFTWARE SYSTEMS ORDERED BY NUMBER OF REVISIONS. KLOC IS THE CODE OF THE MOST RECENT VERSION, # REV IS NUMBER OF REVISIONS WITH MODIFIED C/C++ FILES. LOCATIONS HAVE A URL OF HTTPS://GITHUB.COM/{LOCATION}.GIT, EXCEPT FOR CLANG.

System	KLOC	# Rev	Location
kokkos	141	745	kokkos/kokkos
tera	114	832	baidu/tera
json	62	1,049	nlohmann/json
griefly	21	1,074	griefly/griefly
CRYENGINE	2,337	1,092	CRYTEK/CRYENGINE
MultiMC5	73	1,267	MultiMC/MultiMC5
ChaiScript	33	1,312	ChaiScript/ChaiScript
Mantella	3	1,320	Mantella/Mantella
GamePlay	300	1,354	gameplay3d/GamePlay
Plasma	686	1,395	H-uru/Plasma
antimony	106	1,737	mkeeter/antimony
Rcpp	109	1,947	RcppCore/Rcpp
engine	155	2,078	flutter/engine
oiio	165	2,445	OpenImageIO/oiio
rdkit	252	2,474	rdkit/rdkit
distortos	880	2,498	DISTORTEC/distortos
nix	39	2,704	NixOS/nix
CTK	252	2,823	commontk/CTK
irods	225	2,860	irods/irods
citra	107	2,908	citra-emu/citra
BansheeEngine	344	3,058	BearishSun/BansheeEngine
newton-dynamics	1,678	3,124	MADEAPPS/newton-dynamics
gnuradio	262	3,191	gnuradio/gnuradio
folly	238	3,277	facebook/folly
supertux	97	4,283	SuperTux/supertux
Dlib	479	4,366	davisking/dlib
ola	211	4,470	OpenLightingProject/ola
tfs-old-svn	89	4,665	otland/tfs-old-svn
Urho3D	959	5,225	urho3d/Urho3D
ogre	639	5,280	OGRECave/ogre
appleseed	304	5,369	appleseedhq/appleseed
bitcoin	148	5,785	bitcoin/bitcoin
openFrameworks	190	6,620	openframeworks/openFrameworks
codeblocks	315	7,484	jenslody/codeblocks
codelite	812	7,918	eranif/codelite
QuantLib	497	8,526	lballabio/QuantLib
stellarium	238	8,677	Stellarium/stellarium
Natron	702	10,064	MrKepzie/Natron
opencv	937	11,328	opencv/opencv
cocos2d-x	997	13,890	cocos2d/cocos2d-x
ppsspp	483	15,272	hrydgard/ppsspp
kdevelop	162	16,663	KDE/kdevelop
Dealii	2,463	20,340	dealii/dealii
blender	1,213	24,215	git://git.blender.org/blender.git
xbmc	980	25,280	xbmc/xbmc
Mongo	3,122	26,345	mongodb/mongo
Qt	5,228	28,069	qt/qt
swift	719	30,343	apple/swift
cgal	1,678	39,583	CGAL/cgal
Clang	1,571	56,631	http://llvm.org/git/clang
Total	33,820	445,255	

Then, *StereoCode* is used on the original and modified code to compute the stereotype of each method. *StereoCode* reports these in a CSV format containing the method stereotype and the same ID information collected for the changed methods (line number, file name, class name, and method signature). This is used to lookup the original and modified stereotypes. *StereoCode* is applied to the entire file as opposed to just the each version of the method to avoid error due to any dependency *StereoCode* may have on contextual information.

Finally, information is recorded in two ways. First, each stereotype and ID information pair is recorded. Second, running totals on the counts of the number of times each stereotype transition occurred and when each stereotype did not change are updated. All programs written for the data collection process were tested and verified for accuracy.

VI. RESULTS

Data was collected from all investigated systems. Empirical results are presented in Section A. In Section B we compare the results of Section A to that from six additional closed-source systems, and the manual investigation is presented in Section C.

A. *Empirical Investigation*

Results on all method changes for all systems is now presented. The complete dataset is available at http://www.sdml.cs.kent.edu/stereodiff/. The data is analyzed on all systems as a whole as well as on each system individually. Due to space restrictions, we report the total occurrences out of all systems and the median percentage. The median percentage is computed by calculating the percentages individually for each system and then taking the median. We computed median because the data is not uniform. The number of revisions/method changes for each system is not uniform (some systems contribute significantly more) and computing percentages of on all systems collectively can be affected by this disproportion. However, for the data we collected, the results are similar with either method.

