AN EMPIRICAL INVESTIGATION OF METHOD AND CLASS STEREOTYPE DISTRIBUTIONS ACROSS SOFTWARE SYSTEMS IN C++, C#, AND JAVA

A thesis submitted to

Kent State University in partial

fulfillment of the requirements for the

Degree of Master of Computer Science

by

Ali Al-Ramadan

May 2024

© Copyright

All rights reserved

Except for previously published materials

Thesis written by

Ali Al-Ramadan

B.S., Yarmouk University, Jordan, 2022

M.S., Kent State University, USA, 2024

Approved by

_____, Advisor
Jonathan Maletic

_____, Chair, Department of Computer Science
Javed Khan

_____, Dean, College of Arts and Sciences
Mandy Munro-Stasiuk

TABLE OF CONTENTS

TABLE OF CONTENTS III

LIST OF FIGURES V

LIST OF TABLES VII

ACKNOWLEDGEMENTS VIII

CHAPTER 1 INTRODUCTION 1

1.1 Motivation 2

1.2 Problem Statement 2

1.3 Contributions 3

1.4 Organization of the Thesis 3

CHAPTER 2 BACKGROUND AND RELATED WORK 5

2.1 Method Stereotypes 5

2.2 Class Stereotypes 8

2.3 srcML Infrastructure 10

2.4 Stereocode Tool 11

2.5 Applications of Stereotypes 12

CHAPTER 3 STEREOCODE: A MULTI-LANGUAGE TOOL FOR AUTOMATED STEREOTYPE DETECTION 14

3.1 Overview 14

3.2 Converting Source Code Input to srcML 16

3.3 Collecting Program Information 17

3.4 Computing Method Stereotypes 21

3.4.1 Stereotype: Get 21
3.4.2 Stereotype: Predicate 23
3.4.3 Stereotype: Property 24
3.4.4 Stereotype: Void-Accessor 25
3.4.5 Stereotype: Set 26
3.4.6 Stereotype: Command 27
3.4.7 Stereotype: Factory 28
3.4.8 Stereotype: Collaborator 29
3.4.9 Stereotype: Controller 30
3.4.10 Stereotype: Stateless 31
3.4.11 Stereotype: Wrapper 32
3.4.12 Stereotype: Empty 33
3.5 Computing Class Stereotypes 33
3.6 Outputting the Stereotypes 34
CHAPTER 4 STEREOCODE EVALUATION 36
CHAPTER 5 EMPIRICAL STUDY 48
5.1 Data Collection and Processing 49
5.2 Results and Discussion 54
CHAPTER 6 CONCLUSIONS AND FUTURE WORK 60
6.1 Future Work 60
REFERENCES 62

LIST OF FIGURES

Figure 2.1 An Example of C++ Code in srcML Format 11

Figure 3.1 Overview of the Stereotyping Process Using Stereocode 16

Figure 3.2 XPath Expression Example 18

Figure 3.3 Identifying State-Accessing Calls in Stereocode 20

Figure 3.4 Rules to Identify the Get Stereotype 21

Figure 3.5 Rules to Identify the Predicate Stereotype 23

Figure 3.6 Rules to Identify the Property Stereotype 24

Figure 3.7 Rules to Identify the Void-Accessor Stereotype 25

Figure 3.8 Rules to Identify the Set Stereotype 26

Figure 3.9 Rules to Identify the Command Stereotype 27

Figure 3.10 Rules to Identify the Factory Stereotype 28

Figure 3.11 Rules to Identify the Collaborator Stereotype. 29

Figure 3.12 Rules to Identify the Controller Stereotype. 30

Figure 3.13 Rules to Identify the Stateless Stereotype 31

Figure 3.14 Rules to Identify the Wrapper Stereotype 32

Figure 3.15 Rules to Identify the Empty Stereotype 33

Figure 3.16 Rules to Identify the Empty Stereotype for Classes 34

Figure 4.1 Stereotype Not Capturing Method Behavior 39

Figure 4.2 Incorrectly Stereotyped Methods 40

Figure 4.3 Visualizing Distribution of Method Stereotypes in HippoDraw (Data From Table 4.3) 42

Figure 4.4 Visualizing Distribution of Method Stereotypes in PowerShell (Data from Table 4.4) 45
Figure 4.5 Visualizing Distribution of Method Stereotypes in RxJava (Data from Table 4.5) 46
Figure 4.6 Visualizing Distribution of Class Stereotypes in PowerShell 47
Figure 5.1 Sample of the Distribution of Method Stereotypes for the Analyzed Repositories. Values are Percentages of Each Unique Method Stereotype 52
Figure 5.2 Sample of the Distribution of Class Stereotypes for the Analyzed Repositories. Values are Percentages of Each Unique Class Stereotype 53
Figure 5.3 Result of Hierarchical Clustering on Method Stereotypes using Ward 56
Figure 5.4 Result of Hierarchical Clustering on Class Stereotypes using Ward 56
Figure 5.5 Method from the wxWidget System Stereotyped as a Controller Stateless. ... 57

LIST OF TABLES

Table 2.1 Taxonomy of Method Stereotypes 7

Table 2.2 Taxonomy of Class Stereotypes 9

Table 4.1 Distribution of Primary Method Stereotypes in HippoDraw using Doleh's Stereocode [Doleh 2021] 37

Table 4.2 Distribution of Secondary Method Stereotypes in HippoDraw using Doleh's Stereocode [Doleh 2021] 37

Table 4.3 Distribution of Method Stereotypes in HippoDraw Using the New Stereocode 41

Table 4.4 Distribution of Method Stereotypes in PowerShell 43

Table 4.5 Distribution of Method Stereotypes in RxJava 44

Table 5.1 List of Selected Software Repositories Ordered by Language and Domain 50

Table 5.2 Stereotyped Method and Class Counts for the Analyzed Repositories 51

Table 5.3 Frequency of the Controller Stateless Stereotype for All Systems. 59

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my wonderful parents Faisal and Raidah Al-Ramadan, and the rest of my family, Ahmed, Deena, and Zaina, for their endless love and support throughout my life. Thank you for always being there for me. I would not be where I am today without all of you.

I would like to thank my advisor, Dr. Jonathan Maletic, for his guidance and feedback, and for being an incredible mentor. Thank you for believing in me and helping me grow as a researcher and as a person. I would also like to thank Dr. Natalia Dragan and Dr. Michael Decker for their support and encouragement. I would like to extend my thanks to my research colleague, Joshua Behler, for his great help and support.

To the members of my defense committee, Dr. Michael Collard, Dr. Lei Xu, and Dr. Mikhail Nesterenko thank you so much for taking the time to read my thesis and attend my defense.

This work was supported by the National Science Foundation under grant CCRI: ENS: Collaborative Research: Enabling Automated Language Support for the srcML Infrastructure, srcML.org J. Maletic (PI), M. Collard (UA) $652,485 Total, $397,848 KSU, $254,636 UA, CNS 20-16465/16452, 07/15/2020 - 06/30/2024.

Ali Al-Ramadan

May 2, 2023, Kent, Ohio

CHAPTER 1

Introduction

Software systems are becoming increasingly complex, and understanding their structure and design principles is important for general development and maintenance. Method and class stereotypes [Dragan et al. 2006; Dragan et al. 2009; Dragan et al. 2010] have emerged as powerful abstractions that can capture high-level design and behavior of software systems. These stereotypes can provide developers with a concise and meaningful way to quickly grasp the overall behavior of a software system.

Stereotype information has been shown to be useful in a wide variety of studies [M. J. Decker et al. 2018; Alhindawi et al. 2013; Dragan et al. 2011; Abid et al. 2015; Moreno et al. 2013; Gökmen et al. 2021] among others. Despite their importance, there is a lack of comprehensive tools for the automatic generation of stereotype information from source code. The current tools, including Stereocode [Dragan et al. 2006; Dragan et al. 2009; Dragan et al. 2010; Doleh 2021] and JStereoCode [Moreno and Marcus 2012], only provide basic static analysis functionalities, cannot stereotype whole software systems, are hard to use or install (i.e., research prototypes), and support a single programming language (e.g., C++ or Java). Furthermore, there is a need for empirical studies that investigate the distribution and relevance of stereotypes for software systems across different programming languages. Previous studies are limited to a single programming language (e.g., C++ or Java). As such, we do not know, for instance, how the distribution of stereotypes may vary for systems within the same domain and architecture (e.g., GUI Library) but developed in different languages.

To address these challenges, we introduce a new version of Stereocode that can reliably and accurately stereotype complete software systems written in C++, C#, or Java. Second, the thesis presents an empirical study that examines the distribution of method and class stereotypes for 33 open-source software systems from various domains across C++, C#, and Java. The findings of this study show that programming languages have a strong influence on design. In addition, the study uncovered possible emergent patterns for methods that reflect the characteristics of free functions.

1.1 Motivation

The motivation behind this work is driven by two main goals. First, there is a need for robust and language-agnostic tools for the automated generation of stereotype information for both methods and classes. Existing tools are mostly hacked versions that rely on basic analysis for stereotype computations.

Second, there is a lack of studies investigating the distribution of stereotypes for software systems written in different programming languages mainly due to not having any tools that support that.

1.2 Problem Statement

The goal of this thesis is to improve Stereocode to provide a general tool that is usable to support the automatic identification of method and class stereotypes for complete software systems written in C++, C#, or Java. In addition, we demonstrate the effectiveness and usefulness of the new Stereocode by conducting an empirical study that explores and analyzes the distribution of stereotypes in real-world software systems across C++, C#, and Java.

1.3 Contributions

This thesis presents two main contributions to the field of software engineering. First, it introduces a new version of Stereocode to support the stereotyping of software systems at a large scale. The new Stereocode can perform an in-depth, language-specific analysis, to generate both method and class stereotypes for entire software systems written in either C++, C#, or Java. Moreover, the tool can handle features that are unique to each language such as friend functions, multi-level inheritance, properties, partial classes, generics, structs, and interfaces.

The second contribution is an empirical study investigating method and class stereotype distributions across software systems written in C#, Java, and C++. The study explores whether systems from different domains cluster together based on their domain, or the programming language used. The findings from this study provide valuable insights into the influence of programming languages on software design. Furthermore, the study uncovered possible emergent patterns in software design that can assist developers in program comprehension and design recovery.

1.4 Organization of the Thesis

The work in this thesis is organized as follows: CHAPTER 2 provides background information on method and class stereotypes, their applications in software documentation and recovery, and previous work on the Stereocode tool. CHAPTER 3 discusses the architecture and implementation of the new Stereocode, detailing how it computes each stereotype. CHAPTER 4 evaluates the new version of Stereocode against previous versions as well as demonstrates its effectiveness in stereotyping real-world systems. CHAPTER 5 conducts an empirical study investigating stereotype distributions across C++, C#, and Java

for open-source systems and discusses the findings and their significance. CHAPTER 6 discusses future work and conclusions.

CHAPTER 2

Background and Related Work

This chapter presents background and related work concerning method and class stereotypes. It highlights various applications and studies utilizing stereotype information for software documentation and design recovery. Additionally, the chapter explores prior work on Stereocode and similar tools.

2.1 Method Stereotypes

Modern software development is fundamentally built upon a variety of standard abstractions that promote good Object-Oriented (OO) design. An abstraction is a simple structure that captures an underlying idea and simplifies it to its purest form. These abstractions act as guiding principles that assist developers in structuring their code effectively and efficiently. Among these abstractions are design patterns [Gamma et al. 1995], which provide reusable solutions to common OO design problems. Design patterns allow developers to define a common vocabulary and a naming scheme that helps reduce the overall design complexity to enhance software maintainability and upgradability,

Another form of OO design abstraction was introduced by the work of Dragan et al.[Dragan et al. 2006; Dragan et al. 2009]. The authors introduced a novel taxonomy for method stereotypes as shown in Table 2.1. Method stereotypes are a set of abstracted definitions that describe and reflect the intrinsic or atomic behavioral aspects and functionalities of a method in a class at a much lower level than typical design patterns. More formally, a method stereotype is a concise high-level description of the role of a method in a software system with respect to its context within a class. These nano patterns

of design can be leveraged as building blocks for more advanced forms of code analysis, and knowledge extraction from software systems. For instance, metrics that utilize method stereotypes have a more in-depth view of the software system with little cost compared to traditional metrics that rely on common features (e.g., number of lines of code).

