# STEREOCODE: A TOOL FOR AUTOMATIC IDENTIFICATION OF METHOD STEREOTYPES

A thesis submitted to

Kent State University in partial

fulfillment of the requirements for the

degree of Master of Computer Science

by

Zane Doleh

December 2021

Thesis written by

Zane Y. Doleh

B.S., Kent State University, 2018

M.S., Kent State University, 2021

Approved by

Jonathan Maletic_____, Advisor

Javed I. Khan_____, Chair, Department of Computer Science

Mandy Munro-Stasiuk_____, Interim Dean, College of Arts and Sciences

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

## ACKNOWLEDGEMENTS

I would like to thank my parents Yaser and Dawn Doleh and my sister Emily for always encouraging me to do my best. The constant support and loyalty of my family and friends, Ashley, Mateo, and Lloyd have been vital. Without them, none of this would have been possible.

I would like to thank my advisor Dr. Jonathan Maletic for approaching me with this topic and his valuable guidance throughout the project. His support has been crucial in the development of the thesis.

Lastly, I would like to thank the members of my defense committee for taking the time to hear my defense.

Zane Doleh

October 29, 2021, Kent, Ohio

CHAPTER 1

# Introduction

The work presented here investigates the problem of automatically reverse engineering method stereotypes. Stereotypes widely recognized by the development and maintenance communities include *constructor*, *destructor*, *accessor*, *predicate*, and *mutator*. These are decades old terms that are commonly used. A constructor is a method for initializing an object of a class; destructor is a method for destroying an object (cleaning up the memory) when the object goes out of scope. An accessor is a method used to read the members of a class and it returns the current state of an object but does not change it. A common use for accessors is to test for truth or falsity of a condition and such methods are called predicates. A mutator is a method used to modify members of a class, to change the state of an object.

However, very few software systems have this information explicitly documented in the source code and while this may be simple to do manually for a small number of methods it is very costly to do for an entire (large) system. We feel method stereotype information forms the basis for supporting more sophisticated types of design recovery and program comprehension. Given accurate information about method stereotypes, several things can be deduced/inferred in the context of a class or interacting classes. For instance, determining method stereotypes is the first step in identifying the stereotype of a class, say boundary, entity, or control. Knowing class stereotypes allows us to determine

architectural importance for automated layout of class diagrams or architectural level understanding.

Additionally, stereotype information can support more precise calculation of metrics. For example, it is well known that LCOM [Chidamber and Kemerer 1994] metrics are biased by certain types of methods (e.g., accessors and constructors). One can develop metrics that take this information into account. Good method abstraction is typically a requirement for good object abstraction. As such, metrics to assess how object oriented a class or system is based on method stereotypes is a reasonable objective. Other metrics that deal with change can also be envisioned. Changes in a method's stereotype due to modification may indicate major design changes to the class rather than a simple fix.

A number of studies [Staron et al. 2006; Genero et al. 2008; Ricca et al. 2010; Andriyevska et al. 2005; Yusuf et al. 2007; Sharif and Maletic 2009] demonstrate the benefits of using stereotypes, which reflect semantics, in program comprehension, design, and software maintenance tasks. Using class stereotype information [Andriyevska et al. 2005; Yusuf et al. 2007; Sharif and Maletic 2009] as a factor in laying out UML class diagrams has shown to improve the comprehensibility of the diagram. Staron et al. [Staron et al. 2006] show the effectiveness of class stereotypes based on domain model in program comprehension.

## 1.1 Motivation

Currently there are no general tools usable for automatically identifying method stereotypes. There are two research prototype tools that are difficult to use or limited in functionality. The first tool [Dragan et al. 2006] does not take into account all class

information and is limited in the static analysis it conducts. The other tool is language specific (Java) [Moreno and Marcus 2012].

The work presented here aims to be more language independent and conduct more complex static analysis of a given class declaration and definition. This will support a more accurate calculation of a method's stereotype.

## 1.2 Problem Statement

The goal is to build a tool that takes in header (.hpp) and implementation (.cpp) files of a class, and assign each method a stereotype, based on the work of Dragan [Dragan et al. 2006]. The .hpp and .cpp files are translated into srcML (source code markup language) and combined in an XML file called an archive. The stereotypes are stored in a symbol table (vector) until all the types have been given the chance to label methods. The archive is returned with the stereotypes as attributes on the function element (e.g., <function stereotype= "command collaborator">. Also, a CSV file is created storing the method headers and its stereotype.

## 1.3 Contribution

The main contribution is the tool, called stereocode, that automatically identifies method stereotypes. Stereocode relies on srcML for evaluation of xpath expressions and for the input of archive files. The tool is evaluated and compared to a previous research prototype. Stereocode will be available on GitHub and licensed as an open-source software.

## 1.4 Organization of the Thesis

This work is organized as follows: CHAPTER 2 includes background information on stereotypes and srcML and related work. CHAPTER 3 provides a detailed description of how the tool determines the correct stereotype for each method. CHAPTER 4 discusses the evaluation of the tool on the open-source system HippoDraw, comparing a research prototype from a related work vs stereocode 1.0. CHAPTER 5 talks about future work and conclusions.

CHAPTER 2

# Background and Related Work

In this chapter, we discuss the taxonomy of method stereotypes, background on srcML and related work.

## 2.1 Background on Stereotypes

Method stereotypes are beneficial in more ways than one. The small picture is that method stereotypes help achieve more precise calculation of metrics. The work has now grown into a means to classify entire software systems and categorize changes during software evolution [Dragan 2011].

A taxonomy of method stereotypes is presented by Dragan [Dragan et al. 2006]. This work categorizes stereotypes into three main groups based on the method's purpose. The structural methods include accessors and mutators. These methods are responsible for accessing and changing the state of the object they are a part of. The collaborational methods communicate with other objects, e.g., using a pointer to the object in parameter or local variable. This includes collaborational-accessors, collaborational-mutators, and controllers. The creational types are responsible for creating and destroying objects, and the degenerate methods label unfinished and other code-smell methods.

The accessor methods are read-only, meaning that they do not change the value of any data members. The stereotypes under this category are get, predicate, property and void accessor. The get type returns data members of the class and are usually simple one-line methods. The predicate stereotype returns a Boolean based on information (i.e., data

member's values) of the class. Boolean data members that are returned will be labeled as a get rather than predicate. Property methods, like predicates, return information calculated based on the values of data members. This stereotype returns values that are not bool or void. Lastly, the void accessor methods return information using a parameter. These methods have a return type of void and should have a pass by reference parameter that is assigned a value.

The mutator stereotypes are the ones that make changes to data members. The types in this category are set and command. The set stereotype is the simpler of the two. This type sets the value of data members only once. Multiple changes to the same data member or more than one change to different data members would not count for this stereotype. Setters must return a Boolean or void. The command stereotype is for more complex changes to the class, for example changing multiple data members. The exact cases in which the command stereotype can be applied is covered in Ch 3. Command methods that do not return a Boolean or void are labeled non-void-command.

The collaborational stereotypes signify methods that interact with other objects. The primary types under this category are the collaborator-accessors and collaborator-command. Collaborator-accessors include collaborational-predicate, collaborational-property and collaborational-void-accessor. These are methods that, despite being const, do not access the state of the object they belong to. Similarly, the collaborational-command does not change the state of the object it is a part of, even though it is not const. A secondary type of collaborator is given to any method that uses an object, either as a local variable,

parameter or return type. Controller is the name given to methods that can only be labeled as collaborators.