Over all systems, there is a total of 1,361,348 changes to methods. The vast majority, 1,233,645 (89.9%), of the changes to methods resulted in no change in the stereotype. That is, the stereotype of a method is very stable over time. Of the method changes in these systems, only 127,603 (10.1%) result in a change in stereotype, with the percentage on individual systems ranging from 4.5% to 29.0%.

TABLE IV. presents the top-ten most commonly changed methods grouped by their stereotype. In all cases, the changes that *did not* impact the stereotype. Of all method changes, the percentage drops sharply from 41.9%, for *command collaborator*, down to 0.7% for *unclassified*, with 50% either *command collaborator* or *non_void_command collaborator*.

The top-ten method changes where the stereotype *changed* (transitions) are shown in TABLE V. We found over 2,000 unique transitions in total, many of which occur a small number of times. For the most part, the transitions are benign according to the taxonomy (e.g., *unclassified* to *command*). The main type of transition that is suspicious is *non_void_command collaborator* to *property collaborator* which indicates a significant deviation in behavior.

Also of note in TABLE V. the percentage of each transition is very low (i.e., < 1%). Collectively, out of all method changes, stereotype transitions only occur 10.1% of the time, with each individual transition type occurring with a small percentage. With this, we can now answer **RQ1**. Method stereotypes, as a whole, change infrequently due to changes to methods and thus, a method's stereotype is very stable over evolution.

TABLE IV. TOP TEN MOST COMMON CHANGES TO METHODS THAT DID NOT CHANGE THE STEREOTYPE (GROUPED BY STEREOTYPE), MEDIAN OF THE PER-SYSTEM PERCENT OUT OF ALL METHOD CHANGES, AND THE NUMBER CHANGED.

Stereotype	Median %	Count
command collaborator	41.9%	595,085
non_void_command collaborator	8.1%	164,289
command	4.9%	65,473
property collaborator	4.1%	76,635
collaborator	4.0%	63,473
get collaborator	1.7%	23,614
void_accessor collaborator	1.7%	49,312
non_void_command collaborator factory	1.0%	40,183
predicate collaborator	1.0%	21,425
unclassified	0.7%	10,391

TABLE V. TOP TEN STEREOTYPE TRANSITIONS OUT OF OVER 2,000 UNIQUE POSSIBLE TRANSITIONS. THE STEREOTYPE OF THE METHOD CHANGED FROM ITS ORIGINAL STEREOTYPE TO THE MODIFIED STEREOTYPE, WITH THE MEDIAN OF THE PER-SYSTEM PERCENT OUT OF ALL METHOD CHANGES.

Original Stereotype	Modified Stereotype	Median %	Count
command	***command collaborator***	0.5%	6,327
collaborator	***command collaborator***	0.5%	7,633
command collaborator	***collaborator***	0.3%	4,761
command collaborator	***command***	0.3%	4,358
collaborator	***non_void_command collaborator***	0.1%	2,407
set collaborator	***command collaborator***	0.1%	1,907
unclassified	***command***	0.1%	1,519
non_void_command collaborator	***collaborator***	0.1%	1,645
set	***command***	0.1%	1,567
non_void_command collaborator	***property collaborator***	0.1%	1,889

In order to answer **RQ2**, let us now take a closer look at stereotype transitions. Over, 65% of the time (median, with totals of 1,245/1,742) that a *non_const_get* method changed stereotype, it changed to a *get*. In fact, it is more common for a *non_const_get* method to change into a *get* method, as it is to remain a *non_const_get* (i.e., individually it is not stable). The opposite (*get* to *non_const_get*) is exceedingly rare (22 total instances over all projects and revisions). That is, *get* methods are mistakenly written as *non_const_get* methods and then corrected at a later date.

When a method-stereotype transition occurs, 37,183 (29.7%) of the time, one or more stereotypes are inserted and 30,092 (22.2%) of the time one or more stereotypes are deleted. A summarization of the top-ten deleted and inserted groups of stereotypes are in TABLE VI. and TABLE VII. respectively. In both, the stereotype inserted or deleted is given with the median of the per-system percent, and the number of occurrences. For both, the most common insertion or deletion is that of a single stereotype, with *collaborator* being most common. When just considering method stereotypes that purely inserted or deleted stereotypes, this points to increased collaboration between classes in a system over time (the remaining pure insert and delete data that is not shown here does not offset this trend).