Method stereotypes are separated into five distinct categories. A *Structural Accessor* method returns information about the state of an object on which it is called. A *Structural Mutator* method modifies the state of an object on which it is called. A *Creational* method is responsible for creating and/or destroying objects. A *Collaborational* method deals with objects that belong to other classes. A *Degenerate* method usually makes no use of the object state and often contains no statements.

Each category includes a set of specific stereotypes that indicate a finer-grained definition of a method's behavior. For example, stereotypes in the category of *Structural Mutators* modify an object's state, but the stereotype *set* specifically indicates a modification of a single data member of that object while the *command* stereotype performs a more complex change (e.g., modifying multiple data members). Methods can be stereotyped with one or more stereotypes. For example, a *predicate collaborator* is a method that returns a Boolean value calculated from the use of data members and uses at least one object of another class.

Stereotype Category	Stereotype	Description
Structural Accessors	get	Returns a data member
	predicate	Returns a Boolean value that is not a data member
	property	Returns information about data members (non-Boolean)
	void-accessor	Returns information about data members through method parameters
Structural Mutators	set	Modifies a data member
	command	Performs a complex change to the object's state
	non-void-command	
Creational	constructor	Creates and/or destroys objects
	copy-constructor	
	destructor	
	factory	
Collaborational	collaborator	Works with objects belonging to classes other than itself (parameter, local variable, or return value)
	controller	Changes only an external object's state (not this)
Degenerate	stateless	Does not read/change an object's state
	wrapper	Does not read/change an object's state. Has one call to external class methods
	empty	Has no statements

Table 2.1 Taxonomy of Method Stereotypes

2.2 Class Stereotypes

Dragan et al. [Dragan et al. 2010] also introduced a taxonomy for class stereotypes as shown in Table 2.2. A class stereotype is used to describe the role of a class in a software system. Class stereotypes are derived using the frequency distribution of method stereotypes and a set of predefined rules that map these distributions to the class stereotypes. For example, an *entity* class stereotype is identified when a class satisfies the following rules: has accessors other than *get* and mutators other than a *set*, has a 2:1 ratio of *collaborational* methods to *non-collaborations* methods, and has *controller* methods. Class stereotypes can be useful for a variety of software maintenance and evolution tasks. For example, class stereotype information can give clues on how different classes interact with each other in different design patterns. A class, as in the case of method stereotypes, can be labeled with one or more stereotypes.

Class Stereotype	Description
entity	Encapsulates data and behavior. Keeper of data model and/or business logic.
minimal-entity	Special case of Entity. Consists only of get, set, and command methods
data-provider	Encapsulates data and consists mainly of accessors
command	Encapsulates behavior and consists mainly of mutators
boundary	Communicator with a large percentage of collaborational methods and a low percentage of controller methods. It also does not have many factory methods
factory	Creator of objects and has mostly factory methods
control	Provides functionality to control external objects. Consists mostly of controller and factory methods
pure-control	A special case of control. Consists only of controller and factory methods
large-class	Contains a large number of methods that combine multiple roles, such as Data Provider, Commander, Controller, and Factory
lazy-class	Consists mostly of get, set, and degenerates methods. Occurrence of other methods is low
degenerate	Consists mostly of degenerate methods that do not read/write to the object's state
data-class	Consists only of get and set methods
small-class	Consists only of one or two methods
empty	Has no methods

Table 2.2 Taxonomy of Class Stereotypes

2.3 srcML Infrastructure

srcML is a powerful, robust, and highly scalable infrastructure used for transforming source code into a structured XML representation for multiple programming languages [Collard et al. 2013; Collard et al. 2011]. It includes a powerful parsing technology that allows for efficient transformation of source code to XML and vice versa by preserving the syntactic and semantic complexities of programming languages without any loss of information. The original source code, including the whitespaces, comments, and every statement is preserved in XML. srcML allows for the in-depth exploration, analysis, and manipulation of source code by leveraging the various XML technologies. Figure 2.1 shows an example of a simple C++ program along with its corresponding srcML format (XML representation of the source code). In the srcML format, each source file is encapsulated within a <unit> tag. This allows for multiple source files, or units, to be combined into one XML file, known as srcML archive.

The srcML format can be utilized to manipulate the source code in various ways including querying and transformation of certain parts of the source code using XPath and XSLT. For example, to extract the return type of all functions, the XPath *//src:function/src:type* is used in the srcML client. In addition, the infrastructure includes a C API called libsrcml that allows C++ developers to build tools that leverage the srcML technology. Moreover, Python bindings for libsrcml are available, which support all of libsrcml's functionalities.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<unit xmlns="http://www.srcML.org/srcML/src"
xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="1.0.0" language="C++">
<cpp:include>#<cpp:directive>include</cpp:directive>
<cpp:file>"rotate.h"</cpp:file></cpp:include>

<comment type="line">// rotate three values</comment>
<function><type><name>void</name></type>
<name>rotate</name><parameter_list>(<parameter><decl><type><name>int</name><m
odifier>&amp;</modifier></type> <name>n1</name></decl></parameter>,
            <parameter><decl><type><name>int</name><modifier>&amp;</modifier>
</type> <name>n2</name></decl></parameter>,
            <parameter><decl><type><name>int</name><modifier>&amp;</modifier>
</type> <name>n3</name></decl></parameter>)</parameter_list>
<block>{<block_content>

  <comment type="line">// copy original values</comment>
  <decl_stmt><decl><type><name>int</name></type> <name>tn1</name> <init>=
<expr><name>n1</name></expr></init></decl>;</decl_stmt>
  <decl_stmt><decl><type><name>int</name></type> <name>tn2</name> <init>=
<expr><name>n2</name></expr></init></decl>;</decl_stmt>
  <decl_stmt><decl><type><name>int</name></type> <name>tn3</name> <init>=
<expr><name>n3</name></expr></init></decl>;</decl_stmt>

  <comment type="line">// move</comment>
  <expr_stmt><expr><name>n1</name> <operator>=</operator>
<name>tn3</name></expr>;</expr_stmt>
  <expr_stmt><expr><name>n2</name> <operator>=</operator>
<name>tn1</name></expr>;</expr_stmt>
  <expr_stmt><expr><name>n3</name> <operator>=</operator>
<name>tn2</name></expr>;</expr_stmt>
</block_content></block>}</function>
</unit>
```

Figure 2.1 An Example of C++ Code in srcML Format

2.4 Stereocode Tool

Stereocode, a tool engineered to automatically identify method and class stereotypes using static analysis by leveraging the srcML infrastructure, was first introduced by Dragan et al. [Dragan et al. 2006; Dragan et al. 2009; Dragan et al. 2010] and was later enhanced by Doleh [Doleh 2021]. Laura Moreno and Andrian Marcus also introduced JStereoCode [Moreno and Marcus 2012], an Eclipse plug-in that supports the identification of method and class stereotypes for Java-based software systems. However, these earlier tools are not very usable and are difficult to install. In addition, they perform

very simple static analysis, do not account for all information in the class (e.g., inherited data), cannot stereotype whole software systems (i.e., can stereotype one source file at a time), only support stereotyping for a single programming language (e.g., C++ or Java), and rely on several assumptions for inheritance and calls.

This work in this thesis presents a new version of Stereocode that addresses the limitations found in previous stereotyping tools. The new tool can perform detailed analysis of source code to stereotype software systems in C++, Java, or C#. Stereocode is expected to contribute to the field of software engineering by providing more effective tools for static software analysis and design recovery.

2.5 Applications of Stereotypes

Dragan et al. [Dragan et al. 2009; Dragan et al. 2010] used method and class stereotype distributions to create signature descriptors as an indicator of the overall design of software systems. These stereotype distributions have been shown to be useful in many applications as they provide important insights into the design and functionality of the system. One application is demonstrated in the work of Decker et al. [M. J. Decker et al. 2018] where the authors conduct a study to show how methods' roles and behaviors change or evolve using method stereotypes. Other applications include improving feature location in source code using Latent Semantic Indexing [Alhindawi et al. 2013], introducing a UML layout scheme based on class stereotypes [Andriyevska et al. 2005], generating language summaries for methods in C++ [Abid et al. 2015; Moreno et al. 2013], characterizing commits on version control systems into different types [Dragan et al. 2011], generating commit messages [Linares-Vásquez et al. 2015; Shen et al. 2016], categorizing source-code identifiers [Newman et al. 2017], introducing TeStereo for tagging unit test methods

in Java [Li et al. 2018], assessing design and implementation of OO methods [Andras et al. 2013], using stereotypes as design patterns for serious games to enhance object-oriented software comprehension [Gökmen et al. 2021], and employing an automated machine learning-based approach for classifying the role-stereotype of classes in Java [Nurwidyantoro et al. 2019].

However, there has been no work examining stereotype distributions across different programming languages. Previous studies are limited to a single programming language leaving a gap in our understanding of how these stereotype distributions might vary across different programming languages. The work in this thesis aims to fill this gap by conducting an empirical study that investigates method and class stereotype distributions across software systems written in C++, C#, and Java. The findings from this study provide a base understanding of stereotype distributions across languages that can lead to improved software development tools and practices.

CHAPTER 3

Stereocode: A Multi-Language Tool for Automated Stereotype Detection

This chapter provides a detailed discussion on the development and architecture of the new Stereocode. We will examine its design, the process it employs to collect program information from source code using static analysis, the identification of method and class stereotypes, and the generation of the stereotyped output. In addition, we will evaluate the tool and demonstrate its usefulness in stereotyping full software systems in C++, C#, and Java.

3.1 Overview

Method and class stereotypes are a way of categorizing methods and classes based on their roles and responsibilities in a software system. This information is often not documented in source code. While it might seem easy to document this information manually, it can become quite costly to do so, and maintain, in large software systems. To address this issue, Stereocode was developed. Stereocode makes the process of stereotyping software systems more efficient and manageable by automatically identifying and documenting software systems with stereotype information using static analysis.

The version of Stereocode discussed in this thesis represents a significant advancement over the previous versions. It aims to provide a more complete, accurate, and usable tool that supports the stereotyping of multiple programming languages. Specifically, the incorporated features and enhancements offer several advantages including:

- The ability to accept a srcML input archive that consists of not just a single class, but an entire software system.

- Support for C# and Java programming languages.
- Handling of language-specific features such as friend functions (C++), properties (C#), partial classes (C#), structs (C++ and C#), interfaces (C# and Java), multi-level inheritance, and generics, among others.
- Providing multiple output formats for the stereotyped data with different views. Each view format offers a different perspective on the system's frequency distributions of stereotypes.
- Ability to easily adjust the method and class stereotype rules (e.g., add new rules or remove existing ones).

The process of stereotyping a software system using Stereocode, as shown in Figure 3.1, involves four main steps:

1. Conversion of source code input to srcML format.
2. Collection of program information (e.g., classes, data members, methods, etc.).
3. Computation of method and class stereotypes using static analysis and a set of predefined rules.
4. Output generation by re-documenting the srcML input archive with the option to generate other output views.

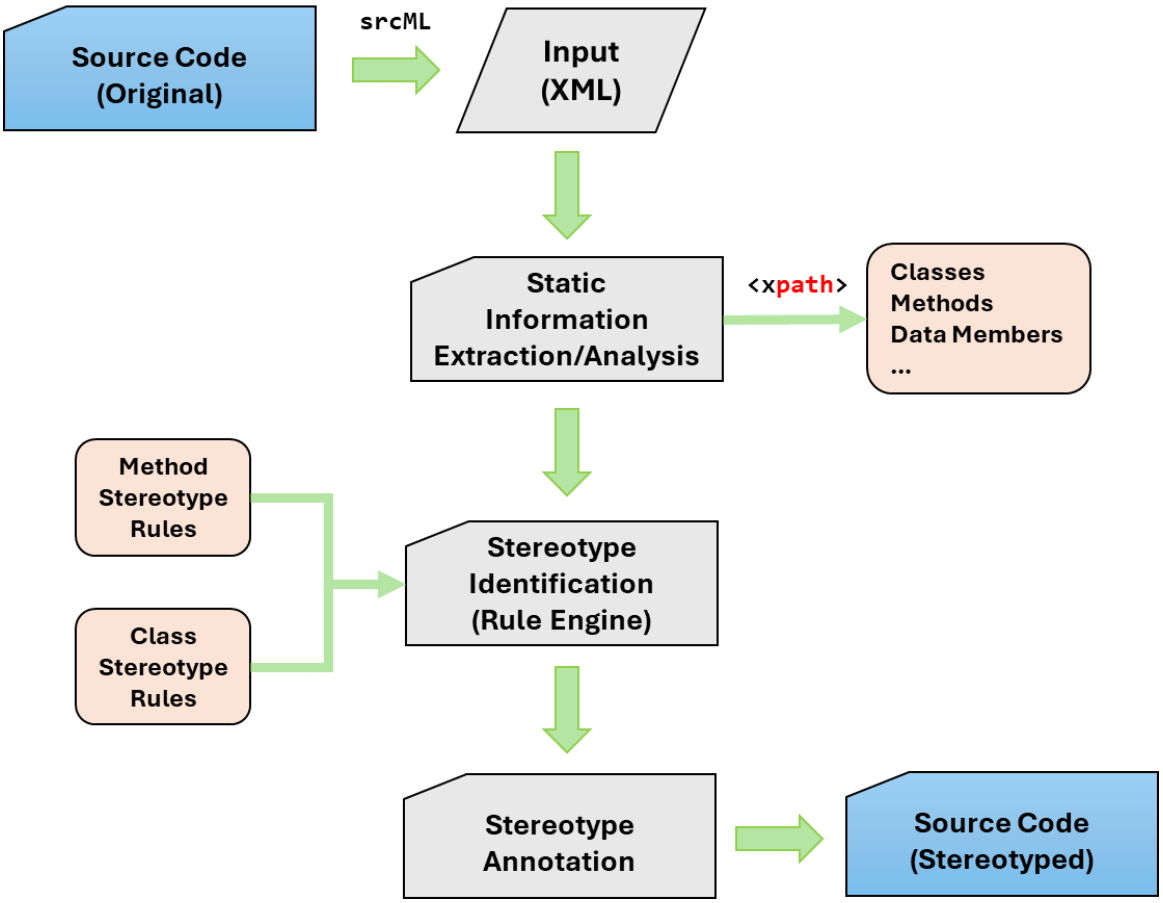


Figure 3.1 Overview of the Stereotyping Process Using Stereocode

3.2 Converting Source Code Input to srcML

The stereotyping process starts with the conversion of source code input to the srcML format. This format preserves all the information from the source code including the order of statements, the nesting of structures, and the original source code. This conversion is important as it transforms the source code into a format that Stereocode can process and analyze to extract the needed data for the computation of method and class stereotypes.

3.3 Collecting Program Information

The second step in the stereotyping process is the collection of program information related to various code elements. The process begins by scanning the units (i.e., source files) in the input archive to collect all the classes. For each class, information including the class name, parent class names, data members, specifiers, and methods are collected and stored for further analysis. For every method, information including the method name, return type, parameters, local variables, return expressions, and calls (including method, function, and constructor calls) are also collected. A function call is a call that does not usually include the *dot*, *arrow*, or *new* operators (e.g., foo() or className::foo()) where *foo* is a call to a method in the current class, a call to a free function, or a call to a static method. A method call is a call that usually includes the *dot* or *arrow* operators (e.g., var.bar() or var->bar()) where *var* is a variable (e.g., data member, local, global, or a parameter) and *bar* is a method defined inside *var*. A constructor call is a call that usually follows the *new* operator.

This information is then statically analyzed by Stereocode to derive the necessary data for the computation of stereotypes. This includes determining whether the method uses, changes, or returns data members, local variables, or parameters. In addition, Stereocode checks if the method is empty, is constant (C++ only), and whether it modifies parameters passed by reference. Stereocode utilizes libsrcml and various XPath expressions to collect the necessary information from the input archive. Figure 3.2 shows an example of an XPath expression that collects the data types of all parameters for all functions that are not nested in another function (C# allows nesting of functions) given that the datatype is followed by a name. C++ allows parameters with only a datatype to exist

for backward compatibility, and in this case, the XPath expression ignores them since they are not important for the computation of stereotypes.