Creational methods are responsible for creating and destroying objects. The stereotypes under this category are factory, copy-constructor, constructors and destructors. Constructors, including the copy-constructor, and destructors have very specific syntax making them easy to identify. In srcML they are even given special constructor and destructor tags, and for these reasons they are not included in the stereocode tool. Factory is included, however, and its purpose is to create objects and return them. A common implementation of a factory is the singleton pattern.

The degenerate category is one where the unimplemented methods belong. This category includes empty, stateless (formerly named pure stateless), and wrapper (renamed from stateless). The empty stereotype is for methods that contain no statements, and this shows methods that are not finished. The stateless stereotype is for methods that do access or change the state of the object. Similarly, the wrapper type does not change or access the object's state, but its purpose is to call another method. A table containing a summary of the stereotypes and a short description in included below.

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

**Table 1. Taxonomy of Stereotypes**

## 2.2 Background on srcML

SrcML is an XML format that represents source code marked with tags that contain information from the abstract syntax tree [Collard et al. 2011; Collard et al. 2003; Maletic et al. 2002]. There is also a tool with the same name to convert source code into the srcML format. The source code's text is preserved so the srcML can be transformed back to the original code without losing any information. An example of a srcML file is shown below in figure 1. Each source code file is wrapped in a tag called unit. Multiple source code files can be combined into one XML file called an archive, meaning an archive can contain more than one unit. SrcML currently supports C/C++/C# and java.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <unit xmlns="http://www.srcML.org/srcML/src" xmlns:cpp="http://www.srcML.org/srcML/cpp" revision="1.0.0"
   language="C++" filename="rotate.cpp"><cpp:include>#<cpp:directive>include</cpp:directive> <
   cpp:file>"rotate.h"</cpp:file></cpp:include>
3
4  <class>class <name>cube</name><block>{<private type="default">
5    <function_decl><type><name>void</name></type> <name>rotate</name><parameter_list>(<parameter><decl><type><name>
     int</name><modifier>&amp;amp;</modifier></type></decl></parameter>, <parameter><decl><type><name>
     int</name><modifier>&amp;amp;</modifier></type></decl></parameter>, <parameter><decl><type><name>
     int</name><modifier>&amp;amp;</modifier></type></decl></parameter>)</parameter_list>;</function_decl>
6
7    </private><private>private:
8      <decl_stmt><decl><type><name>int</name></type> <name>x</name></decl>;</decl_stmt>
9      <decl_stmt><decl><type><name>int</name></type> <name>y</name></decl>;</decl_stmt>
10     <decl_stmt><decl><type><name>int</name></type> <name>z</name></decl>;</decl_stmt>
11 </private>}</block>;</class>
12
13 <comment type="line">// rotate three values</comment>
14 <function><type><name>void</name></type> <name><name>cube</name><operator>::</operator><name>
   rotate</name></name><parameter_list>(<parameter><decl><type><name>int</name><modifier>&amp;amp;</modifier></type> <name>
   n1</name></decl></parameter>, <parameter><decl><type><name>int</name><modifier>&amp;amp;</modifier></type> <name>
   n2</name></decl></parameter>, <parameter><decl><type><name>int</name><modifier>&amp;amp;</modifier></type> <name>
   n3</name></decl></parameter>)</parameter_list>
15 <block>{<block_content>
16   <comment type="line">// copy original values</comment>
17   <decl_stmt><decl><type><name>int</name></type> <name>tn1</name> <init>= <expr><name>
     n1</name></expr></init></decl>, <decl><type ref="prev"/><name>tn2</name> <init>= <expr><name>
     n2</name></expr></init></decl>, <decl><type ref="prev"/><name>tn3</name> <init>= <expr><name>
     n3</name></expr></init></decl>;</decl_stmt>
18
19   <comment type="line">// move</comment>
20   <expr_stmt><expr><name>n1</name> <operator>=</operator> <name>tn3</name></expr>;</expr_stmt>
21   <expr_stmt><expr><name>n2</name> <operator>=</operator> <name>tn1</name></expr>;</expr_stmt>
22   <expr_stmt><expr><name>n3</name> <operator>=</operator> <name>tn2</name></expr>;</expr_stmt>
23 </block_content>}</block></function>
24 </unit>
```

**Figure 1. Example of srcML**

The srcML toolkit also provides a way to select parts the srcML by evaluating xpath queries. This is done using the libsrcml library, a C++ library that includes functions to convert files to and from srcML and many other helpful functions dealing with srcML archives and units.

## 2.3 Related Work on Stereotypes

Fowler [Fowler 2000] classifies methods at the design level (i.e., UML class) and concentrates on the object's state. He gives the categories, *getting*, *setting*, *query* (accessor), and *modifier or command* (mutator). However, there is not distinction within the accessor and mutator groups.

Stroustrup [Stroustrup 2000]classifies methods the aim to help developers design class interfaces in C++. His classification includes *inspector* (accessor), *modifier* (mutator), *conversion* (produces an object of a different type based on the applied object). Several well-known programming and data structure textbooks have similar categories [Deitel and Deitel 2001; Savitch 1999; Tremblay and Cheston 2001; Weiss 1999]. Additionally, Deitel presents *predicate* and *utility* (or *helper*) methods. Predicates test whether a condition is true or false and utility methods are used by other methods and are not a part of the class interface. *Template* and *hook* methods are proposed [Gamma et al. 1995] to describe the behavior of methods within a class hierarchy. Template methods perform self-calls to abstract methods and hook methods are designed to be overridden in child classes. These approaches assume a forward engineering approach, relying on developers to document the stereotypes as methods are being written and this information must be maintained.

Other work uses stereotypes to solve problems. Workman [Workman 2002] proposes a method taxonomy for class categorization in Java to detect plagiarism. Some use of the collaboration between methods is considered but no means for identification is given. Clarke et al. [Clarke et al. 2003] present a taxonomy of classes for the identification of changes in object-oriented software. They consider properties and features of the class based on the inheritance hierarchy and types of data associated with the class. Other types of collaboration between classes and collaboration on the method level are not considered.