The remaining times a stereotype transition occurred it completely changed from one stereotype to another: 24,859 (19.8%), or partially-changed (one or more stereotypes replaced with one or more other stereotypes with at least one stereotype remaining in common to both): 35,469 (27.4%). TABLE VIII. and TABLE IX. provide the top-ten completely changed method stereotypes and partially-changed stereotypes, respectively.

For TABLE VIII. the change from the original stereotype to the modified stereotype, the median of the per-system percentage (with percent individually out of total amount of transitions in that system), and the number of occurrences is given. The most frequently completely changed stereotypes are *unclassified* to *command* (benign), *set* to *command* (within same stereotype category), and *unclassified* to *collaborator* (benign). Out of all the top ten, the only suspicious transitions are those that are no longer able to be classified by *StereoCode* (*unclassified*), which may indicate that the methods have become convoluted, lacking a clear abstraction (e.g., long method code smell).

TABLE VI. TOP TEN DELETED STEREOTYPES. MEDIAN % IS THE MEDIAN OF THE PER-SYSTEM PERCENT WITH THE PERCENT INDIVIDUALLY OUT OF ALL TRANSITIONS IN THAT SYSTEM.

Deleted Stereotype	Median %	Count
collaborator	5.3%	8,496
stateless	5.1%	7,498
command	3.4%	4,859
non_void_command	1.4%	2,283
non_const_get	1.0%	1,891
collaborational_command	0.7%	1,280
factory	0.5%	1,479
empty	0.1%	356
set	0.1%	269
collaborational_command stateless	0.1%	213

TABLE VII. TOP TEN INSERTED STEREOTYPES. MEDIAN % IS THE MEDIAN OF THE PER-SYSTEM PERCENT WITH THE PERCENT INDIVIDUALLY OUT OF ALL TRANSITIONS IN THAT SYSTEM.

Inserted Stereotype	Median %	Count
collaborator	10.1%	12,532
command	5.5%	7,705
stateless	3.6%	5,263
non_void_command	2.4%	3,618
non_const_get	0.8%	1,423
collaborational_command	0.6%	827
property	0.5%	1,037
factory	0.3%	1,000
predicate	0.2%	415
set	0.1%	220

In TABLE IX. the most common partially-changed stereotypes are *set* with *command* (benign) and *non_void_command* with *property* (suspicious as it may be adding a missing *const* or indicate a more serious problem). The remainder are largely partially-changed from within the same category (benign) with the exception of *empty* to *command* and *non_const_get* to *get* (benign); and *command* to *void-accessor,* which crosses categories (suspicious).

In summary, methods do not change stereotype very often. When they do change, most of the frequent types of changes are of relatively little concern. There are, however, a few among the top-ten that are worth investigating. Due to their relative rarity, an automatic detection and reporting/alerting tool would be of great use. Such a tool can easily be integrated into an IDE or versioning system.

TABLE VIII. TOP TEN COMPLETELY CHANGED STEREOTYPES. FROM THE ORIGINAL TO MODIFIED STEREOTYPE. MEDIAN IF OF THE PER-SYSTEM PERCENT (PERCENT OF ALL TRANSITIONS IN THAT SYSTEM).

Original Stereotype	Modified Stereotype	Median %	Count
unclassified	***command***	1.2%	1,519
set	***command***	1.2%	1,567
unclassified	***collaborator***	0.8%	1,077
command	***unclassified***	0.7%	955
command	***set***	0.6%	856
non_const_get	***get***	0.5%	1,245
collaborator	***unclassified***	0.3%	890
stateless	***unclassified***	0.2%	506
unclassified	***command collaborator***	0.2%	420
empty	***command collaborator***	0.2%	531

TABLE IX. TOP TEN PARTIALLY-CHANGED STEREOTYPES. MEDIAN IS OF THE PER-SYSTEM PERCENT (PERCENT OF ALL TRANSITIONS IN THAT SYSTEM).

Original Stereotype	Modified Stereotype	Median %	Count
set	***command***	1.4%	1,907
non_void_command	***property***	1.3%	2,500
command	***non_void_command***	1.1%	1,519
get	***property***	0.9%	1,861
command	***set***	0.9%	1,051
non_void_command	***command***	0.8%	3,003
property	***get***	0.8%	1,212
non_const_get	***get***	0.6%	1,344
command	***void_accessor***	0.5%	1,147
empty	***command***	0.4%	895

B. *Closed Systems*

We additionally performed an empirical study of 6 closed-source systems (TABLE X.) to understand generalizability of the open-source results. Among the 53,876 method changes in the 6 closed-source systems, 49,067 (86.7%) do not change method stereotype and 4,809 (13.2%) do change method stereotype (varying between 0-20%). The number of transitions in the closed systems is slightly higher, however, a larger selection of systems is needed to see if this trend holds. When both data sets are combined, less than 10.4% of all method changes resulted in a method-stereotype change.