```
//src:function[not(ancestor::src:function)]

/src:parameter_list/src:parameter/src:decl

/src:type[following-sibling::*[1][self::src:name]]
```

Figure 3.2 XPath Expression Example

Stereocode can automatically detect the language of each unit in the srcML archive and adjust its information collection process accordingly to suit that specific programming language. For example, if the unit's language is C#, then Stereocode will treat *properties* that are *auto-properties* as data members for the class since they are used to declare data members implicitly where the property name is the name of the data member. Additionally, Stereocode maintains a list of language-specific primitives (e.g., int, double, etc.) that is used to distinguish between a primitive and a non-primitive datatype. Users can also provide their list of user-defined primitives if necessary. Moreover, Stereocode also has the option to collect structs (for C# and C++) and interfaces (for C# and Java) along with their structural information for stereotyping if necessary.

One of the enhancements introduced in this version of Stereocode is the ability to track state-accessing calls. For each method call, Stereocode checks to see if the call is made on a data member. If it is true, the call is kept. Method calls that are not called on data members are removed since they do not use or change the state of the object (i.e., they are *stateless* calls). For each function call, Stereocode initially checks to see if the function call is a call to a method in the class, or a call to a free function. If the call is to a method

in the class, then the call is kept, else it is removed. Additionally, Stereocode keeps a list of ignorable calls for each language. Calls included in this list are assumed to not use or mutate the state of the object. By default, Stereocode includes a predefined list of such calls for each language (e.g., *assert()* for C++, or *printLn()* for Java, or *WriteLine()* for C#, etc.). Users can also provide their list of user-defined calls to ignore if needed. Figure 3.3 shows the summarized process that Stereocode uses to identify calls that utilize the state of the object.

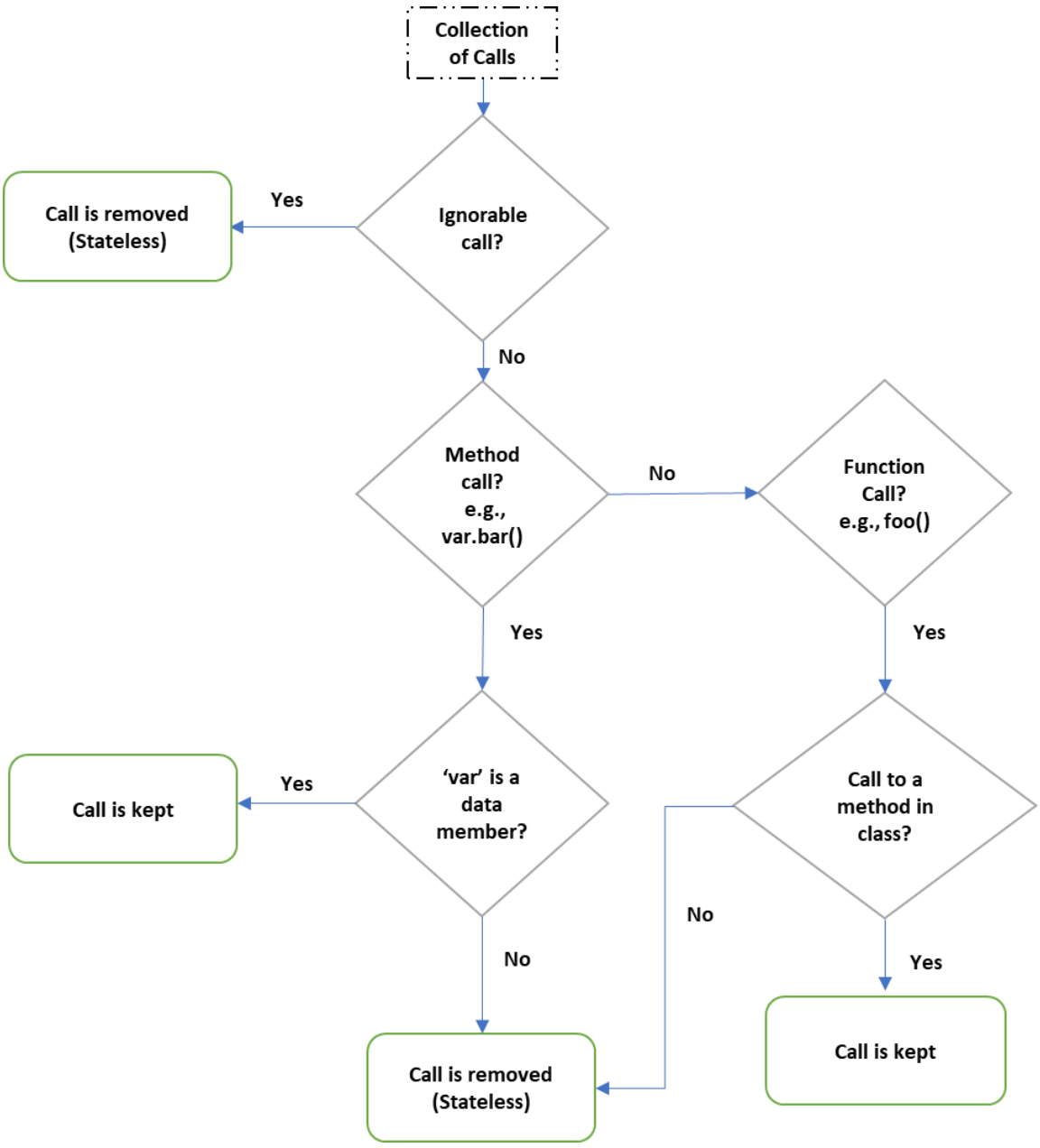


Figure 3.3 Identifying State-Accessing Calls in Stereocode

Another enhancement is the ability to find the inherited data members and methods for all levels of inheritance. For every class collected, Stereocode will visit every parent in the hierarchy chain and collect the non-private data members and methods. Knowing this information is important for accurate computation of stereotypes. Doleh's [Doleh 2021]

version of Stereocode lacks support for inheritance. Instead, assumptions are made based on a basic set of rules. For instance, if a method returns a global variable, then it is assumed to return an inherited data member since the rules simply check if this variable is a local, parameter, a data member, an operator, or some keyword such as *false* or *true*. Currently, no such assumptions are made since each class stores all of the inherited information needed for the computation of stereotypes.

3.4 Computing Method Stereotypes

The third step is the computation of method stereotypes. Stereocode uses static analysis to identify the stereotypes of all methods based on the information collected in the previous step. However, it is important to note that some of these stereotypes have been adjusted and generalized to accommodate the new programming languages (i.e., Java and C#). The following sub-sections will provide an in-depth explanation of how each method stereotype is computed and assigned by Stereocode.

3.4.1 Stereotype: Get

The *get* stereotype is assigned to a method that returns the value of some data member. To label a method with the *get* stereotype, it needs to satisfy the following rules:

```
1] Return type is not void
2] Contains at least one simple return expression that
returns a single data member (e.g., return dm;) where 'dm'
is a data member
```

Figure 3.4 Rules to Identify the Get Stereotype

To determine if a method returns a data member, Stereocode collects the return expressions for each method. Initially, each return expression is filtered to remove certain operators and patterns that may hinder the detection of variables including:

- Brackets [] for arrays (e.g., return var[index])
- Parentheses () and curly braces {} (C++ only) that surround the return expression (e.g., return (var) or return {var})
- Pointers * for dereferencing in C++ or C# (e.g., return *var)

Where *var* is a variable (e.g., data member, global, parameter, or local). This filtering step is necessary to isolate the core components of the return expression to help detect the presence of a variable. Each return expression is then checked to see if it is a *simple* or a *complex* return expression using regex expressions. A *simple* return expression only returns a single variable or the value of a variable (e.g., *return (var)* or *return *var* or *return var[index]*). A complex return expression, on the other hand, may or may not use a variable in its expression (e.g., *return 5 + 5* or *return true* or *return var * factorial(var – 1)*). *Complex* return expressions are ignored for the *get* stereotype and only *simple* return expressions are considered. Each *simple* return expression is then checked against local variables and parameters first before being checked for data members since these could overshadow a data member. Any method that satisfies the above rules is labeled with the *get* stereotype.

3.4.2 Stereotype: Predicate

The *predicate* stereotype is assigned to a method that computes a Boolean value using data member values and returns it. To label a method with the *predicate* stereotype, it needs to satisfy the following rules:

```
1] Return type is Boolean

2] Uses at least one complex return expression

3] Uses a data member in:

  -  An expression or

  -  has at least one call to other accessor methods in
     class
```

Figure 3.5 Rules to Identify the Predicate Stereotype

Method return types are collected in the pre-processing step and checked if they are of a Boolean type. Complex return expressions are also collected as described in the previous sub-section. Additionally, every expression in the method is collected and checked for use of data members. Calls to accessor methods in the class are also collected. Any method that satisfies the above rules is labeled with the *predicate* stereotype.

3.4.3 Stereotype: Property

The *property* stereotype is assigned to a method that computes a non-Boolean value using data member values and returns it. To label a method with the *property* stereotype, it needs to satisfy the following rules:

```
1] Return type is not void or Boolean

2] Uses at least one complex return expression

3] Uses a data member in:

  -  An expression or

  -  has at least one call to other accessor methods in
     class
```

Figure 3.6 Rules to Identify the Property Stereotype

The *property* stereotype is similar to the *predicate* stereotype and requires almost the same information about the method. The return type cannot be a *void* or a *Boolean*. Any method that satisfies the above rules is labeled with the *property* stereotype.

3.4.4 Stereotype: Void-Accessor

The *void-accessor* stereotype is assigned to a method that returns information about the state of the object through a parameter. To label a method with the *void-accessor* stereotype, it needs to satisfy the following rules:

```
1] Return type is void
2] Uses at least one parameter that is:
   -  Passed by non-const reference and
   -  is assigned a value
2] Uses a data member in:
   -  An expression or
   -  has at least one call to other accessor methods in
      class
```

Figure 3.7 Rules to Identify the Void-Accessor Stereotype

The *void-accessor* stereotype is the last of the accessors. This stereotype indicates that a method is returning information about the state of the object via its parameters. Stereocode collects these parameters and verifies if each one is passed by a *non-const reference*. A *non-const reference* parameter implies that any changes to this parameter will affect the original object. In C++, parameters of array type (e.g., param[]), pointer type (e.g., *param), or use the reference operator (e.g., &param) and are not constant, are considered, where *param* is a method parameter. In C#, parameters that use the *ref* or *out* keywords, are of array types, or are non-primitive (i.e., not of simple data types like *int* or *double*) are considered. For Java, parameters that are of array types, or are non-primitive

are considered. A non-primitive parameter in C# (assuming no usage of *ref* or *out*) or Java is automatically passed by value that is a reference to the original object. Therefore, any changes to the state of this non-primitive parameter will be reflected on the original object (e.g., param.dm = value). Any method that satisfies the above rules is labeled with the *void-accessor* stereotype.

3.4.5 Stereotype: Set

The *set* stereotype is the first of the mutators and is assigned to a method that modifies the value of a single data member. To label a method with the *set* stereotype, it needs to satisfy the following rules:

```
1] Only one data member is changed

2] Number of calls on data members or to other methods in
class is at most 1
```

Figure 3.8 Rules to Identify the Set Stereotype

For each method, Stereocode collects all expressions that contain a data member subject to modification (e.g., utilizing the assignment operator). Stereocode uses a comprehensive list of mutating operators (e.g., =, += , /=, --, ++, etc.) to check if an expression modifies a data member. Stereocode then counts how many times each data member is changed in all the expressions. The total number of modified data members is then stored in each method. If a data member is changed multiple times, then it is only counted as one change. Following this, Stereocode looks to see if the method uses any calls. As described previously, Stereocode collects and filters all calls. Any method found

to satisfy the above rules is labeled with the *set* stereotype. For C++, constant methods are considered since these methods can still modify data members that are declared as *mutable*.

3.4.6 Stereotype: Command

The *command* stereotype is assigned to a method that performs a complex change to the state of the object. To label a method with the *command* stereotype, it needs to satisfy one of following cases:

```
Case 1: More than one data member is modified
Case 2: one data member is modifed and:
  -  there is at least two calls to either data members or
     other methods in class
Case 3: No data members are modifed and:
  -  there is at least one call on a data member
     or one call to other methods in class
```

Figure 3.9 Rules to Identify the Command Stereotype

For each method, the total number of data members that are modified and the usage of calls are collected and stored as detailed in the previous subsection. Any method that satisfies any of the cases is labeled with the *command* stereotype. Methods that do not return any value (i.e., have *void* as return type) and satisfy the rules of the *command* stereotype are labeled with *non-void-command* stereotype. For C++, constant methods that satisfy *Case 1* are considered since these data members can be mutable.

3.4.7 Stereotype: Factory

The *factory* stereotype is the first stereotype in the creational category and is assigned to a method that creates an object and returns it to the client. To label a method with a *factory* stereotype, it needs to satisfy the following rules:

```
1] Return type is non-primitive

2] At least one return expression that returns:

   -  A local variable or a parameter or a data member or

   -  a constructor call (e.g., using new)
```

Figure 3.10 Rules to Identify the Factory Stereotype

Method return types are collected and parsed by removing the containers (e.g., vector, list, set, etc.) and specifiers (e.g., const, volatile, public, etc.). The simplified return type is then compared against a list of primitive types. If it is not found in the list, then it is considered non-primitive. The method must also contain at least one return expression that returns a local variable, parameter, or data member that is initialized with an object using a constructor call (e.g., using *new* operator) somewhere in the method or the return expression itself contains a constructor call. Any method that satisfies the above rules is labeled with the *factory* stereotype. The *factory* stereotype is the only stereotype we consider from the creational category as the other stereotypes (i.e., constructor, copy-constructor, and destructor) are well-known in the software community.

3.4.8 Stereotype: Collaborator

The *collaborator* stereotype is the first stereotype in the collaborational category and is assigned to a method that deals with objects of other classes. To label a method with a *collaborator* stereotype, it needs to satisfy the following rules:

```
1] Uses at least one non-primitive data type (not of the
same class) that is:
   -  A local variable or
   -  A Parameter or
   -  A return type or
   -  A data member
```

Figure 3.11 Rules to Identify the Collaborator Stereotype.

Return types for the method, local variables, parameters, and data members are collected and parsed by removing the containers and specifiers as discussed in the previous subsection. Classes cannot collaborate with themselves. Therefore, data types that are of the same class are ignored. Any method that satisfies the above rules is labeled with the *collaborator* stereotype.

3.4.9 Stereotype: Controller

The *controller* stereotype is assigned to a method that only deals with external objects. To label a method with a *controller* stereotype, it needs to satisfy the following rules:

```
1] No data members are modified directly in the method

2] No calls to methods in class

3] Has at least one external call
```

Figure 3.12 Rules to Identify the Controller Stereotype.

A controller stereotype indicates that a method is collaborating with external objects without directly mutating the state of the current object. Any method that satisfies the above rules is labeled with the *controller* stereotype.

3.4.10 Stereotype: Stateless

The *stateless* stereotype is the first stereotype in the degenerate category and is assigned to a method that does not read or change an object's state directly. To label a method with a *stateless* stereotype, it needs to satisfy the following rules:

```
1] Method contains at least one non-comment statement
   (i.e., method is not empty)
2] No data members are used or modified directly in the
   method
3] No calls to methods in class
4] Has at least two external calls
```

Figure 3.13 Rules to Identify the Stateless Stereotype

Methods labeled with stereotypes from the degenerate category are usually an indication of code smells or bad design. Alternatively, they could be candidates for overriding, exception handlers, or utilities that perform an external service. Stereocode collects the usage of data members and calls as described in the previous sub-sections. A simple XPath is then utilized to determine if a method is empty. Any method that is not empty, does not read or modify data members, makes no calls to other methods in the class, and has at least two calls to external classes are labeled with the *stateless* stereotype.

3.4.11 Stereotype: Wrapper

The *wrapper* is assigned to a method that wraps an external call. To label a method with a *wrapper* stereotype, it needs to satisfy the following rules:

```
1] Method contains at least one non-comment statement
   (i.e., method is not empty)
2] No data members are used or modified directly in the
   method
3] No calls to methods in class
4] Has one external call
```

Figure 3.14 Rules to Identify the Wrapper Stereotype

A *wrapper* method is a method that "wraps" an external call to a method of an external class or to a free function and can perform extra tasks before or after the call. These tasks could include logging, resource management, error handling, etc. A wrapper does not read or change an object's state directly. Any method that satisfies these rules is labeled the *wrapper* stereotype.

3.4.12 Stereotype: Empty

The *empty* stereotype is the simplest of stereotypes. To label a method with an *empty* stereotype, it needs to satisfy the following rule:

```
1] Method has no statements:

   - Except for comments
```

Figure 3.15 Rules to Identify the Empty Stereotype

An *empty* method often signifies that the method has yet to be implemented and exists for the purpose of being overridden. Such methods are usually not indicative of poor design. Stereocode uses a simple XPath expression to identify the first non-comment statement in a method. If no statements are found, then the method is classified as *empty*.

3.5 Computing Class Stereotypes

The next step involves the computation of class stereotypes. This is achieved by leveraging the information gathered from the method stereotypes computed in the previous step. The process is relatively straightforward and involves mapping this information into a set of predefined rules for each class stereotype. For instance, a *degenerate* class stereotype is identified when more than half of the methods in a class are degenerate (e.g., empty, stateless, or wrapper). The outcome is a set of stereotypes assigned to each class in the software system.

It is worth mentioning that the rules for class stereotypes, shown in Table 2.2, and outlined in Dragan et al. [Dragan et al. 2010], remain unchanged in this work and are directly applied from the original study. Therefore, a detailed discussion of these rules is

not included. However, the exception to this is the *empty* stereotype, which is newly added to the taxonomy to indicate an empty, or an abstract, class with no methods. This is different from the *small-class* stereotype, which dictates that a class needs to have at least one method and no more than three. To label a class with the empty stereotype, it needs to satisfy the following rule:

```
1] Class has no methods:

   -  Only method declarations are allowed

   -  No method definitions are allowed
```

Figure 3.16 Rules to Identify the Empty Stereotype for Classes

Any class that satisfies the above rules is labeled with the empty stereotype. In addition, any method or class that does not fit into any category and is left without a stereotype is labeled as *unclassified*. Furthermore, Stereocode allows the users to specify a threshold for the average number of methods per class. By default, this value is set to 21, and is needed for the computation of the *large-class* stereotype.

3.6 Outputting the Stereotypes

The final step in the stereotyping process is the output generation. Stereocode outputs the result by re-documenting the srcML input with the computed stereotypes. This can be in the form of XML attributes added to the function tags in the srcML representation of the source code (e.g., <function stereotype = “get”> or <class stereotype = “entity”>). Moreover, Stereocode can generate optional report files that provide different views of the

frequency distribution of the stereotypes in the system. These report files are stored as CSV files and provide four different views including:

- **Method View:** This report provides a count of each method stereotype (e.g., get, predicate, property, etc.)
- **Class View:** This report provides a count of each class stereotype (e.g., entity, minimal-entity, data-provider, etc.)
- **Unique Class View:** This report provides a count of each unique class stereotype assigned to each class (e.g., pure-control small-class, etc.)
- **Unique Method View:** This report provides a count of each unique method stereotype assigned to each method (e.g., command collaborator, get collaborator, etc.)

In the *Unique Class View* and *Unique Method View*, each count corresponds to a unique stereotype assigned to a class or a method as a whole. For example, a method labeled with a "get predicate" is counted as one unique stereotype. This is different from the *Method View*, where each individual stereotype (e.g., "get", "predicate") is counted separately. In addition to these views, Stereocode can generate an optional TXT file, or a CSV file, that shows the complete list of stereotypes for all classes and methods. These outputs serve as valuable resources for understanding, analyzing, and documenting the software system's design and behavior.

CHAPTER 4

Stereocode Evaluation

In this chapter, we compare the new version of Stereocode to its predecessor introduced by Doleh [Doleh 2021]. Doleh's work used HippoDraw v1.21.3, an open-source data analysis environment, as the baseline system for evaluating Stereocode. Therefore, we will utilize the same system in this comparison. In addition, we will stereotype other open-source systems written in C# and Java to better demonstrate the enhancements in this version of Stereocode.

Table 4.1 and Table 4.2 present the method stereotype distribution of the HippoDraw system as generated by Doleh's Stereocode. In their approach, every method is assigned one primary stereotype from the Accessor, Mutator, or Creational categories along with zero or more secondary stereotypes from the Collaborational or Degenerate categories. A secondary stereotype can also be the only primary stereotype of the method. For instance, 393 methods are labeled with *property* as the primary stereotype, 78 methods are labeled as *collaborational-predicate* as the primary stereotype to indicate that these methods use objects of other classes to compute a Boolean value, 27 methods are labeled with the *stateless* secondary stereotype as primary, which could indicate a possible code smell, and 31 methods are labeled with *command* as a secondary stereotype, implying they make a complex change to the state of the object despite being declared as constant methods. The total number of methods stereotyped in the system is 2539.

Primary Method Stereotype	Method count
Get	198
Non-const-get	44
Predicate	64
Collaborational-predicate	59
Property	393
Collaborational-property	78
Voidaccessor	61
Collaborational-voidaccessor	7
Set	106
Command	929
Non-void-command	184
Collaborational-command	159
Controller	22
Factory	148
Empty	60
Stateless	27
Total Methods	**2539**

Table 4.1 Distribution of Primary Method Stereotypes in HippoDraw using Doleh's Stereocode [Doleh 2021]

Secondary Method Stereotype	Method Count
Command	31
Stateless	132
Collaborator	1667
Wrapper	156

Table 4.2 Distribution of Secondary Method Stereotypes in HippoDraw using Doleh's Stereocode [Doleh 2021]

One notable difference between the two versions of Stereocode is the way in which stereotypes are computed. Doleh's Stereocode computes stereotypes in a specific order due to inflexibility in how the rules are implemented. For instance, a method is tested if it is a *get* before being tested if it is a *predicate*, and if it satisfies both, then the method is labeled

as a *get* only. The inflexibility of the rules limits the tool's ability to accurately detect the true behavior of each method with respect to its class. In contrast, the new version of Stereocode employs a more flexible and modular approach where the rules for each stereotype are disjointed and computed separately such that each method is stereotyped with every stereotype that it satisfies regardless of the order in which stereotypes are computed. A method that satisfies the rules of a *get* and a *predicate* will be labeled as both a *get* and a *predicate* by the new Stereocode. While counterintuitive, this explicit labeling gives a better indication of the behavior of the method (e.g., code smell). Figure 4.1 shows an example of a method labeled as *get collaborator* by Doleh's Stereocode. The method is indeed a *get* and a *collaborator* as it returns the *m_title* data member and interacts with another data member, m*_projector*, that is of a different type than the class itself. However, this method is also a *property* due to the first return statement returning a property from the data member m_projector. The enhanced Stereocode labels this method as a *get property collaborator* to capture the full behavior of this method. And in this case, it is a possible code smell.

```
// @stereotype get collaborator
const std::string &
DataRep::
getTitle ( ) const
{
  if ( m_title == "%t" ) {
    return m_projector -> getTitle ();
  }
  //  else
  return m_title;
}
```

Figure 4.1 Stereotype Not Capturing Method Behavior

Figure 4.2 shows an example of two methods that are incorrectly stereotyped by Doleh's Stereocode. The first method (i.e., getConstWid()) returns an inherited data member *m_width* that is of a primitive type. However, Doleh's Stereocode does not track inherited members. As a result, it is incorrectly labeled as a *collaborational-property stateless* because the tool assumes that the returned value is not a data member, the method has no function calls and uses no data members. The second method (i.e., isLog()) is also incorrectly labeled as a *collaborational-predicate stateless* for similar reasoning. On the other hand, the new Stereocode stereotypes these methods correctly as *get* and *stateless* respectively. The *getConstWid()* method is labeled with a *get* since it simply returns an inherited data member of a primitive type (i.e., double), and isLog() is labeled as *stateless* since it makes no use of any data members and has no function calls.

```
// @stereotype collaborational-property stateless
double BinnerAxisLog::getConstWid ( ) const
{
  return m_width;
}