Visualization approaches to support method and class understanding are proposed in [Arevalo et al. 2003] and [Lanza and Ducasse 2001]. Arevalo et al. propose Xray views which groups methods based on state usage (state access), external/internal calls (self and super calls) and behavioral skeleton (client access) using concept analysis. This approach considers collaboration between groups of methods and data members in a single class in terms of direct or indirect accessors. However, no differentiation is done on whether the method reads or updates the data members. Lanza and Ducasse [Lanza and Ducasse 2001] consider categorization of classes based on a visual representation (blueprint) of the class as an attribute layer and a set of four method layers: initialization, interface, implementation and accessor. This approach provides semantic information on the method level, but collaborations between classes is limited to the generalization relationship.

Stereotypes are also used as a powerful tool in UML [Gogolla and Henderson-Sellers 2002]. They are used to emulate metamodel extensions and to support classification of objects in terms of assigning them certain features and properties[Atkinson et al. 2002].

Dragan et al. discuss the meaning of method stereotypes in more detail and the process they used to label methods in an earlier work [Dragan et al. 2006]. They outlined the first formal, in-depth study on method stereotypes, examined all the existing literature and described the taxonomy of method stereotypes. Also, a tool is presented, called stereocode, to automatically label methods by adding a comment above the method prototype with its stereotype. This tool is used as an inspiration for the one being presented in this thesis, and results are compared between these tools in chapter 4. This tool will be referred to as the research prototype from this point on.

Dragan and collogues expand on their method stereotypes in another paper [Dragan et al. 2010]. This paper introduces the tool to automatically identify class stereotypes based on the distribution of method stereotypes. The amount and types of method stereotypes determine which class stereotype is most appropriate for the class. They also show an updated taxonomy of method stereotypes, including incidental, empty, void accessor, controller and non-void command methods. Incidental was later renamed to pure stateless and is now called stateless.

Laura Moreno and Andrian Marcus have a tool to automatically identify java stereotypes called JStereocode [Moreno and Marcus 2012]. The tool works as a plug in for Eclipse and inserts comments for the method and class stereotypes. Additionally, they generate reports both at the class level for the methods and at the project level for the classes. The method stereotype taxonomy they present is very similar, they add two stereotypes to the previous taxonomy. Abstract methods which they define as methods that have no body. This is different from empty but still part of the degenerate category. Local

controller they define as a method that provides control logic using only local methods. This belongs to the collaborational category. The other method stereotypes are the same as previous taxonomy.

Dragan and collogues show some practical uses of stereotypes. Alhindawi explains that enhancing the source code with method stereotypes improves an information retrieval method called Latent Semantic Indexing, and this approach decreases the total effort a developer uses to locate features of the code [Alhindawi et al. 2013]. Dragan uses stereotypes to help categorize commit messages [Dragan et al. 2011]. This helps developers gain a better understanding of design changes over the lifetime of the project and aid in the documentation of commit messages. Stereotypes can also be used to automatically generate natural language summaries of C++ methods, as Abid describes [Abid et al. 2015]. Moreno et al. show this for java classes [Moreno et al. 2013]. This helps developers understand the code in times of missing or insufficient documentation. Decker et al. use changes in stereotypes to locate changes that are detrimental to the design of the system [M. J. Decker et al. 2018]. This can act as an alarm system notifying developers when the design of the system is hurt, and which change has likely caused it.

CHAPTER 3

# Stereocode Development and Architecture

In this chapter, we discuss the execution of stereocode, specifically how methods are assigned a stereotype. Section 3.1 describes the information that is gathered to identify methods and stored in a class. Section 3.2 details the process of assigning the methods a stereotype. Section 3.3 outlines how the assigned stereotypes are written to a srcML archive and a csv file.

Figure 2 shows the high-level architecture for stereocode. Source code files are translated to srcML format before stereocode is run. For each class, a header and an implementation file translate to a single archive. The first unit of the archive is assumed to be the header file and the next unit is the implementation file. Stereocode takes the archive file or file containing a list of archives and creates a new archive for each existing one that contains the methods labeled with their stereotype as an attribute to the function tag (e.g., <function stereotype= “predicate”>). In addition, stereocode generates a report (CSV file) of method headers, the input file path, and the assigned stereotype.

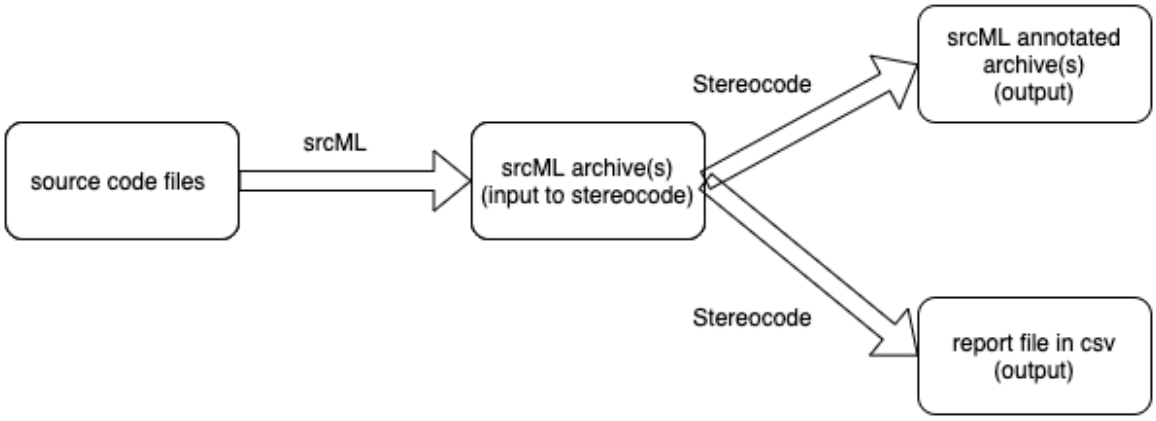


**Figure 2. High-level architecture**

Figure 3 below shows a more in-depth look at how stereocode works. First, the srcML archive is read and split into the header and implementation units. Then, xpath expressions are used to collect the necessary information from the files. Next, stereotypes are assigned to methods starting with accessor types, then mutator, factory, degenerate, and collaborator. After that is finished the report and output archive(s) are generated.

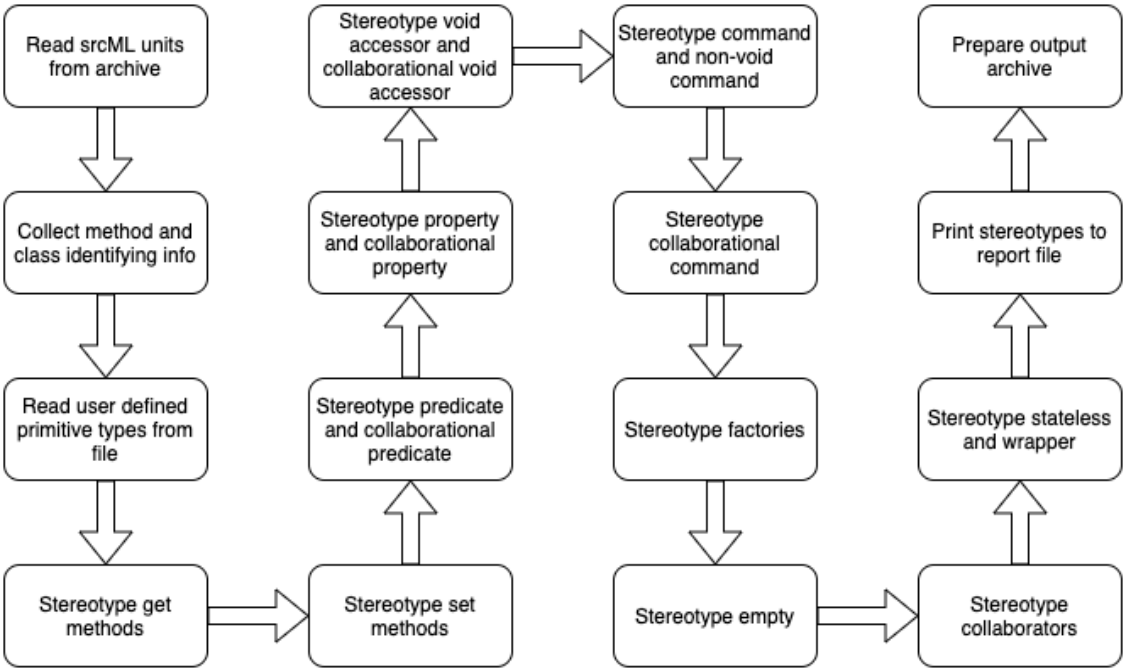


**Figure 3. Stereocode flow chart**

### 3.1 Preprocessing

Before the stereotypes are assigned to the methods, information is collected from the srcML archive that contains header and implementation files of one class given as input to stereocode. The header file is the first srcML unit of the archive and the implementation unit follows. Using libsrcml, xpath expressions can be defined and applied to the srcML units to select desired parts of the class or function. Using this approach, aspects of the class and functions can be easily obtained, for example, the data members (names and types) in the class definition or return expressions of a particular function. Xpath

expressions to find data member names and return expressions are shown in the figure below.