Similar patterns are found for method changes that do not change method stereotype (TABLE XI.). That is, a few stereotypes (first three are identical to those in open-source system) occur with higher-frequency and the values drop quickly. Likewise, all stereotype transitions are uncommon (<1%) with 8/10 stereotypes in common between the open-source and closed systems (although in a different order). Also consistent with the open-source results, all but one transition is benign. The one suspicious transition changed the method to be both an accessor and mutator (*non_void_command collaborator* to *non_void_command non_const_get collaborator*).

As for *get* and *non_const_get* methods, as in the open-source systems *non_const_get* is not stable (i.e., changes stereotypes frequently). However, both are even more unstable than in the open-source systems. *get* methods did not change 15.7% (20/58) of the time and *non_const_get* did not change 14.3% (57/132) of the time. In contrast, in the open-source systems *get* did not change (i.e., is stable) 70.9% (4,781/6,821) of the time and *non_const_get* is stable 31.9% (1,396/3,138). Although *non_const_get* transitioned to *get* 40/132 times, unlike the open-source data, the median of the per-system percent is zero. More consistent with the open-source systems, although *get* is not stable in closed systems, it never changed to a *non_const_get*.

Regarding how method stereotypes change, when a stereotype transition occurs, it is completely changed 16.7% (682) of the time; only consisted of deleted stereotypes 23.2% (1140) of the time; only consisted of inserted stereotypes 40.9% (1,717) of the time; and had stereotypes partially changed 20.5% (1,270) of the time. In comparison to the open-source systems, the median of inserting new stereotypes is more frequent by over 10% of the median values while partial/complete change of stereotypes medians are less (~6.9%/~3.1%).

TABLE X. CLOSED SOURCE SYSTEMS IDENTIFIED WITH ANONYMOUS NAMES. KLOC IS C++ CODE FOR THE MOST RECENT VERSION INVESTIGATED USING WC, AND # REV IS NUMBER OF REVISIONS WITH MODIFIED C++ FILES.

System	KLOC	# Rev
EC	295	1,539
FASA	174	1,546
G	430	1,969
H	654	5,359
On	2,046	34,427
Op	995	315
Total	4,595	45,155

TABLE XI. TOP TEN MOST COMMON CHANGES TO METHODS THAT DID NOT CHANGE THE STEREOTYPE (GROUPED BY STEREOTYPE) FOR THE CLOSED-SOURCE SYSTEMS, MEDIAN OF THE PER-SYSTEM PERCENT IS OUT OF ALL METHOD CHANGES, AND THE NUMBER CHANGED.

Stereotype	Median %	Count
command collaborator	38.8%	22,476
non_void_command collaborator	7.4%	17,207
command	4.9%	1,535
collaborator	4.1%	1,597
property collaborator	1.0%	229
non_void_command collaborator factory	0.7%	360
non_const_get collaborator	0.6%	407
set collaborator	0.5%	215
unclassified	0.4%	224
collaborational_command collaborator	0.4%	439

Considering instances where the only a change in stereotype is the insertion or deletion of one or more stereotypes, the closed systems (similar to open-source systems) show a larger increase in *collaborator* indicating increased collaboration. In contrast, the closed systems have less variability in what stereotypes are inserted/deleted, with only 7/9 types occurring in the closed systems with a median greater than zero (open-source has 13 deleted and 17 inserted).

Similar to the open-source systems, the majority of the top-ten completely changed stereotypes are benign except two that are not able to be classified (*command* to *unclassified* and *collaborator* to *unclassified*). Likewise, partial changes are similar and mostly in the same category (benign). There are two other benign transitions (*non_const_get* to *get* and *stateless* to *command*), and two suspicious transitions (*non_void_command* to *non_const_get* and *command* to *empty*).

Ultimately, there is some variation in the specific details of the two data sets. It is worth investigating in the future if a larger set of closed systems will produce results more consistent with the open-source systems. However, although there are differences between the two data sets, the conclusions are largely the same: stereotypes transition occur infrequently, and when they do, few among the top-ten that are worth investigating. Thus, automatic detection and reporting will also be valuable to closed systems.