// @stereotype collaborational-predicate stateless
bool AxisModelLog::isLog() const
{
  return true;
}
```

Figure 4.2 Incorrectly Stereotyped Methods

Another difference between the two versions of Stereocode is the ability to stereotype complete software systems. Doleh's Stereocode is not capable of stereotyping complete software systems and is limited to stereotyping individual classes. This limits the useability of the tool and the depth of analysis that it can perform as certain features require the analysis of the full system (e.g., finding inherited data members and methods). As a result, the data presented in Table 4.1 and Table 4.2 was generated by stereotyping each individual class of the HippoDraw system separately and then aggregating the results together, which takes considerable time and effort. With the new Stereocode, the stereotyping process for the entire HippoDraw took less than 15 seconds. Table 4.3 shows the method distribution of the HippoDraw system as generated by the new Stereocode.

Method Stereotype	Stereotype Count
get	355
predicate	152
property	733
void-accessor	292
set	220
command	1035
non-void-command	309
collaborator	1865
controller	545
factory	214
empty	91
stateless	462
wrapper	203
unclassified	17
Total Stereotypes	**6493**
Total Methods	**3343**

Table 4.3 Distribution of Method Stereotypes in HippoDraw Using the New Stereocode

The total number of stereotyped methods found in the HippoDraw system by the enhanced Stereocode is 3343 as shown in Table 4.3. This is 804 more stereotyped methods compared to those shown in Table 4.1. The increase in the number of stereotyped methods is due to the new Stereocode's support for structs and the ability to accurately collect all methods. The total number of method stereotypes is 6493 and it represents the total number of all the individual stereotypes for all methods. This number is justified since each method can be labeled with multiple stereotypes. Figure 4.3 shows a pie chart to visualize the HippoDraw system as seen by the new Stereocode.

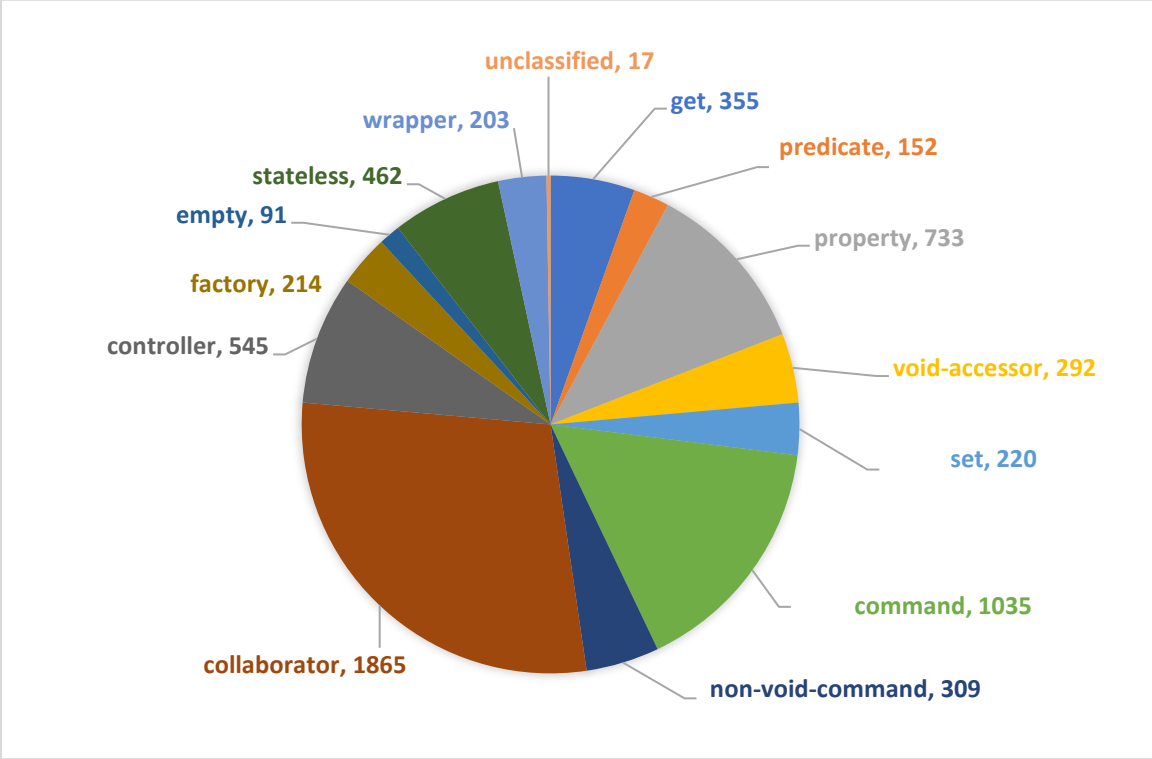


Figure 4.3 Visualizing Distribution of Method Stereotypes in HippoDraw (Data From Table 4.3)

It is important to note that the definition of a *controller* stereotype has been refined in the new Stereocode. Previously, a method was labeled as a *controller* if it satisfied the rules of a *collaborator* and had no other primary stereotypes. In the new Stereocode, a *controller* is a method that does not mutate the state of the current object and deals only with external objects, but it can access data members from the current class. The *stateless* stereotype has also been adjusted to indicate that a method can have more than two calls as long as they are *stateless calls*, meaning they are not calls to methods in the class, and the method is not using data members. Previously, the *stateless* stereotype was given to a method that has no calls of any type, including *stateless* calls, and does not use any data members. This led to many methods not being classified as *stateless* when they are

stateless. Other stereotypes have been slightly adjusted, as discussed in CHAPTER 3, to accommodate the new languages (e.g., C# and Java). The overall distribution of method stereotypes shown in Figure 4.3 is a good indication of the design of the HippoDraw system.

We will now Stereotype two open-source systems: PowerShell v7.4.1 and RxJava v3.1.8 to show the support for C# and Java in the new Stereocode. Both systems are available on GitHub. We start by converting both systems to the srcML format, and then using Stereocode to find the Stereotypes. It is worth noting that structs and interfaces are considered. Moreover, Stereocode does not consider static classes or static methods, In addition, the repositories are stereotyped as is with no cleaning or preprocessing. The process took less than two minutes for each system. Table 4.4 and Table 4.5 show the method distribution for PowerShell and RxJava respectively.

Method Stereotype	**Stereotype Count**
get	2085
predicate	542
property	4230
void-accessor	94
set	2190
command	2809
non-void-command	4253
collaborator	12638
controller	3311
factory	1172
empty	134
stateless	2091
wrapper	1032
unclassified	1
Total Stereotypes	**36582**
Total Methods	**21076**

Table 4.4 Distribution of Method Stereotypes in PowerShell

Method Stereotype	Stereotype Count
get	93
predicate	200
property	879
void-accessor	29
set	162
command	3274
non-void-command	1006
collaborator	3945
controller	10975
factory	134
empty	75
stateless	9792
wrapper	464
unclassified	0
Total Stereotypes	**31028**
Total Methods	**16642**

Table 4.5 Distribution of Method Stereotypes in RxJava

We can clearly see a difference in the distribution of method stereotypes for both systems. The PowerShell system has high percentages for many Stereotypes. This suggests that PowerShell heavily relies on methods that interact with other objects, perform actions, and manage the flow of control within the system. This could reflect the nature of the system as a command-line shell and a scripting language where methods usually collaborate with external components to execute specific tasks.

The RxJava system on the other hand has a large concentration of *stateless* methods. The large number of *stateless* stereotypes in the new Stereocode can be attributed to the tool's ability to accurately identify methods that do not change the state of the object. In the general sense, many of these *stateless* methods indicate the existence of many utility methods that carry out general tasks (e.g., unit testing or providing an external service)

without altering the state of the object. Figure 4.4 and Figure 4.5 visualize the distribution of method stereotypes for PowerShell and RxJava respectively.

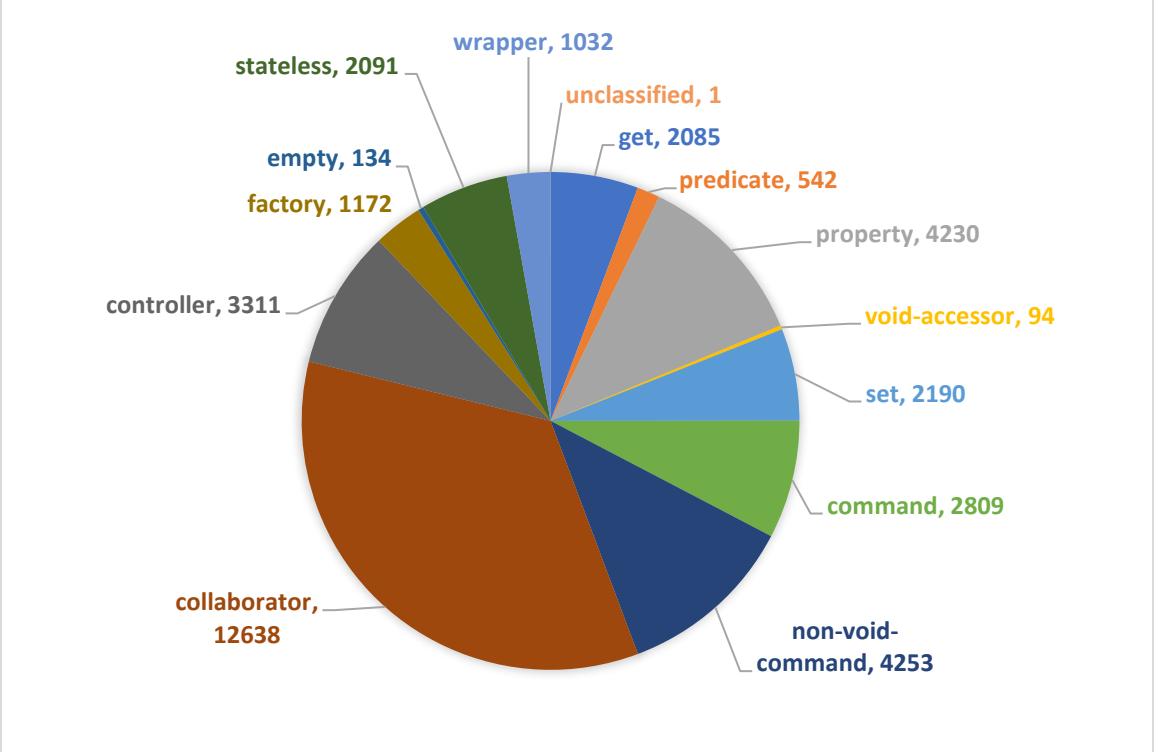


Figure 4.4 Visualizing Distribution of Method Stereotypes in PowerShell (Data from Table 4.4)

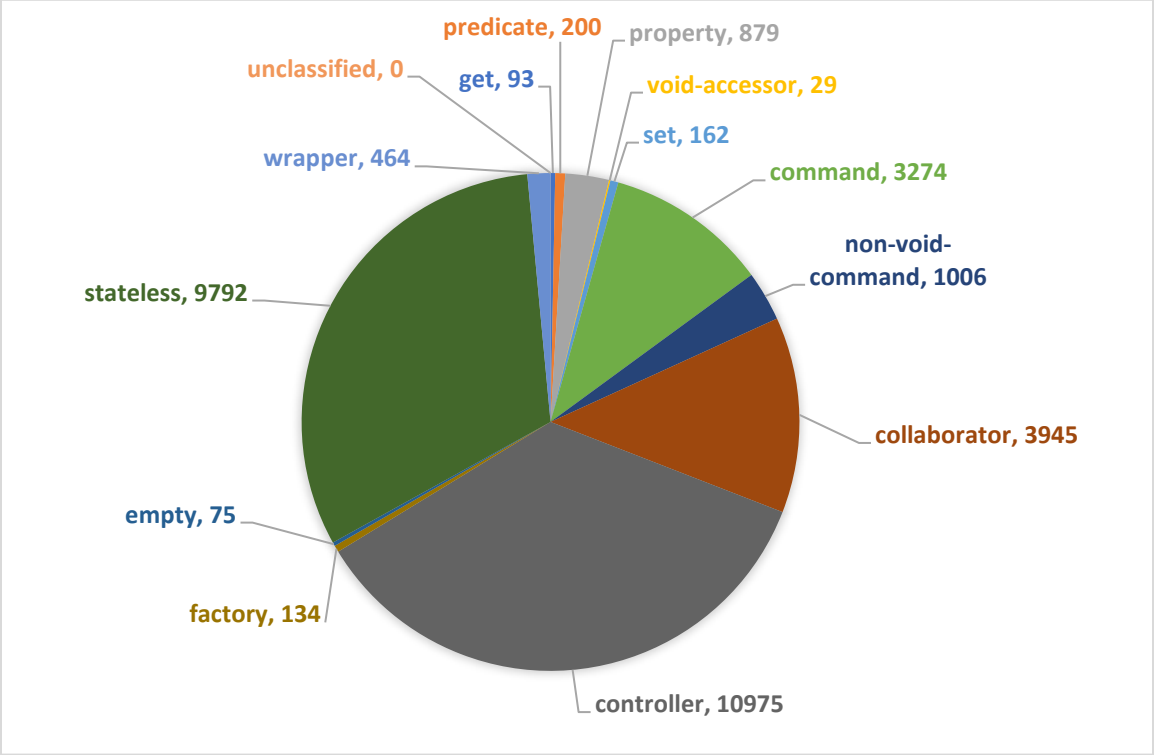


Figure 4.5 Visualizing Distribution of Method Stereotypes in RxJava (Data from Table 4.5)

The new Stereocode can also generate views at a higher level, that is, at the class level. These views can reveal how classes collaborate and interact with each other. A class stereotype can be an indicator of a potential design pattern or a code smell indicating a need for refactoring. Figure 4.6 shows the distribution of class stereotypes in the PowerShell system as an example.

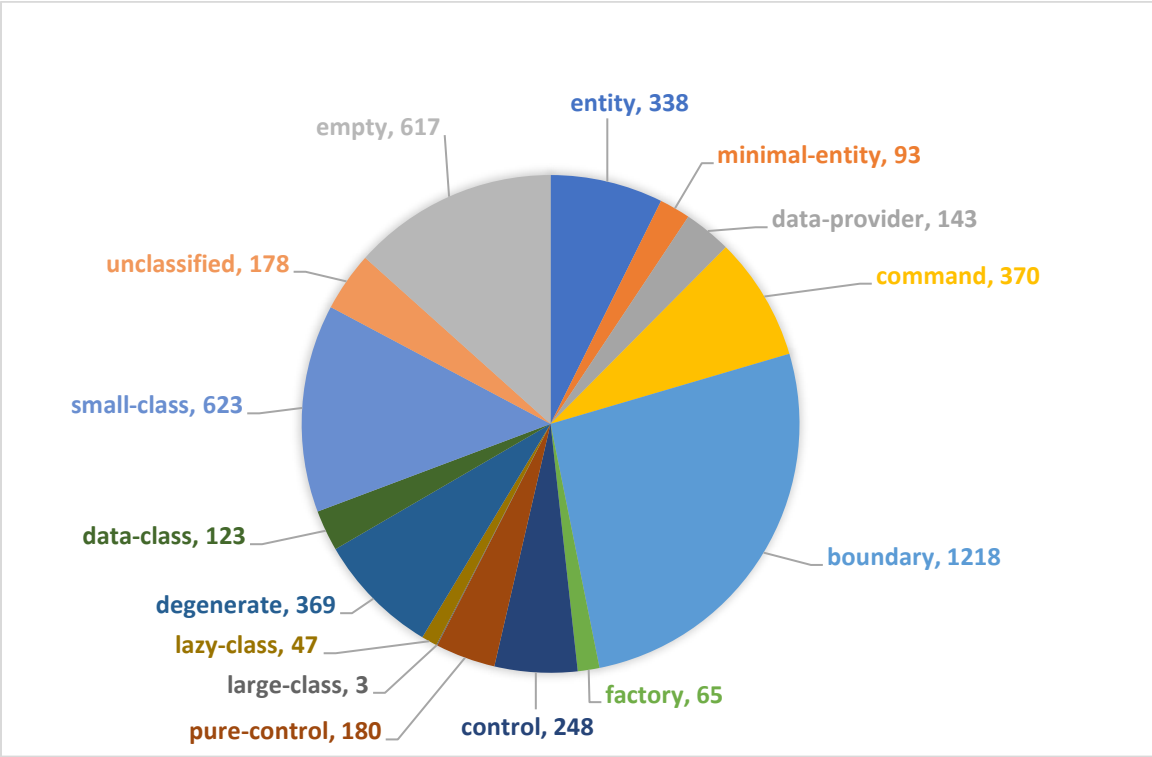


Figure 4.6 Visualizing Distribution of Class Stereotypes in PowerShell

CHAPTER 5

Empirical Study

This chapter presents a study that examines the distribution of method and class stereotypes across three popular programming languages: C++, C#, and Java. Stereotypes provide insights into the roles of methods and classes within software systems. While previous research has focused on individual languages, our study explores stereotype usage across different languages.

We collected a diverse set of software systems to ensure a broad coverage of system domains. We then used Stereocode to extract method and class stereotypes from these systems to form a signature for each system. These signatures were then used as input for a clustering algorithm. The main objective of this empirical study is to investigate the following research question:

- Do software systems cluster primarily by their domain, or do they cluster based on the programming language used?

5.1 Data Collection and Processing

To conduct this empirical study, we collected a set of open-source software systems from three different domains: GUI Libraries, Web Frameworks, and Game Engines for three programming languages: C++, C#, and Java. The systems are manually curated from GitHub using the advanced search feature by searching for the most popular repositories in each category. To ensure a fair comparison, we selected systems within each domain that share architectural similarities across the three languages. Furthermore, we ensured that the primary language (i.e., C++, C#, or Java) makes up over 80% of the total codebase for each selected system, wherever possible. Table 5.1 shows the complete list of systems selected for the study.

After the repositories were collected and cloned locally, we employed a two-step process to extract the stereotypes. Initially, we convert each system to the srcML format. Then, we run Stereocode on each archive to extract the method and class stereotypes. The stereotyping process is automated using Python scripts. The resulting count of stereotyped methods and classes is presented in Table 5.2.

System Name	Language	Domain
yue-0.14.1	C++	GUIs: Libraries/Framework used for building graphical user interfaces
wxWidgets-3.2.4		
nana-1.7.4		
cegui-0-8-7		
qtbase-6.6.1		
eto-2.8.2	C#	
wpf-8.0.3		
guiSharp-1.0.31		
avalonia-11.0.7		
maui-8.0.14		
lanterna-3.1.1	Java	
modernUI-3.10.0		
legui-3.3.1		
awt-23.6		
swing-23.6		
urho3d-1.9.0	C++	Game Engines: Applications used for video game creation
lumixEngine-2.0		
stride-4.2.0.2067	C#	
sanAndreasUnity-4.0		
litiengine-0.8.0	Java	
FXGL-21		
CPPWebFramework-1.0.0	C++	Web: Frameworks designed to support the development of web applications including web services, web resources, and web APIs
oatpp-1.3.0		
drogon-v1.9.3		
wfrest-0.9.4		
aspnetboilerplate-9.1	C#	
nancy-1.4.4		
aspnetcore-8.0.2		
abp-8.0.3		
jooby-3.0.7	Java	
rapidoid-5.1.0		
spark-2.9.3		
ninja-7.0.0		

Table 5.1 List of Selected Software Repositories Ordered by Language and Domain

System Name	Stereotyped Method Count	Stereotyped Class Count
yue-0.14.1	2133	479
wxWidgets-3.2.4	43232	4392
nana-1.7.4	2875	583
cegui-0-8-7	19528	962
qtbase-6.6.1	88676	11196
eto-2.8.2	6729	1192
wpf-8.0.3	40899	6147
guiSharp-1.0.31	612	78
avalonia-11.0.7	10106	3304
maui-8.0.14	19052	6689
lanterna-3.1.1	1726	319
modernUI-3.10.0	5784	443
legui-3.3.1	1338	331
awt-23.6	3604	389
swing-23.6	9262	666
urho3d-1.9.0	23603	5879
lumixEngine-2.0	4016	1356
stride-4.2.0.2067	20861	6031
sanAndreasUnity-4.0	1591	373
litiengine-0.8.0	3871	545
FXGL-21.zip	2198	386
CPPWebFramework-1.0.0	445	76
oatpp-1.3.0	1313	421
drogon-v1.9.3	1054	230
wfrest-0.9.4	747	201
aspnetboilerplate-9.1	4609	2128
nancy-1.4.4	3343	863
aspnetcore-8.0.2	3357	1039
abp-8.0.3	8661	5475
