Stereocode: A Tool for Automatic Identification of Method and Class Stereotypes for Software Systems

Ali F. Al-Ramadan
Department of Computer Science
Kent State University
Kent, Ohio, USA
aalramad@kent.edu

Joshua A. C. Behler
Department of Computer Science
Kent State University
Kent, Ohio, USA
jbehler1@kent.edu

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

Natalia Dragan
Department of Information Systems and Business Analytics
Kent State University
Kent, Ohio, USA
ndragan@kent.edu

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

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

***Abstract*— We present *Stereocode*, a static analysis tool engineered to automatically identify, and re-document software systems written in C++, C#, and/or Java with method and class stereotypes. A stereotype is a simple abstraction that encapsulates the high-level behavior of a method or a class. The tool is built around the srcML infrastructure, an XML representation of source code. *Stereocode* annotates the srcML input with the computed stereotypes as XML attributes to the function and class tags. We showcase *Stereocode*'s efficiency in conducting large-scale analysis of software systems, which involves using 1050 repositories from GitHub across C++, C#, and Java. The results provide valuable insights into the distribution of stereotypes. A demo video is available at: https://youtu.be/D9oxwUlPbOI.**

Keywords—Stereocode, srcML, method stereotypes, class stereotypes, static analysis

I. Introduction

Software systems are becoming increasingly complex and understanding their design is critical for maintenance and evolution. Method and class stereotypes [1–3] have emerged as powerful abstractions in the context of class design. A stereotype is a concise statement that captures the intrinsic atomic behavior of a method or a class at a much lower level than high-level abstractions such as design patterns. An example of a method stereotype is an accessor (e.g., getter) or a mutator (e.g., setter) to indicate a method that gets or sets a data member in an object. Similarly, common examples of class stereotypes are boundary, entity, and controller.

Stereotype information has been shown to be useful in a variety of applications related to program comprehension, documentation, and maintenance activities [4–14]. Despite their importance, there is a lack of an accurate and usable tool that can automatically identify stereotype information from source code at scale. Previous research relied on tools with very simplistic analysis techniques to identify stereotypes in C++ or Java systems. Furthermore, manually documenting and maintaining stereotype information in large software is costly.

To address these challenges, we introduce a public release of *Stereocode*. This tool is designed to reliably and accurately compute stereotypes for software systems written in C++, C#, and Java (or a combination of these languages), at a large scale by leveraging the srcML infrastructure format [15, 16] allowing for direct access to the syntactic information to support static analysis. *Stereocode* begins by extracting detailed program information (e.g., classes, methods, data members, etc.) from the srcML input. This information is used to build a complete symbol table for the entire software system. Once the symbol table is constructed, *Stereocode* then uses it to identify stereotypes. First, the stereotype of each method is computed using a set of predefined rules [1]. Then, the stereotypes of classes are identified using the frequency distribution of the computed method stereotypes [3]. Some of the features that *Stereocode* offers include:

- The ability to identify stereotypes for both individual source files and complete software systems.
- Supports the identification of stereotypes for C++, C#, and/or Java programming languages.
- Provides output in various formats.

Stereocode can contribute to the field of software engineering by providing a more effective tool for software analysis and design recovery. The tool is publicly available at https://github.com/srcML/Stereocode and licensed under GPL3.

II. Related Work

The initial version of a stereotyping tool [1–3, 17] was developed as a research prototype to support the automatic identification of method and class stereotypes for C++ systems. This tool applies the stereotype rules as transformation in XSLT to the srcML input of the source code. The resulting stereotypes are applied to the function and class tags in the header files. However, this version of *Stereocode* does not account for all information in the class (e.g., inherited data) and only works on C++ systems. Another similar tool, *JStereocode*, was introduced

© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. This is the accepted version of the paper published in the Proceedings of the 2024 IEEE International Conference on Software Maintenance and Evolution (ICSME), DOI 10.1109/ICSME58944.2024.00099.

by Moreno and Marcus [18]. This tool worked as an Eclipse plug-in and supports the identification of method and class stereotypes for Java-based systems. Nevertheless, *JStereoCode* also has numerous limitations. It only supports the Java programming language and is difficult to use or install. Other research has explored the use of machine learning to automatically classify class stereotypes in Java [19, 20], and to examine the development of class role stereotypes and anti-patterns in software systems [21]. These approaches often struggle to accurately detect stereotypes, especially less common ones.