```
1   "//src:class//src:decl_stmt[not(ancestor::src:function)]/src:decl/src:name"
2
3
4   "//src:function[string(src:name)='method_names[i]' and string(src:type)=;
5       return_types[i] + and string(src:parameter_list)=;
6       parameter_lists[i] + and string(src:specifier)=;
7       specifiers[i] + ]//src:return/src:expr
8           [(count(*)=1 and src:name) or
9            (count(*)=2 and *[1][self::src:operator='*']
10           and *[2][self::src:name])]"
```

**Figure 4. Example Xpath Expressions**

Figure 4 shows xpath expressions. Line 1 is the expression to select data member names, and line 4-10 shows the expression to find return expressions that are candidates for the get stereotype. The values *method_names[i], return_types[i], parameter_lists[i] and specifiers[i]* are for a specific method. These are used to ensure the correct method is selected.

The information that is stored in a class is, class name, parent class name, if there is one, attribute names, attribute types, method information. The method information, that is stored for later identification of methods, includes return type, name, parameter list, and specifier. A list of data types that will not be considered collaboration types are also collected in the preprocessing step. These data types are provided in a file and may be added to by users. After all the information is collected the stereotypes can be assigned to methods starting with the get stereotype.

## 3.2 Stereotype Assignment

In order to label a method as a get, it must have at least one return expression that contains a data member. In order to determine if a method returns a data member, the return expressions must be collected. Return expressions with one element or two elements and the first element is a pointer operator are collected in order to better eliminate cases where the return expression cannot be a data member. Once return expressions are collected, they are compared with a list of data members collected from the class definition in the header unit of the archive. If a match is found the method is flagged that it returns a data member. In the case where none of the return expressions collected are matched with the list data members, the return expression is checked if it might be an inherited data member. An expression is an inherited data member if it does not contain an operator, is not true, false, or a literal, and is not a local variable or parameter. If any return expression is found to be an inherited data member, it is added to the list of attributes only if the class has a parent class. Classes without parent classes should not be inheriting any data members. The method is flagged as returning a data member if it contains a return expression that is an attribute or inherited attribute. Any method flagged as returning a data member will either be labeled as a get for const methods, or a non-const-get if the method is not const.

### 3.2.1 Predicate Stereotype

In order to label a method as a predicate the following must be true. The return type is Boolean, the method is const, and it does not return a data member. All the methods' specifiers and return types are saved from the preprocessing step, and the methods that

return data members were identified when stereotyping the getters. Predicates must also use a data member in an expression or contain at least one pure call.

Collaborational-predicate is a collaborator type that looks like a predicate, except it does not use data members and contains zero pure calls. A pure call is a call that is not static and is not a call for any method including constructors. In order to determine if a method uses data members, it either must include a call on a data member or the data member must be part of an expression. An xpath expression is applied that collects all expressions that are not below a throw statement, a catch statement, or part of a generic argument list (e.g., a vector argument). Expressions under a throw or catch statement are not considered part of normal execution of the method, and an expression under a generic argument list will always be a type so they are ignored. The names in those expressions are checked against the saved list of data members. Any names that do not appear to be data members are checked if they could be inherited data members, meaning they are not local variables or parameters. Inherited data members are treated as data members and added to the list of data members. In order to count the number of pure calls, another xpath expression is applied that collects all calls that do not follow the dot operator, the arrow operator or the new keyword. Calls that come after either the dot or arrow operator are considered real calls but not pure calls. Calls that follow the new keyword are constructor calls and do not count as a pure or real call. The dot operator and arrow operator may still be a part of the call that is collected, as well as the scope resolution operator for static calls. The count of the calls without these operators is the count of the pure calls of the method. Now that it is known if the method contains data members, and number of pure calls are

collected, the method may be labeled as a predicate or collaborational-predicate if appropriate.

### 3.2.2 Property Stereotype

The property stereotype is like predicate in that the same information about the methods is required for labeling. The return type of property methods cannot be void or Boolean, they must not return data members and they must be const. The differences between collaborational-property and property are the same as predicate and collaborational-predicate. The collaborational properties do not use data members and have zero pure calls in addition to property's requirements for return type, specifier and not returning data members. Return types and all methods that return data members have already been collected, and the number of pure calls and data members used are found in the same way as they were with predicate.

