Using Method Stereotype Distribution as a Signature Descriptor for Software Systems

Natalia Dragan, Michael L. Collard, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, Ohio 44242
{ndragan, collard, jmaletic}@cs.kent.edu

Abstract

Method stereotype distribution is used as a signature for software systems. The stereotype for each method is determined using a presented taxonomy. The counts of the different stereotypes form a signature of the system. Determining method stereotypes is done automatically and is based on language (C++) features, idioms, and the main role (purpose) of a method. The intent is to use the distribution of method stereotype is an indicator of system architecture.

1. Introduction

Previously, we demonstrated the ability to automatically, and efficiently, reverse engineer method stereotypes [1] from object oriented source code (specifically C++). In practice, methods are rarely documented with stereotypes (e.g., get, set, predicate, etc), yet we feel this information can be used to help infer the context of a class and how classes interact. That is, good method abstraction is typically a requirement for good class abstraction. Our hypothesis is that we need to understand the methods before we can infer the role and design of a class.

Knowing the method stereotypes will support sophisticated types of design recovery and form a foundation for a range of approaches based on method stereotypes. For example, metrics based on method stereotype have a much more fine-grained perspective and include structural information with little cost. Also changes to method stereotypes may indicate major design changes to a class or system.

Towards verifying our hypothesis we investigate the relative frequency of particular method stereotypes within a system. This distribution forms a signature of a system.

Before a systematic study of different systems can be undertaken we refined our previous method stereotype taxonomy based on a broader examination of open source systems. Over 20 open source systems were examined and method stereotypes for each were automatically computed. We then determined if any large numbers of methods went unclassified or represented categories not in our original taxonomy. We present this refined taxonomy and an example signature for an open source systems.

In the next section a summary description of our method stereotype taxonomy is given. This represents the main result of our previous work [1] with a few minor extensions that reflect the larger breadth of systems examined for this work. Following that is an example system signature and then related work and conclusions.

2. Method-Stereotype Taxonomy

The taxonomy of method stereotypes is organized by the main purpose and role of an object-oriented method while simultaneously emphasizing its creational, structural, and collaborational aspects with respect to a class's design. Creational methods are responsible for creating or destroying objects of the class. Structural methods provide and support the structure of the class. Collaborational methods define the communication between objects and how objects are controlled in the system. The first version of the taxonomy presented in [1] was extended to include a four small, however important, categories that are indicators of particular design issues. The new category Degenerate is now included along with additional Structural and Collaborational stereotypes namely, void-accessor, non-void-command, and controller. The overview of the method stereotype taxonomy is given in Table 1. We now give a brief description of new stereotypes (in italics in Table 1) focusing on primary stereotypes first, then secondary ones. For additional details and examples we point the reader to [1].

The stereotypes in the categories Accessor, Mutator, and Creational are termed *primary stereotypes*. A method can have only a single primary stereotype.

Table 1. A taxonomy of method stereotypes. Methods have a single primary stereotype from any category and may have secondary stereotypes from the categories Collaborational and Degenerate.

Structural		Creational	Collaborational	Degenerate
Accessor	Mutator			
Get	Set	Constructor	Collaborator	*Incidental*
Predicate	Command	Copy-Constructor	*Controller*	*Empty*
Property	*Non-void-Command*	Destructor		
Void-Accessor		Factory		

Accessors are methods that do not change an object state (i.e., `const` specifier in C++). The new proposed stereotype in the accessor category is:

- Accessor::Void-Accessor returns some information about data members through a parameter (method's return type is void).

Mutators are methods that change the object state. The differences between stereotypes in this category reflect how and by how much the state is changed. The new mutator stereotype is:

- Mutator::Non-void-Command performs a complex change to the object's state and returns a value (i.e., is not void) of a non-boolean type. In the code we check the return type, and look for assignments to data members.

Creational methods include the stereotypes Constructor, Copy-Constructor, and Destructor that match the standard C++ language features. We restrict the consideration of creational methods to the factory method because constructor, copy constructor, and destructor methods are well-known and are fairly easy and straightforward to identify. In fact most languages have specific syntax for these special-purpose methods and C++ is no exception. The remaining stereotype Creational::Factory returns an object created in the method's body.

A method may also have a *secondary stereotype* in the category Collaborational or Degenerate. This allows multiple stereotypes to be assigned to a single method, e.g., property-collaborator or predicate-incidental. However, both Collaborational and Degenerate can also be just the single primary stereotype of a method.

Collaborational methods work on an external object (of a different type) that is either a parameter or a local variable. The new collaborational stereotype is:

- Collaborational::Controller does not read/write to the object's state, i.e., works only on objects different from itself. This is a primary stereotype.

Degenerate are methods where the primary stereotypes are limited. The name is based on the mathematical term for a limiting case for which a stereotype cannot be any simpler. These include the following stereotypes:

- Degenerate::Incidental does not read or change an object's state directly nor indirectly: it is an utility, an exception handler, or a candidate for overriding. This is a secondary or primary stereotype.
- Degenerate::Empty has no statements at all and perhaps is created with the eventual goal of overriding. This is a secondary or primary stereotype.