Stereocode differentiates itself from prior work in several ways. The approach employed is not limited to a single language, and it supports a more comprehensive, efficient, and accurate stereotyping for both methods and classes at a large scale covering C++, C#, and/or Java systems. A key strength of the new *Stereocode* is its ability to build a comprehensive symbol table that captures detailed information about all classes and their relationships. This in-depth static analysis provides a far more complete picture of the whole system, something that previous work did not consider.

III. Architecture

Stereocode is built using the srcML[1] infrastructure [15, 16]. srcML is a robust and highly scalable infrastructure used for transforming source code into a structured XML representation without any loss of lexical information. srcML provides access to this information to support a wide range of tasks, including analysis, exploration, and manipulation of source code. *Stereocode* utilizes the srcML format to extract detailed information from the source code needed to compute stereotypes.

Stereotyping using *Stereocode* involves three primary steps: collection of program information using static analysis, stereotype identification using a rule engine, and stereotype annotation (as shown in Fig. 1). We cover these activities in detail in the following three subsections.

A. Collection of Program Information

The stereotyping process begins by scanning the units (i.e., source files) in the srcML format input to collect all the classes. For each class, the class name, parent class names, data members, specifiers, and methods are gathered and stored for further analysis. For every method, information, including the method name, return type, parameters, specifiers, local variables, return expressions, and calls (including method, function, and constructor calls), are collected. The tool utilizes various XPath expressions to collect the necessary information from the input. Moreover, *Stereocode* can detect the language of each unit to handle language-specific information correctly, such as friend functions (C++), partial classes (C#), properties (C#), structs (C# and C++), interfaces (C# and Java), enums (Java), and unions (C++), among others. In addition, *Stereocode* maintains a list of language-specific primitives (e.g., `int`, `double`, etc.) to distinguish between primitive and non-primitive datatypes. Users can also provide a list of user-defined primitives if desired.

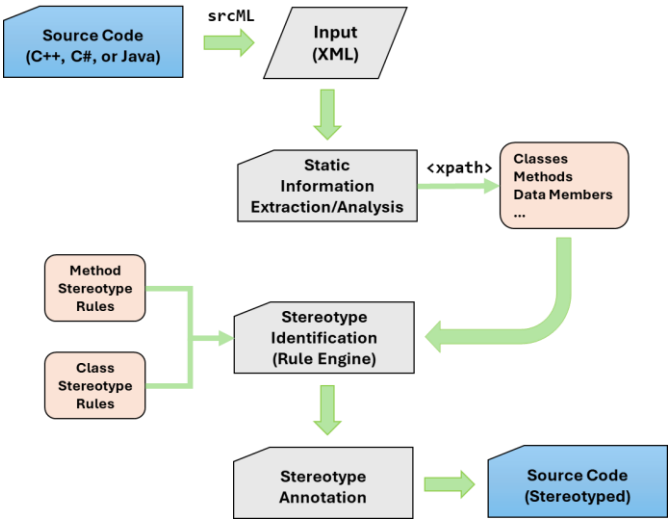


Fig 1. Overview of the stereotyping process using *Stereocode*

B. Stereotype Identification

After collection, the information is statically analyzed by *Stereocode* to derive the necessary data needed to identify stereotypes. This includes determining whether the method uses, changes, or returns data members, local variables, or parameters. In addition, *Stereocode* also checks if the method is empty, is constant, i.e., `const` (C++ only), and whether it modifies parameters passed by reference. Furthermore, *Stereocode* filters all calls to determine stateless calls (i.e., calls that do not read/change the state of the object). A list of calls (e.g., `assert, println`) is provided to ignore certain calls from the analysis, and users have the option to expand this list. Method stereotypes are then identified by directly mapping the extracted data to a list of predefined rules. To illustrate, a method is classified as a *get* if it contains at least one return of a data member (e.g., `return dm;`), where `dm` is a data member of the class. Class stereotypes are then identified from the ratio of the totals of method stereotypes

TABLE I and TABLE II show the taxonomies for method and class stereotypes identified by *Stereocode* as introduced by Dragan et al. [1–3]. The project's GitHub wiki page details the definitions and rules used. Method stereotypes (TABLE I) are separated into five distinct categories, and each category includes a set of specific stereotypes that indicate a finer-grained definition of a method's basic behavior. For example, mutator stereotypes modify an object's state, but the stereotype *set* specifically indicates a modification of a single data member, while the *command* stereotype performs a more complex change (e.g., modifying multiple data members). Methods and classes may be labeled with one or more stereotypes. For example, a *predicate collaborator* is a predicate method that uses an object of another class.