jooby-3.0.7	3804	843
rapidoid-5.1.0	2952	750
spark-2.9.3	794	181
ninja-7.0.0	2317	423

Table 5.2 Stereotyped Method and Class Counts for the Analyzed Repositories

To prepare the extracted stereotypes for clustering analysis, we used another Python script to process the stereotyped repositories and consolidate the data into two CSV files. Figure 5.1 and Figure 5.2 show samples of the combined data files for the unique method and class stereotypes where each record, or row, represents a system's distribution signature. The list of columns, or features, represents the set of all unique stereotypes of all systems.

	predicate set collaborator	get set collaborator	get predicate collaborator	non-void-command factory collaborator	get collaborator	get non-void-command	get predicate set collaborator	get void-accessor command collaborator	void-accessor controller	get factory collaborator	property set factory collaborator	get non-void-command factory	get set factory
yue-0.15.0 (GUI) (C++)	0.1406	0	0	0	3.0005	0.0469	0	0	1.7815	0	0	0	0
wxWidgets-3.2.4 (GUI) (C++)	0.5783	0.0625	0	0.0393	4.7627	0.0139	0.0046	0	0.4626	0	0.0254	0	0.0162
qtbase-6.6.1 (GUI) (C++)	0.2019	0.0259	0.009	0.0767	2.1945	0.0056	0	0.0011	0.3158	0.0011	0.0056	0	0.0023
nana-1.7.4 (GUI) (C++)	0.4522	0.0348	0.0348	0	2.8174	0.0348	0	0	0.3478	0	0	0	0
cegui-0-8-7 (GUI) (C++)	0.0666	0.0205	0	0.0051	2.2685	0	0	0	0.169	0	0.0256	0	0
avalonia-11.0.7 (GUI) (C#)	0.188	0.2276	0	0.1385	1.1083	0.0099	0	0	0.2672	0	0.2474	0	0
guiSharp-1.0.31 (GUI) (C#)	0	0	0	0	1.7974	0	0	0	0	0	0	0	0
eto-2.8.2 (GUI) (C#)	0.1337	0.0892	0	1.07	3.5072	0	0	0	0.2972	0	0.9511	0	0.0149
maui-8.0.14 (GUI) (C#)	0.105	0.1102	0.0262	0.084	1.9945	0.0105	0	0	0.1417	0	0.189	0	0
wpf-8.0.3 (GUI) (C#)	0.1516	0.401	0.0073	0.1271	7.5674	0.0513	0.0024	0	0.1394	0	0.1174	0.0024	0.0049
legui-3.3.1 (GUI) (Java)	0.1495	0	0	0.5232	13.901	0	0	0	0	0.2242	0	0	0
modernUI-3.10.0 (GUI) (Java)	0.1729	0.0346	0	0.0346	2.9391	0.1729	0.0173	0	0.0519	0	0.0346	0	0
lanterna-3.1.2 (GUI) (Java)	0	0.2897	0	0.1159	5.562	0	0	0	0	0	0.1159	0	0
swing-23.6 (GUI) (Java)	0.0324	0.2159	0	0.2267	4.0488	0.0864	0	0	0.0108	0.0108	0.0108	0	0
awt-23.6 (GUI) (Java)	0.0555	0.0832	0	0.0832	2.9412	0.1942	0	0	0.0277	0	0	0	0
urho3d-1.9.0 (Engine) (C++)	0.1737	0.0254	0.0085	0.0169	3.2242	0.0169	0	0	0.8135	0	0.0169	0.0085	0
lumixEngine-2.0 (Engine) (C++)	0.3237	0.0498	0	0	8.6404	0	0	0	0.8217	0	0	0	0
stride-4.2.0.2067 (Engine) (C#)	0.115	0.0719	0	0.1246	3.4466	0	0.0048	0	0.326	0.0048	0.1678	0	0.0048
sanAndreasUnity-4.0 (Engine) (C#)	0.0629	0.1257	0	0.1886	2.6398	0	0	0	0.0629	0	0.1257	0	0
litiengine-0.8.0 (Engine) (Java)	0.0258	0	0	0.155	1.1625	0	0	0	0.0258	0	0	0	0

Figure 5.1 Sample of the Distribution of Method Stereotypes for the Analyzed Repositories. Values are Percentages of Each Unique Method Stereotype

	command boundary	entity minimal-entity data-provider boundary	entity factory control data-class small-class	command boundary lazy-class	entity data-provider boundary data-class small-class	lazy-class small-class	factory degenerate	command data-class	data-class	boundary control	pure-control degenerate small-class	control lazy-class data-class small-class
yue-0.15.0 (GUI) (C++)	1.8789	0	0	0	0	0	0	0	0	0	0.8351	0
wxWidgets-3.2.4 (GUI) (C++)	3.7568	0.0228	0	0	0	0.0911	0.0228	0.0455	0.3188	0	5.123	0
qtbase-6.6.1 (GUI) (C++)	3.144	0	0	0.0268	0	0.0982	0	0.0268	0.1876	0	4.3944	0
nana-1.7.4 (GUI) (C++)	2.0583	0	0	0	0	0	0	0	0.3431	0	5.1458	0
cegui-0-8-7 (GUI) (C++)	4.0541	0	0	0	0	0.5198	0	0	0	0	5.8212	0
avalonia-11.0.7 (GUI) (C#)	4.8729	0	0	0.0303	0.0303	0.3329	0	0	0.0303	0	8.7167	0
guiSharp-1.0.31 (GUI) (C#)	8.9744	0	0	0	0	0	0	0	0	0	6.4103	0
eto-2.8.2 (GUI) (C#)	5.0336	0	0	0.0839	0	0.0839	0.0839	0	0	0	5.3691	0
maui-8.0.14 (GUI) (C#)	5.7109	0	0.0149	0.0299	0	0.4186	0	0.0598	0.0149	0	12.0945	0.0299
wpf-8.0.3 (GUI) (C#)	2.1637	0	0	0.0325	0	0.4067	0	0	0.1464	0.0488	2.5866	0
legui-3.3.1 (GUI) (Java)	1.8127	0	0	0	0	0.3021	0	0	0	0	12.3867	0
modernUI-3.10.0 (GUI) (Java)	6.0948	0	0	0	0	0.2257	0	0	0.2257	0	2.0316	0
lanterna-3.1.2 (GUI) (Java)	4.3887	0	0	0	0	0	0	0	0	0.3135	5.3292	0
swing-23.6 (GUI) (Java)	11.1111	0	0	0	0	0	0	0	0	0	1.3514	0
awt-23.6 (GUI) (Java)	4.1131	0	0	0	0	0	0	0	0	0	1.2853	0
urho3d-1.9.0 (Engine) (C++)	2.1432	0.017	0	0	0	0.017	0	0	0.051	0	2.0752	0
lumixEngine-2.0 (Engine) (C++)	1.3274	0	0	0	0	0.1475	0	0	0.295	0	2.4336	0
stride-4.2.0.2067 (Engine) (C#)	6.7816	0	0	0.0497	0	0.0663	0.0332	0	0.0166	0	7.478	0
sanAndreasUnity-4.0 (Engine) (C#)	14.7453	0	0	0	0	0.2681	0	0	0.2681	0	4.2895	0
litiengine-0.8.0 (Engine) (Java)	7.5229	0	0	0	0	0	0	0	0	0	6.2385	0

Figure 5.2 Sample of the Distribution of Class Stereotypes for the Analyzed Repositories. Values are Percentages of Each Unique Class Stereotype

To answer the research question outlined in the introduction, we need to cluster the collected method and class stereotypes. Clustering is an unsupervised clustering method that groups similar entities together, and in our case, it groups systems that are architecturally similar together. This can be helpful for the discovery of common patterns. In this study, we perform hierarchical clustering on the combined data files for the method and class stereotypes to find the similarities between the systems irrespective of the language or domain. Hierarchical clustering is an agglomerative technique that seeks to build a hierarchy of clusters using some distance measure by considering each system as a

separate cluster and iteratively merges the most similar systems using until a single cluster remains without having to specify a predefined number of clusters. We specifically utilized the Ward's method with the Euclidean distance as the clustering criterion. Ward's method clusters the systems by minimizing the total within-cluster variance to ensure the creation of meaningful distinct clusters [Vijaya et al. 2019]. In addition, Ward's method does not require specifying the number of clusters in advance, which is helpful since it allows us to explore the natural grouping of the systems. The approach is well-suited for our analysis as we can visualize the clustering process as a dendrogram, which is a tree-like structure that shows the hierarchical structure of the clusters.

5.2 Results and Discussion

The result of the clustering process using method stereotypes yielded three large clusters highlighted in orange, green, and red as shown in Figure 5.3. The gray dotted line represents the cutting threshold for the dendrogram. This threshold is a distance value that determines where the tree is cut, thus defining the clusters. Any branches in the dendrogram with a distance greater than this threshold are considered separate clusters. The dendrogram was visually inspected, and 20 was chosen as the threshold because it resulted in a meaningful grouping of the data into clusters. The orange cluster is composed entirely of systems programmed in C++, while the green cluster is fully C# except for one system. The red cluster consists mostly of Java systems withs some exceptions.

The clustering process for class stereotypes, shown in Figure 5.4, produced approximated results to the clustering using method stereotypes. The cutting threshold was selected to be 17 for similar reasoning as before. Class stereotypes are a set of rules that map to the distribution of method stereotypes. This clustering result indicates two things.

First, they show that these rules map relatively well over the method stereotypes, which validates the taxonomy of class stereotypes. Second, it shows that systems still cluster mostly based on the programming language. As for the small orange and gray clusters, they form distinct clusters separate from the larger ones where they could belong. This could suggest some fundamental differences in the design or architecture of the systems within these, despite most of them being written in the same programming language (i.e., Java or C#). It is possible that these systems use certain libraries, constructs, or design patterns that are different from those used in the systems within the purple cluster.

Overall, we see that systems cluster mainly based on the programming language used rather than their domain. This finding suggests that the choice of a programming language significantly impacts the design and architecture of software systems. This outcome can be attributed to several factors:

- Each language has a set of language constructs, libraries, and design principles that influence how developers' structure and organize their code.
- Each language seems to have communities and ecosystems with established coding practices and design guidelines specific to that language.

These factors together influence the way software systems are designed and structured. These clustering results highlight the importance of the choice of programming language in software development. It is not just a tool for implementation, but a determinant of the system's design and architecture.

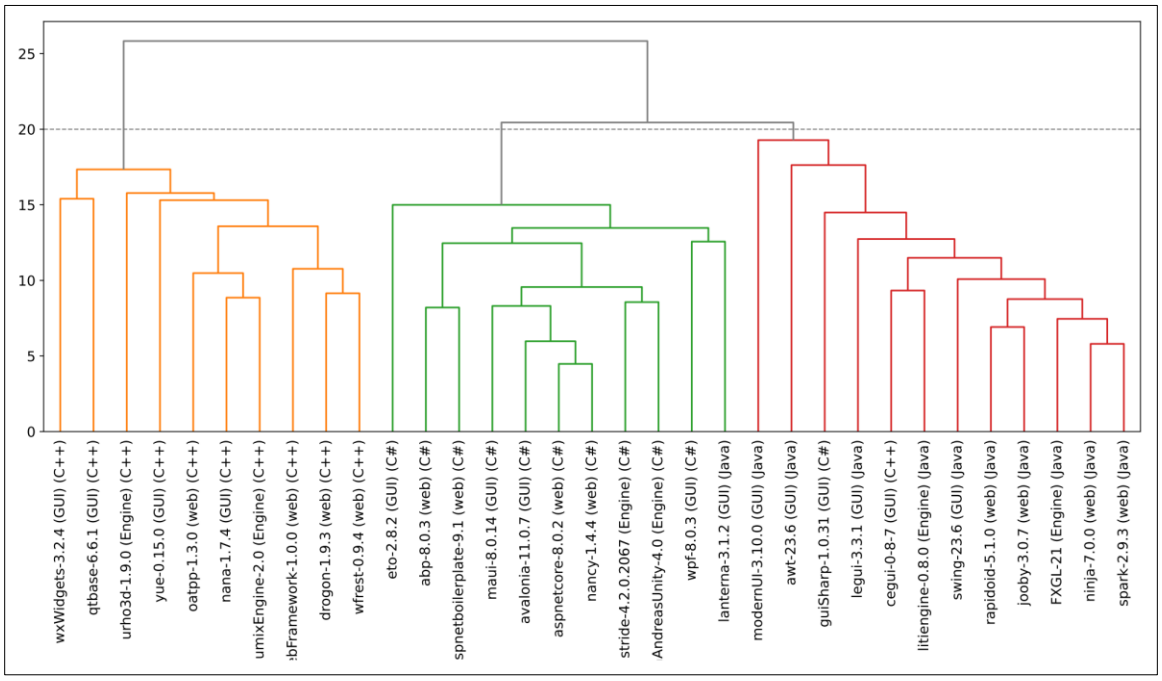


Figure 5.3 Result of Hierarchical Clustering on Method Stereotypes using Ward

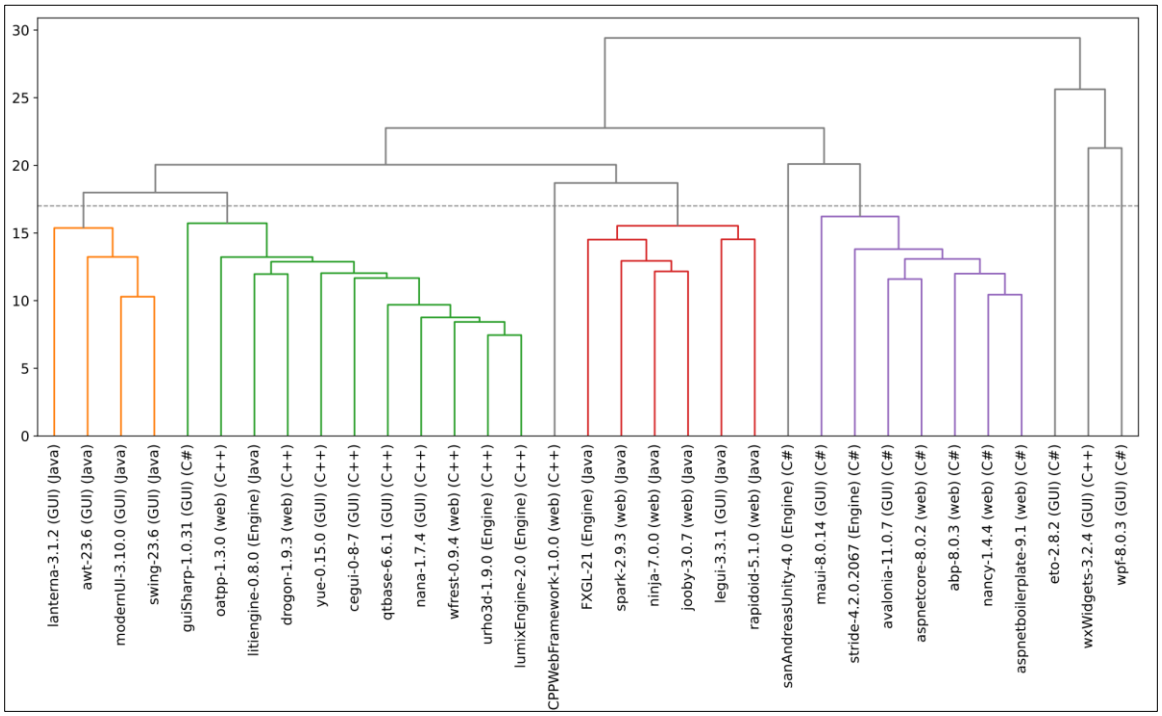


Figure 5.4 Result of Hierarchical Clustering on Class Stereotypes using Ward

To gain further insights into the stereotype distributions, we conducted a manual investigation of the systems across all three languages. One notable observation was the prevalence of methods classified as *controller stateless*. A *controller stateless* is a method that controls external objects without mutating the state of the current object.

The prevalence of *controller stateless* methods in C++ systems is interesting as C++ allows the use of free functions. This raises the question of why these methods are implemented as part of the class instead of being free functions. It suggests that there might be a design choice or pattern that favors the use of *stateless* methods within classes that perform external tasks. Figure 5.5 shows an example of such methods from the wxWidget that is stereotyped as a *controller stateless*. This method is a simple event handler that is triggered for a specific event (e.g., on button click) related to displaying screenshots. It is worth noting that this method does not access data members or other member functions of its class.