### 3.2.3 Void Accessor Stereotype

Void accessor is the last of the accessor stereotypes to be labeled. Void accessors must have a return type of void and a const specifier. Void accessors have at least one parameter that was passed by non-const reference, and the parameter name that is assigned to, meaning that it precedes an equal sign. If a parameter passed by non-const reference cannot be found, or cannot be found before an equal sign, the method is assumed to be a void accessor if it meets all other requirements and is not another type of accessor. This is an accurate assumption because void accessor is the last of the accessors labeled.

### 3.2.4 Set Stereotype

The set stereotype is the first mutator type to be labeled. The return type of set methods must be void or Boolean, the method cannot be const and the method must make only one change to data members. The return types and method specifiers are already saved so the number of changes to data members is calculated.

For each method, counting the number of changes to data members is split into two parts, changes using an assignment operator and changes using an increment or decrement operator. For each assignment operator, including compound operators, an xpath expression finds all uses of the operator anywhere in the method and if any of the preceding names is a data member, then the count of changes for that method is incremented based on the results of the xpath. For the increment and decrement operators the same process applies, an xpath expression looks for all increment and decrement operators in the method, but the name of the variable may be before or after the operator so both places are checked. Operators that appear inside loops are be counted as two changes because any loop should execute multiple times. Therefore, the assignment, increment and decrement operators which are children of for or while loops are counted again. For both assignment and increment operators, names are checked against local variable names and parameters to see if they could be inherited data members.

### 3.2.5 Command Stereotype

The command stereotype is one of the most complicated stereotypes to label. In order to label a method as a command it must not be const and be one of the three following cases as shown in the figure below. Case one, highlighted in blue, is when the method

changes data members one time and the number of real calls is more than one. Case two, shown in red, is when the method changes zero data members and there is at least one pure call or there is at least one call on a data member. Case three, in green, is when there is more than one change to data members. Xpath queries are used to find the names of real calls and the number of pure calls. The names of the real calls along with the local variable and parameter names are used to determine if there is a call on a data member, keeping in mind the data member may be inherited. To find the calls on data members the calling object is separated from the rest of the call by string manipulation. The calling object is the name that appears before the dot or arrow operator, if it is a data member than it meets the condition for case two. For all three cases, command's return type must be void or bool, otherwise the method will be labeled as non-void-command. Command may be a secondary stereotype as well. This only occurs when a method makes any number of changes to data members and is const. The purpose of this is to alert the user that the method should not have the command stereotype if it is an accessor.

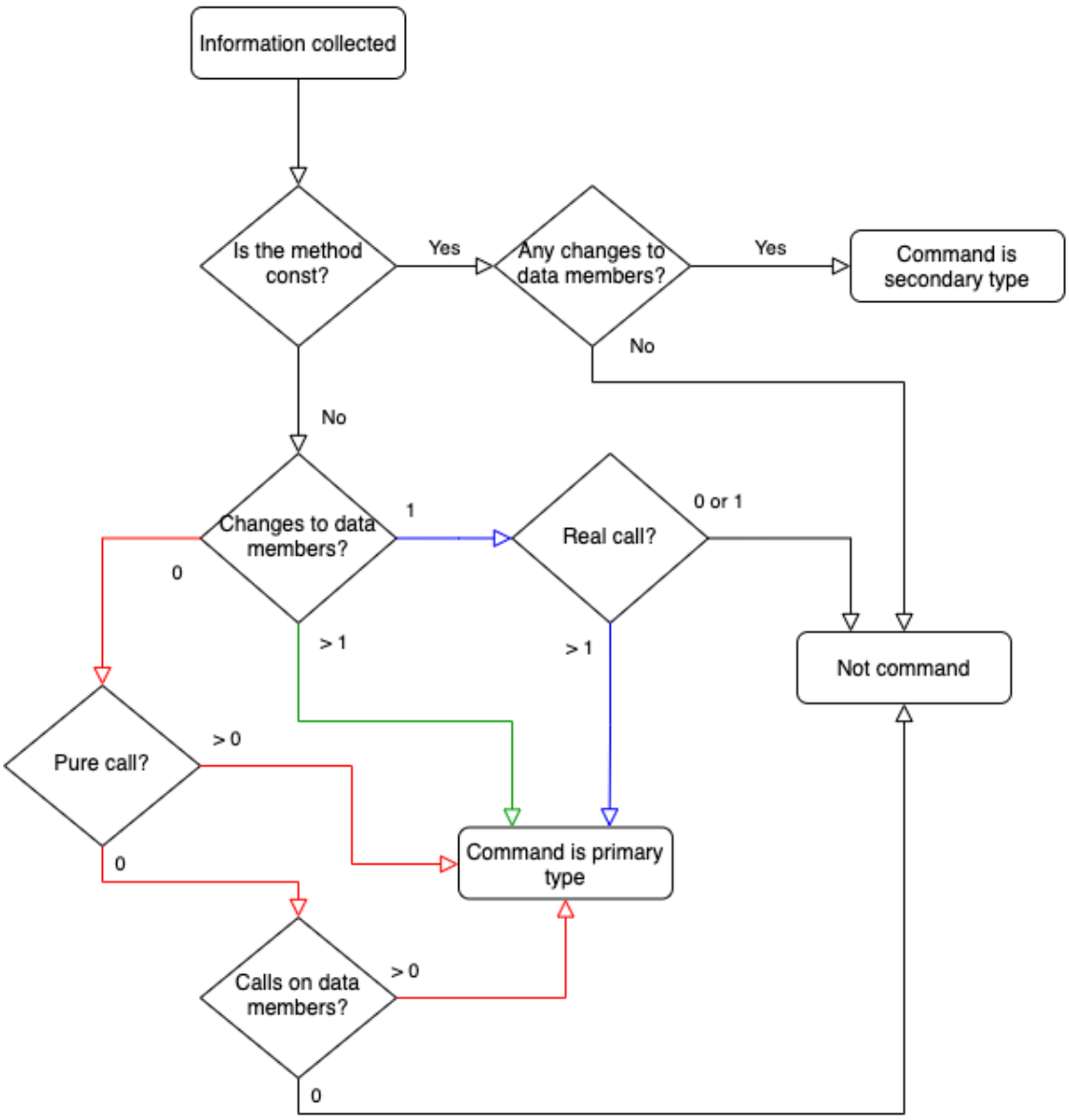


**Figure 5. Command flow chart**

Methods that have the collaborational command stereotype are not const, and do not contain any changes to data members. Collaborational command does not overlap with command, so there are no pure calls and no calls on data members. Also, there must be at least one real call, or a parameter or local variable is changed. The types of calls that will count are ones that contain the dot or arrow operator where the calling object is not a data member or calls using the new operator. These real calls are found using an xpath query. Next, the parameter and local variable names are collected, each name is checked if it is to the left of an equal sign. This is done using an xpath query that finds the name anywhere in the method and returns the next element following it. If the next element contains only one equal symbol, then the condition of a parameter or local variable changed is met. Collaborational-command is then assigned to methods that are not const, have zero changes to data members, do not have pure calls, have no calls on data members, and contain one real call that is not a pure call or write to a parameter or local variable. This is illustrated in the figure below.

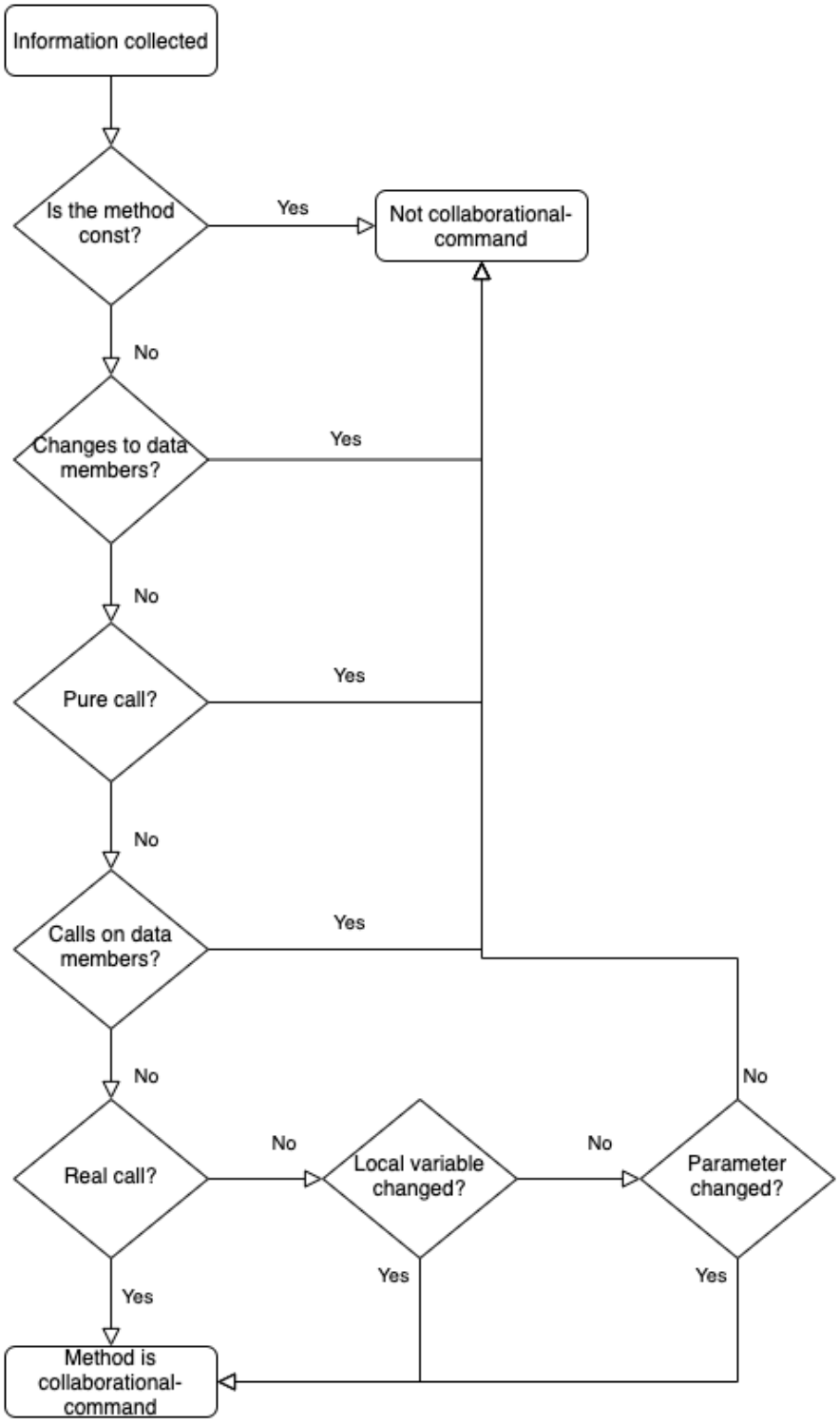


**Figure 6. Collaborational-command flow chart**

### 3.2.6 Collaborator Stereotype

The collaborator stereotype labels methods that use classes of a different type. Collaborator is a secondary stereotype, meaning that methods will have another stereotype that is the primary. Methods that are found to be only a collaborator are labeled controller. In order to label a method as a collaborator it must have a non-primitive type as a parameter, local variable or return type. Also, methods that use data members that are non-primitive pointer types will be a collaborator. A list of the attribute names with non-primitive types is created by testing each attribute type. Attribute types were collected from the header file in the preprocessing step. Inherited attribute's types are not collected so they are assumed to be standard types. Any method that uses an attribute in the list of non-primitive attributes will be labeled as a collaborator. Each method is tested with separate xpath queries to find non-primitive types in the local variables and parameters. Also, any non-primitive attribute found, with an additional query will set a flag to mark the method as a collaborator. Return types have already been collected so it is only checked if it is a primitive type. Classes cannot collaborate with themselves, so any type of the same class will not be counted towards collaboration.