C. Stereotype Annotation

Stereocode generates the stereotyped output by re-annotating the srcML input with the identified stereotypes. These stereotypes are incorporated as XML attributes into the

[1] See www.srcML.org

respective function and class tags in srcML. Fig. 2 illustrates a basic getter method represented in the srcML format with its stereotype annotated as an attribute in a function tag. Additionally, *Stereocode* has an option to annotate the stereotypes as a comment inserted before the function or class definition, as shown in Fig. 3. The first method, `GetValueRaw`, is stereotyped as *get collaborator* since it returns a data member `value` of non-primitive type, and the second method `setModule` is stereotyped as *set collaborator* since it modifies a single data member `Module` of a non-primitive type. The class `PSVariable` is stereotyped as a *boundary* using the distribution of stereotypes of all its methods. Moreover, *Stereocode* can generate optional report files in various formats, including `.csv` and `.txt`.

TABLE I. TAXONOMY OF METHOD STEREOTYPES

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, data member, or return value)
	controller	Changes only an external object's state (not this)
	wrapper	Does not change an object's state. Has at least one free function call
Degenerate	incidental	Does not read/change an object's state. No calls to other class methods or to free functions
	stateless	Does not read/change an object's state. Has at least one call to other class methods or to a free function
	empty	Has no statements

```
<function st:stereotype="get"><type><name>int</name></type>
<name>getID</name><parameter_list>()</parameter_list> <block>{<block_content>
<return>return<expr><name>id_</name></expr>;</return>
</block_content>}</block></function>
```

Fig 2. Example of getter method with stereotype (highlighted) inserted as an attribute in the srcML format.

TABLE II. TAXONOMY OF CLASS 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
commander	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
controller	(aka control) Provides functionality to control external objects. Consists mostly of controller and factory methods
pure-controller	A special case of controller. 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 degenerate 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

```
// @stereotype boundary
public class PSVariable :
IHasSessionStateEntryVisibility {
    ...
    public PSModuleInfo Module
    { get; private set; }
    private object _value;

    // @stereotype get collaborator
    internal virtual object GetValueRaw()
    { return _value; }

    // @stereotype set collaborator
    internal void SetModule(PSModuleInfo module)
    { Module = module; }
    ...
}
```

Fig 3. Example of a C# class PSVariable from PowerShell with stereotype information annotated as a comment.

IV. Demonstration and Scalability

To demonstrate the usefulness and scalability of *Stereocode*, we apply the tool to determine the stereotype information across a large number of open-source software systems. We collected 1,050 repositories from GitHub, with 350 each for C#, C++, and Java, to ensure a variety of systems with different sizes and domains. The selection of the repositories is based on their popularity determined by their star ranking. We collect only the default branch (e.g., main or master) for each repository. Repositories that are empty or archived are excluded from the selection. We then convert each system into the srcML format

using *srcml*. For most systems, the conversion process takes less than a second (on a typical laptop). Following this, *Stereocode* generates stereotype information for each system with most systems finishing in under six seconds. It is worth noting that structs, interfaces, enums, and unions are also stereotyped to provide a more complete picture of each system's design. In addition, we stereotype the entire system, including all test files. We do not currently stereotype static methods, static classes (C# and Java), or free functions (C++) (i.e., functions defined outside of classes or structs) as they are not covered in the current stereotype taxonomy. The entire process is automated using a Python script.

The stereotyped data is stored in an artifact[2] that offers a complete view of the stereotype distributions across the analyzed software systems. This includes a breakdown of individual stereotypes (e.g., *get*, *set*, etc.) and unique stereotypes (e.g., *command collaborator*) for both methods and classes.

Fig. 5 presents a chart showing a concise overview of the prevalence of different method stereotypes within all the stereotyped systems organized by language. We observe a large number of *collaborators*, *commands*, and *properties* in all systems. This suggests that these systems heavily rely on methods that provide access to information derived from data members (property), interact with other classes (collaborators), or perform a complex change the state of the system (commands).We also observe a significant number of *stateless* methods in all three languages. Many of these *stateless* methods indicate the existence of utility methods that carry out general tasks (e.g., unit testing or providing an external service) without altering the state of the object. It is worth noting that there are no destructors in Java as in C# and C++. The *unclassified* stereotype in Fig. 5 refers to any method that did not satisfy any of the method stereotypes shown in TABLE I.