```
// @stereotype controller stateless
void ScreenshotFrame::OnSeeScreenshots(wxCommandEvent& WXUNUSED(event))
{
    wxString defaultDir = AutoCaptureMechanism::GetDefaultDirectoryAbsPath();

    if (wxFileName::DirExists(defaultDir))
        wxLaunchDefaultBrowser(defaultDir);
    else
        wxMessageBox(_("There isn't any screenshots yet."));
}
```

Figure 5.5 Method from the wxWidget System Stereotyped as a Controller Stateless.

In C# and Java, the situation is a bit different. Every function must be defined within a class as a method. As a result, developers are forced to include functions that would

typically be implemented as free functions in languages like C++. These functions often act as utility functions or perform tasks without relying on the state of the class.

Given the high percentage of these *controller stateless* methods, there is a possibility that these methods might themselves contain some form of free functions or self-contained logic that operates independently of the class state. In other words, while these methods are part of a class, they might be exhibiting characteristics that are similar to free functions. This observation could suggest the presence of emergent patterns in the form of free functions. These functions have no existing taxonomy and are left for future work for further investigation.

System Name	Controller Stateless Count
yue-0.15.0	94
wxWidgets-3.2.4	4364
qtbase-6.6.1	18101
nana-1.7.4	481
cegui-0-8-7	6480
avalonia-11.0.7	3043
guiSharp-1.0.31	37
eto-2.8.2	510
maui-8.0.14	4220
wpf-8.0.3	1748
legui-3.3.1	94
modernUI-3.10.0	134
lanterna-3.1.2	58
swing-23.6	277
awt-23.6	151
urho3d-1.9.0	2729
lumixEngine-2.0	313
stride-4.2.0.2067	1751
sanAndreasUnity-4.0	58
litiengine-0.8.0	614
FXGL-21	337
oatpp-1.3.0	139
drogon-1.9.3	135
CPPWebFramework-1.0.0	54
wfrest-0.9.4	113
aspnetcore-8.0.2	1371
nancy-1.4.4	1026
abp-8.0.3	1678
aspnetboilerplate-9.1	519
ninja-7.0.0	261
spark-2.9.3	121
rapidoid-5.1.0	132
jooby-3.0.7	668

Table 5.3 Frequency of the Controller Stateless Stereotype for All Systems.

CHAPTER 6

Conclusions and Future Work

The Stereocode tool has been significantly enhanced with the ability to perform deep static analysis to compute method and class stereotypes for entire software systems written in C++, C#, and Java at large scale. In addition, the tool is designed in C++ with proper documentation and build process to make it easy to build and use by users.

The empirical study conducted using the new Stereocode has revealed valuable new data into the usage of stereotypes across C++, C#, and Java. The results of the study show that systems mainly cluster by the programming language used rather than the domain (e.g., GUI). This observation highlights the significant influence of language-specific characteristics on software design and architecture. Furthermore, the study uncovered what seems like emergent patterns for methods that exhibit the characteristics of free functions.

6.1 Future Work

For future work in terms of Stereocode, support for classes with the same name could be added. Certain software systems contain multiple classes utilizing the same name. (e.g., in different namespaces or differentiated by preprocessor directives). There is no simple way to consider multiple classes with the same name. It could be problematic for locating the parent class for methods defined outside of the class (C++) or for collecting the inherited data members and methods. Currently, the tool combines multiple classes with the same name into one class and stereotypes these classes based on the combined class. In addition, we plan to investigate nested classes and nested local functions (C# only). Tracking nested structures can be tricky since a nested class could use or modify the

state of the parent class. Other future work includes adding proper support for namespaces and supporting additional programming languages.

In addition to these improvements to the tool, we also plan to further investigate the emergent patterns identified in this study. We aim to examine the behavior and purpose of these methods, which could potentially lead to the creation of a new taxonomy specifically for free functions and methods that exhibit similar characteristics.

The insights gained from the study not only enhance our understanding of software design and architecture, but also have practical implications for automatic refactoring. For instance, the identification of *controller stateless* methods that are not *static* could form the basis for an automatic refactoring process. This process could convert these methods to *static to* improve the efficiency and performance of the software system.

References

ABID, N.J., DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2015. Using Stereotypes in the Automatic Generation of Natural Language Summaries for C++ Methods. *IEEE International Conference on Software Maintenance and Evolution (ICSME'15)*, IEEE, 561–565.

ALHINDAWI, N., DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2013. Improving Feature Location by Enhancing Source Code with Stereotypes. *2013 IEEE International Conference on Software Maintenance*, 300–309.

ANDRAS, P., PAKHIRA, A., MORENO, L., AND MARCUS, A. 2013. A Measure to Assess the Behavior of Method Stereotypes in Object-Oriented Software. *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, 7–13.

ANDRIYEVSKA, O., DRAGAN, N., SIMOES, B., AND MALETIC, J.I. 2005. Evaluating UML Class Diagram Layout based on Architectural Importance. *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, 14–20.

COLLARD, M.L., DECKER, M.J., AND MALETIC, J.I. 2011. Lightweight Transformation and Fact Extraction with the srcML Toolkit. *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 173–184.

COLLARD, M.L., DECKER, M.J., AND MALETIC, J.I. 2013. srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration. *2013 IEEE International Conference on Software Maintenance*, IEEE, 516–519.

DOLEH, Z. 2021. *Stereocode: A Tool for Automatic Identification of Method Stereotypes*. Kent State University.

DRAGAN, N., COLLARD, M.L., HAMMAD, M., AND MALETIC, J.I. 2011. Categorizing Commits Based on Method Stereotypes. 520–523.

DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2006. Reverse Engineering Method Stereotypes. *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, IEEE, 24–34.

DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2009. Using Method Stereotype Distribution as a Signature Descriptor for Software Systems. *25th IEEE International Conference on Software Maintenance (ICSM'09)*, 567–570.

DRAGAN, N., COLLARD, M.L., AND MALETIC, J.I. 2010. Automatic Identification of Class Stereotypes. *IEEE International Conference on Software Maintenance (ICSM'10)*, IEEE, 1–10.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley.

GÖKMEN, R., HEIDRICH, D., SCHREIBER, A., AND BICHLMEIER, C. 2021. Stereotypes as Design Patterns for Serious Games to Enhance Software Comprehension. *2021 IEEE Conference on Games (CoG)*, 1–3.

LI, B., VENDOME, C., LINARES-VÁSQUEZ, M., AND POSHYVANYK, D. 2018. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. *Proceedings of the 26th Conference on Program Comprehension*, Association for Computing Machinery, 52–63.

LINARES-VÁSQUEZ, M., CORTÉS-COY, L.F., APONTE, J., AND POSHYVANYK, D. 2015. ChangeScribe: A Tool for Automatically Generating Commit Messages. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 709–712.

M. J. DECKER, C. D. NEWMAN, N. DRAGAN, M. L. COLLARD, J. I. MALETIC, AND N. A. KRAFT. 2018. Which Method-Stereotype Changes are Indicators of Code Smells? *18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*, 82–91.

MORENO, L., APONTE, J., SRIDHARA, G., MARCUS, A., POLLOCK, L., AND VIJAY-SHANKER, K. 2013. Automatic Generation of Natural Language Summaries for Java Classes. *21st International Conference on Program Comprehension (ICPC'13)*, 23–32.

MORENO, L. AND MARCUS, A. 2012. JStereoCode: automatically identifying method and class stereotypes in Java code. *27th IEEE/ACM International Conference on Automated Software Engineering*, 358–361.

NEWMAN, C.D., ALSUHAIBANI, R.S., COLLARD, M.L., AND MALETIC, J.I. 2017. Lexical Categories for Source Code Identifiers. *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 228–239.

NURWIDYANTORO, A., HO-QUANG, T., AND CHAUDRON, M.R.V. 2019. Automated Classification of Class Role-Stereotypes via Machine Learning. *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*, Association for Computing Machinery, 79–88.

SHEN, J., SUN, X., LI, B., YANG, H., AND HU, J. 2016. On Automatic Summarization of What and Why Information in Source Code Changes. *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 103–112.

VIJAYA, SHARMA, S., AND BATRA, N. 2019. Comparative Study of Single Linkage, Complete Linkage, and Ward Method of Agglomerative Clustering. *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, 568–573.