C. Manual Investigation

In Section IV, we presented a taxonomy on the consequences of a change to a method's stereotype. In Section VI.A, we show that method stereotypes as a whole are stable with regards to method change with a method's stereotype changing 10% of the time in the 50 open source systems we examined. In addition, the most frequent changes in method stereotype are generally safe and unsuspicious.

In order to answer **RQ3**, we describe the results of a manual investigation of stereotype transitions. The approach taken is as follows. First, a list is constructed containing all the method-stereotype transitions that occur in the subject systems (TABLE III.) and the frequency for each is computed. Based on the behavior of method-stereotypes as described by Dragan et al. [8], an author who is an expert on method stereotypes examined every method-stereotype transition (the expert had no knowledge of TABLE II.) and they noted all the transitions that should be investigated (i.e., was a possible code smell).

Then, for each transition that occurred more than ten times in the revision history of all 50 systems, a separate expert developer examined an instance of each of the method-stereotype transitions noted previously. For each instance, detailed notes are taken about the change in method stereotype and the source code modified. When necessary, additional information is consulted such as related commits and their changes. This information is then used to make a determination of whether the change in method stereotype is of *benign* (no concern/positive) or *suspicious* (the method change is worth inspection by a developer) such as when the change to the method is clearly wrong or revealed issues with the design and maintainability of the method/class.

This type of manual investigation is a slow and tedious process. In order to make the investigation process more manageable, a syntactic source-code difference (i.e., srcDiff) supporting a unified view of changes and *StereoCode* were used to examine the changed methods.

In total, 33 method-stereotype changes were examined in this level of detail. The investigation took approximately 20-person hours. TABLE XII. presents a summary of the results with each of the stereotype transitions grouped according to the taxonomy presented in Section IV. *Other* consists of insignificant changes not included in the taxonomy (four instances of changes within the same stereotype category and one *command collaborator* to *collaborator*). In addition, one method change included both the *Cross Stereotype Boundaries* and *Multiple Category Stereotype*, this is counted as part of both (hence the count adds to 34 but one method is in the table twice and we give 33 as the total).

For each of the categories present, a count of the number of transitions belonging to those categories along with a count and percentage of the number of times those changed methods are *benign* or *suspicious*.

A *benign* change is typically a neutral change or one that fixes a bug or code smell. *Suspicious* change is a change that is clearly wrong or indicates a problem with the function. With corrections for the one change counted twice, 32% of changes in method stereotype are *suspicious*, while 68% are *benign*.

In agreement with the taxonomy in TABLE II. *From Non_const_get* (convert from *non_const_get* to *get*), *Remove Collaborational*, *Remove Degenerate*, and *Other* are benign indicators. The one instance of stereotype becoming unclassified is suspicious (method is poorly designed). *Cross Stereotype Boundaries*, and *Add Categories/Remove Categories* (both those that become ones and those that lost the multiple categories) are mixed. That is, the method-stereotype change belonging to the *Cross Stereotype Boundaries* and *Add/Remove/Replace* transition types are indicators of problems with the method design or change, but only a portion of the time.

TABLE XII. SUMMARY OF MANUAL INVESTIGATION. FOR EACH METHOD-STEREOTYPE TRANSITION TYPE THE TABLE STATES THE OVERALL NUMBER OF OCCURRENCES. THIS IS BROKEN INTO NUMBER AND PERCENTAGE OF OCCURRENCES FOR BOTH SUSPICIOUS CHANGES (I.E., SHOULD BE INVESTIGATED FURTHER) AND BENIGN CHANGES (I.E., ARE OF NO CONCERN)..

Type	Overall	Suspicious		Benign	
	#	#	%	#	%
To Unclassified	1	1	100%	0	0%
From Non_const_get	1	0	0%	1	100%
Remove Collaborational	1	0	0%	1	100%
Remove Degenerate	2	0	0%	2	100%
Add Categories	8	4	50%	4	50%
Remove Categories	4	2	50%	2	50%
Cross Stereotype Boundaries	11	4	36%	7	64%
Other	6	0	0%	5	100%
Total	33	11	32%	23	68%

Fig. 1 and Fig. 2 give examples of a benign change and suspicious change, respectively. For both figures, the method is annotated with the changes in a unified-view. Inserted code is marked with a green background color, while code common to both versions is left with a plain-white background. No deleted code is present in the examples.

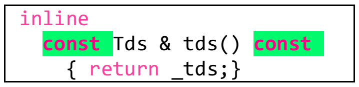
In Fig. 1, the developer makes a *non_const_get* method into a *const get* method, thus disallowing any indirect modification to the class data member and making the system easier to maintain. In agreement with the taxonomy, we regard this as a benign change.