This information can be leveraged as building blocks for more advanced forms of code analysis and knowledge extraction. For instance, consider a method from the stereotyped data that is stereotyped as a *property set*, as shown in Fig. 4. The method changes the value of `seed`, a data member of the object. It then creates and returns a local variable `refId` using information about data members (i.e., `seed`). At first glance, this label might seem counterintuitive as the method is both an accessor and a mutator. However, this explicit labeling provides a clearer understanding of the method's behavior, in which case, it could be a possible code smell [4]. In other words, the method's behavior of modifying a data member and returning a property might be an indicator of poor design since the method is doing more than it should.

```
// @stereotype property set
private UInt64 GetNewReferenceId() {
   UInt64 refId = seed++;
   return refId;
}
```

Fig 4. Potential code smell indicated by both property and set stereotypes in a method from the PowerShell system.

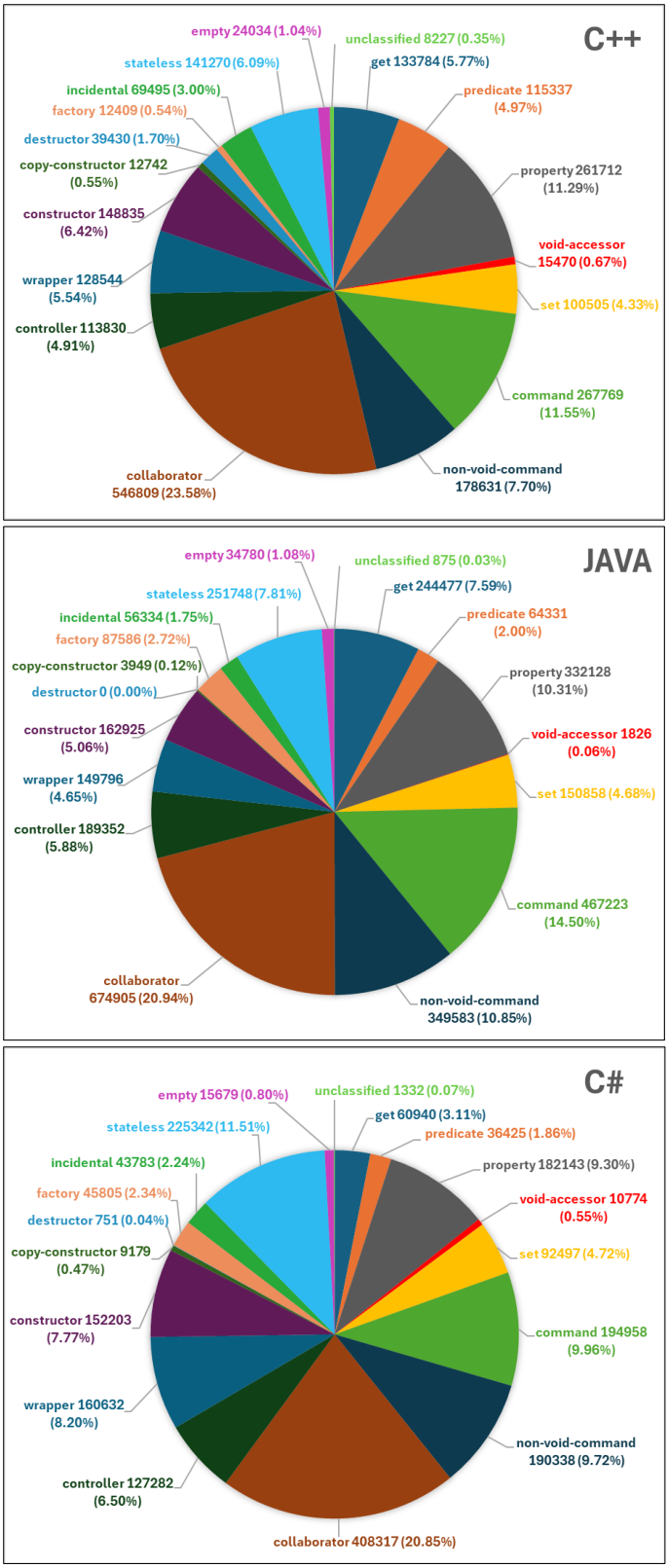


Fig 5. Distribution of method stereotypes for all systems used in the study. The systems are divided in to the C++, C#, and Java. There are 1050 total systems and 350 each for C++, C#, and Java.