### 3.2.7 Factory Stereotype

The factory stereotype is implemented as a special type of collaborator but belongs to the creational category. Factories must include a pointer to an object in their return type, and their return expression must be a local variable, a parameter, a data member or a call to another object's constructor using the keyword new, illustrated in figure 7. For each method, all return expressions, local variable names and parameter names are collected.

Whether the method returns a data member has already been collected as well as return types. Flags are set for the return type containing a pointer operator and if the return type is not a primitive type. All the return expressions are checked against the names of local variable and parameter names, or if the expression contains the keyword new. Since all factories must contain a new keyword, it may not always be in the return expression. Another xpath is evaluated, and a flag is set if the method contains a call that uses the new keyword. Now all the flags to determine if the method is a factory are set. In order to be a factory, the flags that need to be true are, the method returns an object, the method returns a pointer, the method contains a new call, the method has a return expression that is one of the following, a local variable, a parameter, a new call, or a data member as shown in the figure below.

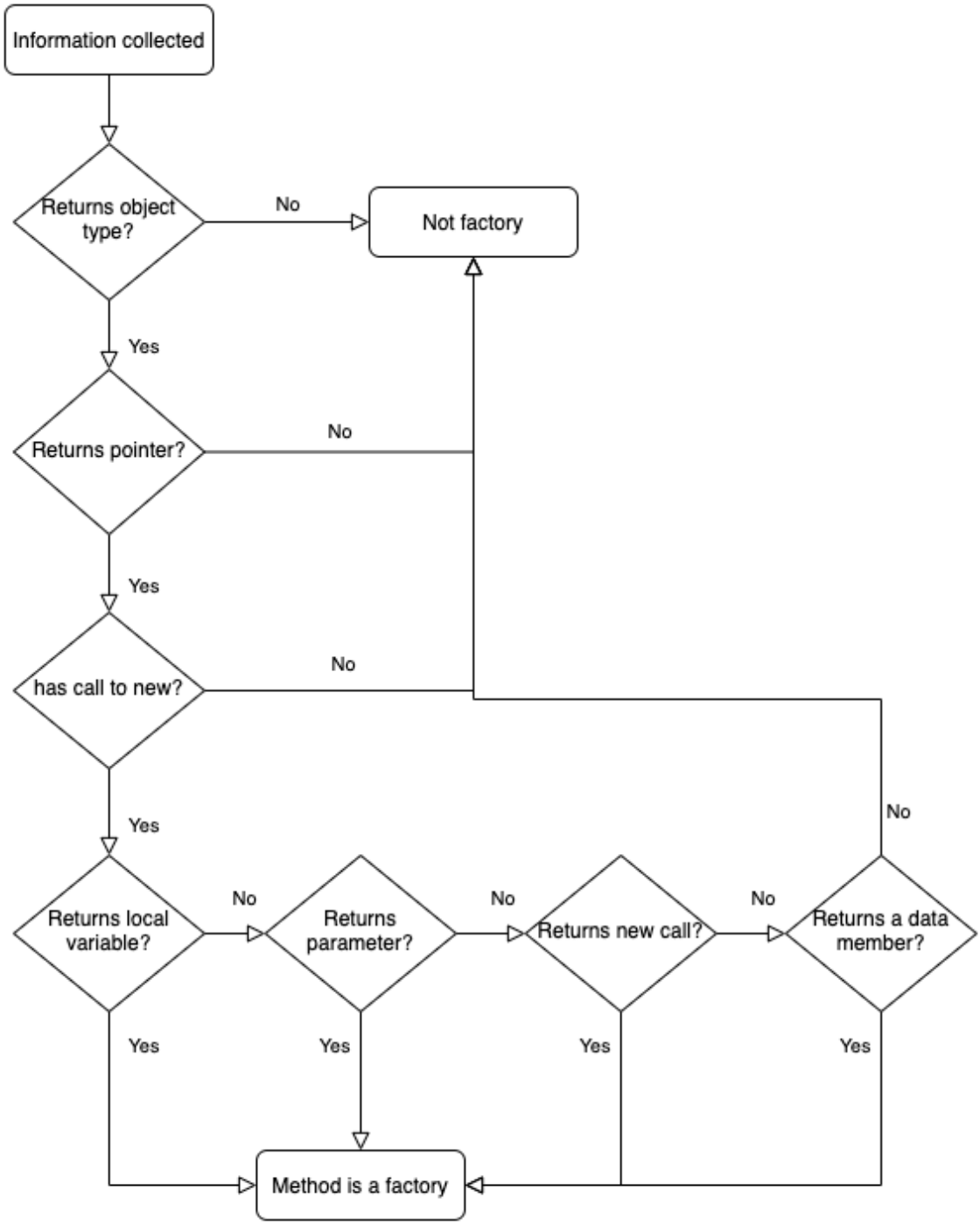


**Figure 7. Factory flow chart**

### 3.2.8 Empty Stereotype

The empty stereotype is a very simple one if the method does not have any statements, it is empty. Empty methods signify methods that have not been implemented yet. An xpath is written to show the methods that do not have statements. The xpath query is tested against each method, if it gives one result the method is empty if there are no results the method is not empty.

### 3.2.9 Wrapper and Stateless Stereotypes

The wrapper and stateless methods are labeled at the same time because they are very similar. The wrapper and stateless methods are secondary and degenerate stereotype, meaning they are not usually the first stereotype applied to methods and do not use any data members. A method with the primary stereotype of stateless or wrapper would be an indication of a bed method. The wrapper type has one call of any kind while stateless has more than one. This means calls can include the new keyword or the dot or arrow operator. Whether Data members are used in the method is found using the same technique that collaborational-accessors used.

## 3.3 Outputting the Results

While the information is being collected for each specific stereotype, the methods' stereotypes are stored as a vector of strings. After all the types have been given the chance to label the methods, all the methods should have a value for the stereotype. The next step is to label the function tag with an attribute called stereotype and insert the value of that function's stereotype. This is done by selecting each method with an xpath query and

inserting the attribute with a libsrcml function. With the annotated archive, it is easy to take the information in the attribute of the function tag and redocument the code with the method's stereotype. An optional report file containing all the method headers, their stereotypes and input file is created in csv format. With the report file you can better examine the stereotypes for the whole system.

CHAPTER 4

# Evaluation

This chapter discusses the evaluation of stereocode. Section 4.1 outlines how stereocode is tested and compares the results to the research prototype.

## 4.1 Results

Stereocode is tested on an open-source tool called HippoDraw containing 214 files and classes that have 2539 methods. A few of the files are discarded because they do not correctly translate to srcML. HippoDraw is chosen as the system to test because the research prototype has been run on the system and the method stereotypes are available. In the analysis of the research prototype, Dragan takes a sample of 365 methods and 329 of the stereotypes are found to be good or very good by a developer with multiple years of industry experience [Dragan et al. 2006].

Tables 2 and 3 above show the distribution of method stereotypes for HippoDraw found by stereocode. Every method is given a primary stereotype and zero or more secondary stereotypes. The controller stereotype is another name for collaborators that do not have any other primary stereotype. In 27 of the functions, the primary stereotype is stateless even though stateless is usually a secondary stereotype. In these cases, the tool is not able to label them with any other primary stereotype and should be a warning to the user that something may be off with that method (i.e., a possible code smell). Another warning to users is when command is a secondary stereotype. This occurs when methods

that are marked as const change a data member marked as mutable. Const methods should not be changing data members, so they are labeled with the secondary stereotype command to signal to the user the code smell in the method.

| Primary Stereotypes | 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 2. Distribution of primary stereotypes**

| Secondary Stereotypes | Count |
|---|---|
| Command | 31 |
| Stateless | 132 |
| Collaborator | 1667 |
| Wrapper | 156 |

**Table 3. Distribution of secondary stereotypes**

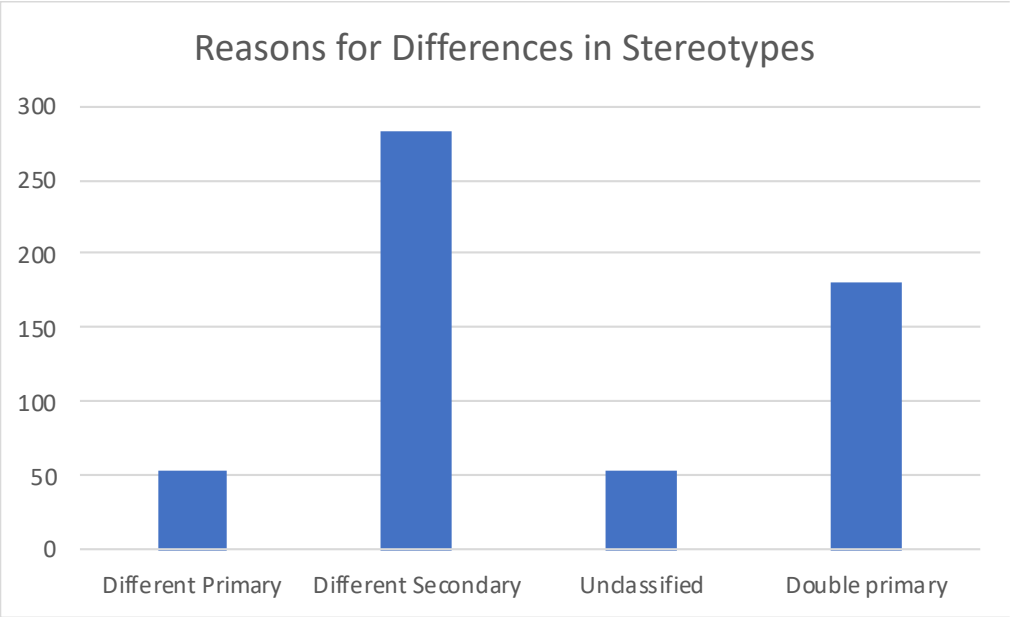


**Figure 8. Reasons for differences in method stereotypes**

Using the research prototype's stereotypes as a baseline, 563 methods are labeled differently and each of these methods were closely examined. All the methods that are differently stereotyped are found to be more accurate than the older version. The differences in stereotypes between the new tool and the old tool are shown above in Figure 8.

Many of the differences are the old tool labeling methods with multiple primary stereotypes and the new version picked one of them. For example, one method the old tool labeled as a property collaborator factory, and the new tool labeled it as a factory collaborator as shown in the figure below. This is due to the old tool labeling all methods with all stereotypes that apply when the new tool labels all methods with only one primary stereotype. This occurred in 180 methods.