In the next section we present how the percentage occurrences of stereotypes in a system form a signature of that system.

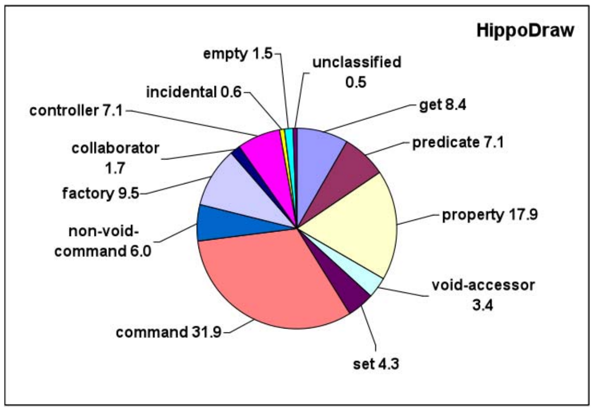

Figure 1. A signature by the method-level perspective of the system HippoDraw.

3. The System Signature

The distribution of method stereotypes represents a *signature* of a system. It describes the degree of prevalence of static structure such as collaboration and state change. We will first describe our tool to automatically extract a signature, and then present an example of what the signature represents.

A system signature is obtained using our tool, *StereoCode* [1], which automatically identifies method stereotypes using lightweight static analysis methods and an infrastructure based on srcML[1] (SouRce Code

[1] Pronounced source M L.

Markup Language) [2] an XML representation that supports both document and data views of source code. A very usable and efficient tool to translate C/C++ to/from srcML is freely available[2]. The srcML format, combined with standard XML tools, has been successfully used for querying and fact extraction of source and transformation (refactoring).

The automatic detection of the stereotypes is based on the analysis of the source code in the srcML format. For each stereotype, an XPath expression is used to detect that particular pattern. For every method, each stereotype XPath expression is executed on the code. This allows for the detection of secondary stereotypes, and verified that the XPath expressions are distinct, i.e., that each method has a single primary stereotype.

StereoCode re-documents the original source code with the stereotypes with a special `@stereotype` tag in the comments. For system-wide totals, these stereotype comment tags are collected and totaled. The entire process from the original source code, to srcML, to the stereotype totals only takes a few minutes, even for large systems.

We now examine the frequency of each different stereotype that occurs within a system. The distribution of these frequencies forms a signature of the system. As an example we present the system signatures from the C++ software system HippoDraw, an open-source application. The application includes data-analysis and visualization with a GUI interface. Version 1.21.3 contains approximately 96 KLOC of source code in 692 files with 3315 methods. We automatically determined the stereotype of each method using the StereoCode tool. Extracting the stereotypes took under 15 seconds.

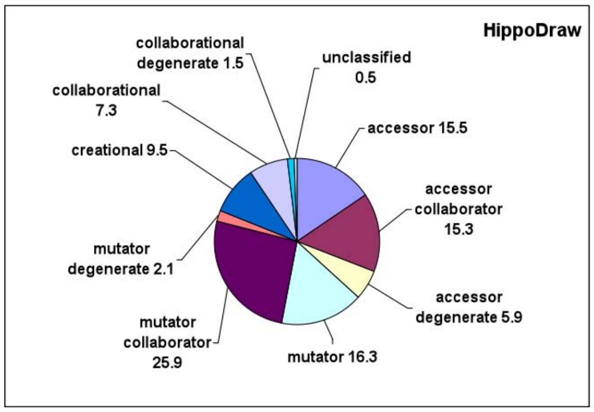

Figure 2. A signature by the class-level perspective of the system *HippoDraw*.

To better interpret the data visually we organize it at two different levels to highlight the logical structure of the code i.e., the method and class levels. At the method level we grouped by primary stereotype. This reflects the role of a method as it ignores, to a large degree, interaction with other classes.

The alternative class level perspective highlights the degree of coupling and collaboration among classes in a system, along with some internal coupling (cohesion) of a class through accessors/mutators. Additionally, parts of the system not yet implemented (degenerate) are reflected. These two different perspectives complement each other and highlight different aspects of a system's design/architecture. That is, it presents a view of the method alone and a view of how the methods collaborate inter- and intra-class. These two perspectives (or slices of the data) are now discussed in more detail.

3.1. Method-level Perspective

First we examine the distribution at the method level according to primary stereotypes: get, predicate, property, void-accessor, set, command, non-void-command, factory, collaborator, controller, incidental, and empty. The methods with secondary stereotypes are not counted separately but included in the count of the primary stereotype (e.g., get-collaborator is counted only once under get).