In Fig. 2, the developer makes an initialize method *const*, however, to make it compile, the developer also made several data members of the class *mutable*, and thus making the system more difficult to maintain. To make matters worse, a comment by the developer reveals that they were baffled that the code worked previously. Clearly, this individual does not understand the language concepts of *const* or *mutable* very well. This is an obvious mistake. In agreement with the taxonomy, we regard this as a suspicious change.

From this, we can conclude that although not perfect, certain categories of method-stereotype changes (To Unclassified, Cross Stereotype Boundaries, and Add/Remove/Replace Categories) can be useful indicators for method-design problems or inappropriate method changes. Since changes in method stereotype are uncommon (with specific categories being even less common), investigation by a developer will not require substantial time.

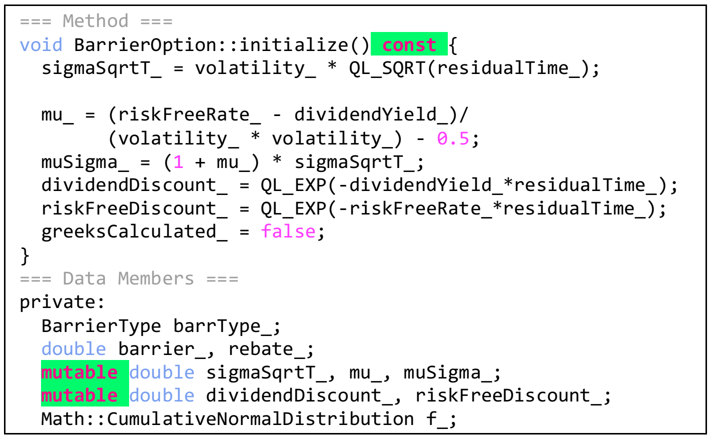
From the manual investigation, we note that a finer-grained categorization may produce more accurate predictions. For example, many of the *Cross Stereotype Boundaries* did so because method *constness* is added/removed. The addition of *const*, allowed *StereoCode* to make a correct assessment of behavior and are benign changes, while the removal of *const* is suspicious. That is, refining *Cross Stereotype Boundaries* into cases where adding *const* or removing *const* will increase prediction ability. Similar types of improvements may be made to *Add/Remove/Replace Categories*.

With this we finish answering **RQ2** and **RQ3**. That is, changes in method stereotypes are statistically uncommon. When they do change, many categories of changes are of little concern. However, specific categories of changes in method stereotypes (*To Unclassified*, *Cross Stereotype Boundaries,* and *Add/Remove/Replace Categories*) can indicate code smells, that is method design problems and inappropriate changes to code.



```
inline
  const Tds & tds() const
    { return _tds;}
```

Fig. 1. Example of a *benign* change. Method stereotype changes from a *non_const_get* method (Structural Accessor) to a *get* method (Structural Accessor). Developer simply added const to return type and method.



```
=== Method ===
void BarrierOption::initialize() const {
  sigmaSqrtT_ = volatility_ * QL_SQRT(residualTime_);

  mu_ = (riskFreeRate_ - dividendYield_)/
        (volatility_ * volatility_) - 0.5;
  muSigma_ = (1 + mu_) * sigmaSqrtT_;
  dividendDiscount_ = QL_EXP(-dividendYield_*residualTime_);
  riskFreeDiscount_ = QL_EXP(-riskFreeRate_*residualTime_);
  greeksCalculated_ = false;
}
=== Data Members ===
private:
  BarrierType barrType_;
  double barrier_, rebate_;
  mutable double sigmaSqrtT_, mu_, muSigma_;
  mutable double dividendDiscount_, riskFreeDiscount_;
  Math::CumulativeNormalDistribution f_;
```

Fig. 2. Example of a suspicious change. Method stereotype changes from command (Structural Mutator) to void-accessor (Structural Accessor). Developer made an initialize method const and the data members it modifies mutable.

VII. Threats to Validity

Only standard C++ extensions are used to locate C++ files. Most of the projects use only these extensions, however, some use unusual extensions which possibly contained C++ code. For example, a C++ extension followed by .in, which possible denotes a file that is used to generate code as part of the build system, and .tpp, which can possible contain template code. These are only noticed in a few of the repositories. For future work, these files can be examined more closely and a determination made as to whether they should be included as part of the data collected.

The .h extension is treated as C++, however, it is used both as a C and C++ header extension. Some of the projects included both C and C++ code and use .h for both. As C++ is largely a super-set of C (with a few exceptions), the parsing of C code as C++ code is a minor threat.