[2] See https://github.com/KSU-SDML/Stereocode-artifact

V. Running Stereocode

Stereocode is a command-line tool that runs on the srcML format of source code (see www.srcML.org). Initially, the input (either an individual source code file or an entire software system) is converted to srcML using the *srcML* command-line client. Then, *Stereocode* is run on that srcML:

```
srcml main.cpp -o main.cpp.xml
stereocode main.cpp.xml -o stereotypes.cpp.xml
```

There are downloadable executables for *srcML* at the website. *Stereocode* needs to be compiled. Instruction to compile and run are in the GitHub repo for the project (see README). It is developed in C++ and built via CMake. *Stereocode* runs on Mac, Windows, and Linux.

VI. Conclusion and Future Work

Stereocode is a tool designed to automatically identify method and class stereotypes in software systems across C++, C#, and/or Java. It uses static analysis to do this automatically and accurately. The tool assists developers to understand the roles and behaviors of methods and classes in software systems. We demonstrate *Stereocode* on 1,050 repositories from GitHub. The results show that *Stereocode* can effectively analyze software systems and is very scalable. In the future, we plan to improve *Stereocode* by adding support for more programming languages (e.g., Python). In addition, we plan to use *Stereocode* to explore the stereotypes of free and static functions to provide a more complete and holistic picture of the software system. Other future work includes handling classes with the same name, and adding proper support for nested structures (e.g., nested classes or nested functions).

Acknowledgment

This work was supported in part by a grant from the US National Science Foundation: CNS 20-16465/16452

References

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

[2] N. Dragan, M. L. Collard, and J. I. Maletic, "Using Method Stereotype Distribution as a Signature Descriptor for Software Systems," in *25th IEEE International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 567–570.

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

[4] M. J. Decker, C. D. Newman, N. Dragan, M. L. Collard, J. I. Maletic, and N. A. Kraft, "Which Method-Stereotype Changes are Indicators of Code Smells?," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'18)*, 2018, pp. 82–91.

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

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

[7] O. Andriyevska, N. Dragan, B. Simoes, and J. I. Maletic, "Evaluating UML Class Diagram Layout based on Architectural Importance," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, 2005, pp. 14–20.

[8] N. Dragan, M. L. Collard, M. Hammad, and J. I. Maletic, "Categorizing Commits Based on Method Stereotypes," presented at the 27th IEEE International Conference on Software Maintenance (ICSM'11), 2011, pp. 520–523.

[9] L. F. Cortes-Coy, M. Linares-Vasquez, J. Aponte, and D. Poshyvanyk, "On Automatically Generating Commit Messages via Summarization of Source Code Changes," presented at the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'14), 2014, pp. 275–284.

[10] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, "On Automatic Summarization of What and Why Information in Source Code Changes," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, 2016, vol. 1, pp. 103–112.

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

[12] B. Li, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Aiding comprehension of unit test cases and test suites with stereotype-based tagging," in *Proceedings of the 26th Conference on Program Comprehension*, New York, NY, USA, 2018, pp. 52–63.

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

[14] R. Gökmen, D. Heidrich, A. Schreiber, and C. Bichlmeier, "Stereotypes as Design Patterns for Serious Games to Enhance Software Comprehension," in *2021 IEEE Conference on Games (CoG)*, 2021, pp. 1–3.

[15] M. L. Collard, M. J. Decker, and J. I. Maletic, "Lightweight Transformation and Fact Extraction with the srcML Toolkit," in *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, pp. 173–184.

[16] M. L. Collard, M. J. Decker, and J. I. Maletic, "srcML: An Infrastructure for the Exploration, Analysis, and Manipulation of Source Code: A Tool Demonstration," in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, 2013, pp. 516–519.

[17] D. Guarnera, M. L. Collard, N. Dragan, J. I. Maletic, C. Newman, and M. Decker, "Automatically redocumenting source code with method and class stereotypes," in *2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3)*, 2018, pp. 3–4.

[18] L. Moreno and A. Marcus, "JStereoCode: automatically identifying method and class stereotypes in Java code," in *27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, 2012, pp. 358–361.

[19] A. Nurwidyantoro, T. Ho-Quang, and M. R. V. Chaudron, "Automated Classification of Class Role-Stereotypes via Machine Learning," in *Proceedings of the 23rd International Conference on Evaluation and Assessment in Software Engineering*, New York, NY, USA, 2019, pp. 79–88.

[20] T. Ho-Quang, A. Nurwidyantoro, S. A. Rukmono, M. R. Chaudron, F. Fröding, and D. N. Ngoc, "Role stereotypes in software designs and their evolution," *J. Syst. Softw.*, vol. 189, p. 111296, 2022.

[21] D. Nguyen Ngoc and F. Fröding, "The Evolution of Role-Stereotypes and Related Design (Anti) Patterns," 2020.