As an example the distribution is given in Figure 1. As can be seen the system has significant percentage of command, property, factory, and get methods. This example also shows that the method stereotypes added in this work occur in a non-trivial 17.2% of the signature. The system also exhibits some non-standard programming practices. Mutators typically return a type of `void` or `bool`, however the stereotype non-void-command at 6.0% indicates mutators that are returning a type besides `void/bool.` Accessors typically return a value, however the stereotype void-accessor at 3.4% indicates accessors that are returning values through a reference parameter instead. Additionally, the factory stereotype occurs a significant number of times as it is used in almost 10% of the methods.

3.2. Class-level Perspective

The class level perspective involves organizing the data by main stereotype categories combined with secondary stereotype. This list includes the categories accessor, mutator, creational, and collaborational, along with secondary stereotype categories such as accessor-collaborator, accessor-degenerate, etc. This perspective reveals the prevalence of reads and writes to an object state. It also highlights interaction with other classes by inspecting collaborational versus non-collaborational methods within a system.

An example of the class level perspective is given in Figure 2. In this system, accessors, mutators, and external collaborators (factory, collaborator and

[2] See www.sdml.info for translator tool download.

controller) are evenly distributed. We characterize this system as an *Accessor-Mutator-Controller*. The term controller is used because external collaborators control the behavior of external classes. There is also an almost equal distribution of collaborational subcategories compared to non-collaborational.

4. Related work

Analysis of software with respect to architectural/design patterns on the coarse- and fine-grained levels such as method, class, and package, and the evolution of these patterns have been investigated by a number of researchers [3], [4], [5], [6], [7], [8], [9], [10], [11].

Analysis of design patterns at the method-level is performed in [6] and [7]. Arevalo et al. [6] group methods based on the state usage (state access), external/internal calls (self and super calls), and behavioral skeleton (client access) using concept analysis. Workman presents [7] a method taxonomy for Java for class categorization to detect plagiarism.

Class-level design patterns are presented in [4], [5], and [10]. Gil et al. [4] introduce micro patterns, i.e., class-level traceable patterns for Java code with the eventual goal of design assessment. To quickly grasp the purpose of a class and its inner structure in a foreign system, Lanza et al. [5] also consider class-level patterns and categorization of classes based on different class blueprints, i.e., visual representation of the class as a set of four method layers: initialization, interface, implementation and accessor, and an attribute layer. Clarke et al. [10] present a taxonomy of classes to identify changes in object-oriented software based on generalization relationships and the types of data associated with the class. The work presented by Dong et al is the closest to our work in terms of the granularity level [11], [9]. They present a hybrid model reverse engineered at a coarse-grained level, such as package diagrams, and identify architectural change patterns during software evolution.

To the best of our knowledge analysis of design patterns on the system-level and system categorization/classification according to architectural categories has not yet been performed and investigated extensively. Additionally, we examine systems from the perspective of behavioral and control characteristics and their functionality.

5. Conclusions and Future Work

Our preliminary results show that the frequency and distribution, across a system, of the method stereotypes described by our taxonomy, could be an indicator of system architecture/design.

We plan to continue this work by studying a wide range of open-source C++ systems and cluster systems with similar architecture/design. The intent is to classify systems based only on their signature. Additionally, we plan to examine the change of signatures over the evolution of a system. This will investigate whether the signature of a system changes over the development of a project, and when the signature becomes stable.

6. References

[1] N. Dragan, M. L. Collard, and J. I. Maletic, "Reverse Engineering Method Stereotypes," presented at 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, 2006.

[2] M. L. Collard, J. I. Maletic, and A. Marcus, "Supporting Document and Data Views of Source Code," presented at ACM Symposium on Document Engineering (DocEng'02), McLean VA, 2002.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Addison-Wesley, 1995.

[4] J. Gil and I. Maman, "Micro Patterns in Java Code," presented at Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05), San-Diego, California USA, 2005.

[5] M. Lanza and S. Ducasse, "A Categorization of classes based on the visualization of their Internal Structure: the Class Blueprint," presented at 16th ACM Conference on Object-Oriented Programming, Systems. Languages and Applications (OOPSLA ' 01), 2001.

[6] G. Arevalo, S. Ducasse, and O. Nierstrasz, "Understanding Classes using X-Ray Views," presented at 2nd. International Workshop on MASPEGHI 2003 (MAnaging SPEcialization/Generalization HIerarchies) in ASE 2003, 2003.

[7] D. Workman, "A Class and Method Taxonomy for Object-Oriented Programs," Software Engineering Notes, vol. 27, pp. 53-58, 2002.

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

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

[10] P. J. Clarke, B. A. Malloy, and J. P. Gibson, "Using a Taxonomy Tool to Identify Changes in OO Software," presented at 7th European Conference on Software Maintenance and Reengineering, 2003.

[11] X. Dong and M. W. Godfrey, "A Hybrid Program Model for Object-Oriented Reverse Engineering," presented at IEEE International Conference on Program Comprehension, Banff, AB, Canada, 2007.