Some of the projects have code committed that is auto-generated by some tooling. Auto-generated files may exist throughout the history of a project or may appear and disappear at any point in the history of the software. For the most part, we are uninterested in changes to auto-generated code, however, as they can appear at any point in history, they can be difficult to identify. Investigation into auto-detection of all generated files through the history of projects is a valid research topic. In our study we found two, one each in *ppsspp* and *codelite*, that contained auto-generated files, these were ignored.

Only C++ is used as part of the study. Results may be somewhat different for other languages. Currently, *StereoCode* only supports C++ and is thus a limitation.

One threat is that the amount of revisions contributed by each project is not uniform. That is, large systems such as Clang (over 56K revisions) contribute much more than a much smaller project such as Tera (832 revisions), and the distribution of these may be different between them. Mitigating this, we normalized the data per system and present those results. In addition, the largest project (Clang), contributed less than 13% of the total number of revisions and thus no one system dominated the results. In addition, in a preliminary run of the experiment that differed by well over 100K revisions, the authors found very similar results. That is, even when different projects are investigated the results are very much the same.

Lastly, manual investigation is subject to human error. Mitigating this, the process was done carefully, and detailed notes taken. When the encoder was unsure about the change, another expert programmer was consulted. In addition, a breadth instead of depth look at the changes in method stereotypes is performed. One sample is certainly not representative of every possible change that could induce that particular change in stereotype. That is, we can only conclude that changes in method stereotype can indicate problems, however, we cannot give the extent to which they do.

VIII. Conclusions

The work presented examined how methods change in the face of evolution. Specifically, we examine how the stereotype of a method (an abstraction of a method's role and behavior) changes as the method is changed. The version histories of 50 open source systems were examined. Out of the approximately 1.4 million changes to methods, we found that the stereotype of a method rarely changed. About 90% of changes to methods resulted in no change to its stereotype. Hence, we can conclude that a method does not often change in original intent and are quite stable in that regard during the lifetime of a system. The 6 closed-source systems display similar characteristics.

However, this gives question to the remaining ~10% of changes that do impact the stereotype. Further manual examination of these cases show that certain categories of method-stereotype changes can be indicators of poor method design or of inappropriate changes to a method. A categorization of various stereotype transitions and their potential to impact the design is also proposed and supported by evidence from the manual investigation.

While method stereotypes are a powerful descriptor of method role/behavior they will need to be combined with other analysis techniques to uncover other code smells. Since a method stereotype can be automatically and quickly computed, these results could be easily integrated into a development process to notify developers or project managers of potentially problematic changes to a method; potentially triggering code reviews or other precautionary measures.

In this study, we investigate how individual methods evolve and not a project as a whole. As such, we did not investigate benign cases (e.g., *Add Collaborational*) that can be suspicious in large numbers. For future work, we will look at how collections of method stereotype changes can be used as indicators. Additionally, we would like to take this work further to explore how we can use stereotype transitions in combination with log and structural information to improve upon past research using commit history to predict/analyze code health.

This work was supported in part by a grant from the US National Science Foundation CNS 13-05292/05217.

References

[1] B. W. Boehm, *Software Engineering Economics*, vol. 197. Prentice Hall PTR, 1981.

[2] M. Tufano *et al.*, "When and Why Your Code Starts to Smell Bad," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 1, pp. 403–414.

[3] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[4] R. Marinescu, "Detection Strategies: Metrics-based Rules for Detecting Design Flaws," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 350–359.

[5] N. Moha, Y. G. Gueheneuc, L. Duchien, and A. F. L. Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan. 2010.

[6] F. A. Fontana, M. Zanoni, A. Marino, and M. V. Mäntylä, "Code Smell Detection: Towards a Machine Learning-Based Approach," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 396–399.

[7] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," presented at the 22nd IEEE International Conference on Software Maintenance (ICSM'06), 2006, pp. 24–34.

[8] L. Moreno and A. Marcus, "JStereoCode: Automatically Identifying Method and Class Stereotypes in Java code," presented at the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 2012.

[9] N. J. Abid, N. Dragan, M. L. Collard, and J. I. Maletic, "Using Stereotypes in the Automatic Generation of Natural Language Summaries for C++ Methods," presented at the IEEE International Conference on Software Maintenance and Evolution (ICSME'15), 2015, pp. 561–565.