```
1   /** @stereotype property collaborator factory */
2   DataSource *
3   LineProjector::
4   createNTuple () const
5   {
6     unsigned int columns = dp::SIZE;
7     NTuple * ntuple = new NTuple ( columns );
8     vector < string > labels;
9
10    labels.push_back ( "X" );
11    labels.push_back ( "Y" );
12    labels.push_back ( "nil" );
13    labels.push_back ( "nil" );
14
15    ntuple -> setLabels ( labels );
16
17    fillProjectedValues ( ntuple );
18
19    return ntuple;
20  }
```

**Figure 9. Double primary method**

Where the two tools deferred the most was the methods' secondary stereotypes. In 282 of the methods, the secondary stereotypes are different. For example, the old tool labels a method command and the new tool correctly labels it command collaborator, as you can see in the figure below. The new tool adds secondary stereotypes in 276 of the methods and does not include a secondary stereotype in seven of the methods. The tool adds and removes secondary stereotypes in one method. Both tools label the methods with the same primary stereotype for all the methods where the secondary stereotype is different. Many of the differences in secondary stereotypes are the result of the tool adding the collaborator

stereotype. Often, this occurs because the new tool is checking for attributes in the header file that are objects. If those attribute objects are used the method, it will be labeled a collaborator.

```
1   /** @stereotype command */
2   void DataRep::setSelected ( bool yes )
3   {
4     m_rep->setSelected ( yes );
5   }
6
7   /** @stereotype predicate */
8   bool
9   DataRep::
10  isSelected () const
11  {
12    return m_rep ->isSelected ();
13  }
14
15  RepBase * m_rep;
```

**Figure 10. Corrected collaborators**

Figure 10 shows two methods that use a data member that is an object. The data member *m_rep* appears in both methods and its declaration from the header file is shown on line 15. *RepBase* is another class in Hippodraw, so this data member is an object. Stereocode correctly labels *setSelected* and *isSelected* with command collaborator and predicate collaborator types respectively.

In 53 of the methods the old tool did not give a stereotype, or the stereotype was given the unclassified label, and 54 methods were labeled with different primary stereotypes. 25 of methods with different primary stereotype were collaborational

command vs command and two were different in property vs collaborational property. This means both tools found the methods to be of a similar structure but could not agree on whether the methods used data members or contained a pure call. Since pure calls are easy to locate the issue is the data members. Examples of this non-agreement on data members and a more accurate primary stereotype are shown below in figures 11 and 12 respectively.