[10] P. Andras, A. Pakhira, L. Moreno, and A. Marcus, "A Measure to Assess the Behavior of Method Stereotypes in Object-Oriented Software," in *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 2013, pp. 7–13.

[11] N. Alhindawi, J. I. Maletic, N. Dragan, and M. L. Collard, "Improving Feature Location by Enhancing Source Code with Stereotypes," presented at the 29th IEEE International Conference on Software Maintenance (ICSM'13), 2013, pp. 1–10.

[12] G. Canfora and L. Cerulo, "Impact Analysis by Mining Software and Change Request Repositories," presented at the 11th IEEE International Symposium on Software Metrics (METRICS'05), 2005, pp. 29–37.

[13] C. D. Newman, R. S. AlSuhaibani, M. L. Collard, and J. I. Maletic, "Lexical Categories for Source Code Identifiers," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 228–239.

[14] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, "ChangeScribe: A Tool for Automatically Generating Commit Messages," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, vol. 2, pp. 709–712.

[15] N. Dragan, M. L. Collard, and J. I. Maletic, "Automatic Identification of Class Stereotypes," presented at the IEEE International Conference on Software Maintenance (ICSM'10), 2010, pp. 1–10.

[16] N. Dragan, M. L. Collard, M. Hammad, and M. I. Maletic, "Using Stereotypes to Help Characterize Commits," presented at the 27th IEEE International Conference on Software Maintenance (ICSM'11), 2011, pp. 520–523.

[17] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," presented at the 11th IEEE Interational Conference on Source Code Analysis and Manipulation, 2011, pp. 173–184.

[18] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic Generation of Natural Language Summaries for Java Classes," in *21st International Conference on Program Comprehension (ICPC'13)*, 2013, pp. 23–32.

[19] Natalia Dragan, "The Emergent Laws of Method and Class Stereotypes in Object Oriented Software," Ph.D. Dissertation, Kent State University, Kent, Ohio USA, 2010.

[20] M. Mantyla, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in *International Conference on Software Maintenance (ICSM'03)*, 2003, pp. 381–384.

[21] A. Rani and J. K. Chhabra, "Evolution of Code Smells Over Multiple Versions of Softwares: An Empirical Investigation," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, 2017, pp. 1093–1098.

[22] M. Tufano *et al.*, "When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away)," *IEEE Trans. Softw. Eng.*, vol. 43, no. 11, pp. 1063–1088, Nov. 2017.

[23] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting Bad Smells in Source Code Using Change History Information," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, Piscataway, NJ, USA, 2013, pp. 268–278.

[24] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Type-Checking Bad Smells," in *2008 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 329–331.

[25] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "JDeodorant: Identification and Application of Extract Class Refactorings," in *Proceedings of the 33rd International Conference on Software Engineering*, New York, NY, USA, 2011, pp. 1037–1039.

[26] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou, "JDeodorant: Identification and Removal of Feature Envy Bad Smells," in *2007 IEEE International Conference on Software Maintenance*, 2007, pp. 519–520.

[27] D. Mazinanian, N. Tsantalis, R. Stein, and Z. Valenta, "JDeodorant: Clone Refactoring," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, 2016, pp. 613–616.

[28] D. Sahin, M. Kessentini, S. Bechikh, and K. Deb, "Code-Smell Detection As a Bilevel Problem," *ACM Trans Softw Eng Methodol*, vol. 24, no. 1, pp. 6:1–6:44, Oct. 2014.

[29] U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-Objective Code-Smells Detection Using Good and Bad Design Examples," *Softw. Qual. J.*, vol. 25, no. 2, pp. 529–552, Jun. 2017.

[30] A. Hindle, D. M. German, M. W. Godfrey, and R. C. Holt, "Automatic Classication of Large Changes into Maintenance Categories," in *2009 IEEE 17th International Conference on Program Comprehension*, 2009, pp. 30–39.

[31] L. P. Hattori and M. Lanza, "On the Nature of Commits," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, Piscataway, NJ, USA, 2008, pp. III–63–III–71.

[32] J. J. Amor, G. Robles, J. M. Gonzalez-Barahona, and A. Navarro, "Discriminating Development Activities in Versioning Systems: A Case Study," in *Proceedings PROMISE*, 2006, vol. 2006, p. 2nd.

[33] A. Hindle, M. W. Godfrey, and R. C. Holt, "What's Hot and What's Not: Windowed Developer Topic Analysis," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 339–348.

[34] Michael John Decker, "srcDiff: Syntactic Differencing to Support Software Maintenance and Evolution," Dissertation, Kent State University, 2017.