```
1   /** @stereotype collaborational-command */
2   void PyDataRep::makeColorMap() {
3
4      int black[] = {0, 0, 0};
5      s_colorMap["black"] = std::vector<int>(black, black+3);
6
7      int red[] = {255, 0, 0};
8      s_colorMap["red"] = std::vector<int>(red, red+3);
9
10     int green[] = {0, 255, 0};
11     s_colorMap["green"] = std::vector<int>(green, green+3);
12
13     int blue[] = {0, 0, 255};
14     s_colorMap["blue"] = std::vector<int>(blue, blue+3);
15
16     int yellow[] = {255, 255, 0};
17     s_colorMap["yellow"] = std::vector<int>(yellow, yellow+3);
18
19     int cyan[] = {0, 255, 255};
20     s_colorMap["cyan"] = std::vector<int>(cyan, cyan+3);
21
22     int magenta[] = {255, 0, 255};
23     s_colorMap["magenta"] = std::vector<int>(magenta, magenta+3);
24  }
```

**Figure 11. Data member disagreement**

In figure 11, the research prototype does not recognize any data members. However, stereocode looks at the header file and finds the data member *s_colorMap.*

```
1   /** @stereotype set */
2   void Bins1DProfile::accumulate( double x, double y, double, double )
3   {
4     int i = binNumber ( x );
5
6     m_sum[i] += y;
7     m_sumsq[i] += y * y;
8     m_num[i] ++;
9
10    m_empty = false;
11  }
```

**Figure 12. More accurate primary stereotype**

In figure 12, the research prototype does not recognize the data members *m_sum* and *m_sumsq*. This results in the method being incorrectly labeled set instead of command.

Both tools have labeled the same methods with many of the same stereotypes. Only 22% of the methods have different stereotypes, 2% have different primary stereotypes, and of the ones that have different primary stereotypes half of them are only different because the tools disagree on if there are data members are used in the method. The unclassified methods are clearly an improvement because the new tool gave a stereotype when none was given before. The double primary methods are more accurate because each method only has one primary stereotype. The different secondary stereotypes were almost always adding information compared to the old tool. Each of the methods that have different primary stereotypes, and the new tool was found to be more accurate based on the rules described in chapter 3. For these reasons, we believe the new stereocode is an improvement on the old tool.

CHAPTER 5

# Conclusions and Future Work

In this thesis, the design and implementation of stereocode is presented. This chapter includes possible plans for future work and closing statements.

As shown in chapter 2 the stereocode has many uses. Chapter 3 describes the design and implementation of stereocode. Stereocode determines method stereotypes in one or more srcML archives. This is done by using xpath expressions to collect information from the header file (class definition) and implementation file (method definitions). Each method gets assigned a primary stereotype and zero or more secondary types. These are saved in a CSV file and returned in an annotated archive. After testing stereocode against a previous research prototype, the results show that stereocode is more accurate because of the information gathered from the header file. Stereocode will be freely available and open source on GitHub.

## 5.1 Future Work

There are several improvements and features yet to be implemented in the tool. For example, the tool can be tested to support Java and C#. The tool currently does not redocument the code with stereotypes but can be added as an option. Class stereotypes can also be found easily and would be a welcome addition. Another improvement is to collect the data members more accurately. While this tool is a step up from the old version by looking at the header files of the classes, inherited data members are still mostly

assumptions. To solve this, we would need to figure out or be given the inheritance hierarchy. If it is given, we can order the archive files to be processed in such a way that data members from all previous files are saved and able to be found in the subclasses. If the inheritance hierarchy isn't given, then we would need to examine the names of the classes and parent classes from the header files in the preprocessing step. Additional processing is required to figure out which classes' data members can appear in which subclasses.

## 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.

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.

AREVALO, G., DUCASSE, S., AND NIERSTRASZ, O. 2003. XRay Views: Understanding the Internals of Classes. 267–270.

ATKINSON, C., KUHNE, T., AND HENDERSON-SELLERS, B. 2002. Stereotypical Encounters of the Third Kind. 100–114.

CHIDAMBER, S.R. AND KEMERER, C.F. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering 20*, 476–493.

CLARKE, P.J., MALLOY, B.A., AND GIBSON, J.P. 2003. Using a Taxonomy Tool to Identify Changes in OO Software. 213–222.

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., KAGDI, H., AND MALETIC, J.I. 2003. An XML-Based Lightweight C++ Fact Extractor. IEEE-CS, 134–143.

DEITEL, H.M. AND DEITEL, P.J. 2001. *C++ How to Program Third Edition*. Prentice Hall.

DRAGAN, N. 2011. Emergent Laws of Method and Class Stereotypes in Object Oriented Software. *27th IEEE International Conference on Software Maintenance*, 550–555.

DRAGAN, N., COLLARD, M.L., HAMMAD, M., AND MALETIC, M.I. 2011. Using Stereotypes to Help Characterize Commits. IEEE, 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. 2010. Automatic Identification of Class Stereotypes. *IEEE International Conference on Software Maintenance (ICSM'10)*, IEEE, 1–10.

FOWLER, M. 2000. *UML Distilled Third Edition A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley.

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

GENERO, M., CRUZ-LEMUS, J.A., CAIVANO, D., ABRAHÃO, S.M., INSFRÁN, E., AND CARSÍ, J.A. 2008. Does the use of stereotypes improve the comprehension of UML sequence diagrams? 300–302.

GOGOLLA, M. AND HENDERSON-SELLERS, B. 2002. Analysis of UML Stereotypes within the UML Metamodel. 84–99.

LANZA, M. AND DUCASSE, S. 2001. A Categorization of classes based on the visualization of their Internal Structure: the Class Blueprint. ACM Press, 300–311.

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.

MALETIC, J.I., COLLARD, M.L., AND MARCUS, A. 2002. Source code files as structured documents. *10th International Workshop on Program Comprehension*, IEEE, 289–292.

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.

RICCA, F., DI PENTA, M., TORCHIANO, M., TONELLA, P., AND CECCATO, M. 2010. How Developers' Experience and Ability Influence Web Application Comprehension Tasks Supported by UML Stereotypes: A Series of Four Experiments. *IEEE Transactions on Software Engineering 36*, 96–118.

SAVITCH, W. 1999. *Problem Solving with C++ The Object of Programming Second Edition*. .

SHARIF, B. AND MALETIC, J.I. 2009. The Effect of Layout on the Comprehension of UML Class Diagrams: A Controlled Experiment,. *5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'09)*, 11–18.

STARON, M., KUZNIARZ, L., AND WOHLIN, C. 2006. Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments. *Journal of Systems and Software 79*, 727–742.

STROUSTRUP, B. 2000. *The C++ Programming Language*. Addison-Wesley.

TREMBLAY, J.-P. AND CHESTON, G.A. 2001. *Data Structures and Software Development in an Object-Oriented Domain Eiffel Edition*. Prentice Hall.

WEISS, M.A. 1999. *Data Structures & Algorithm Analysis in C++*. Addisson-Wesley.

WORKMAN, D. 2002. A Class and Method Taxonomy for Object-Oriented Programs. *Software Engineering Notes 27*, 53–58.

YUSUF, S., KAGDI, H., AND MALETIC, J.I. 2007. Assessing the Comprehension of UML Diagrams via Eye Tracking. *15th IEEE International Conference on Program Comprehension (ICPC 2007)*, 113–122.