Automatic Identification of Class Stereotypes

Natalia Dragan
Department of Computer Science
Kent State University
Kent, OH 44242
ndragan@cs.kent.edu

Michael L. Collard
Department of Computer Science
The University of Akron
Akron, OH 44325
collard@uakron.edu

Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent, OH 44242
jmaletic@cs.kent.edu

***Abstract*—An approach is presented to automatically determine a class's stereotype. The stereotype is based on the frequency and distribution of method stereotypes in the class. Method stereotypes are automatically determined using a defined taxonomy given in previous work. The stereotypes, boundary, control and entity are used as a basis but refined based on an empirical investigation of 21 systems. A number of heuristics, derived from empirical evidence, are used to determine a class's stereotype. For example, the prominence of certain types of methods can indicate a class's main role. The approach is applied to five open source systems and evaluated. The results show that 95% of the classes are stereotyped by the approach. Additionally, developers (via manual inspection) agreed with the approach's results.**

Keywords-method stereotypes, class stereotypes, program comprehension, reverse engineering

I. INTRODUCTION

The work presented here investigates how to automatically identify a class's stereotype in an existing object-oriented software system. Stereotypes are a simple abstraction of a class's role and responsibility in a system's design. Very few software systems have this sort of documentation explicit in the source code. Manually documenting this type of abstraction is relatively easy for a small number of classes but doing so for entire systems would be quite costly.

Accurate information about a class's stereotype is useful for a number of software maintenance and evolution tasks. Knowing a class's stereotype implies the role of the class in the design. It gives clues to how a class collaborates with other classes in different design patterns. A class's stereotype may also be an indicator of bad smells and give clues for refactoring. It can be an indicator of comprehensibility.

A number of studies [1-6] demonstrate the benefits of using stereotypes, which reflect semantics, in program comprehension, design, and software maintenance tasks. Using class stereotype information [4-6] as a factor in laying out UML class diagrams has shown to improve the comprehensibility of the diagram. Staron et al. [1] show the effectiveness of class stereotypes based on domain model in program comprehension.

Hence, we feel this is a very important, yet unexamined area of object oriented design recovery. This work directly leverages our prior work on recovery of method stereotypes. However, automatic identification of class stereotypes proved to be a much more difficult problem, requiring a more in-depth empirical study and understanding of how method stereotypes are used across systems and classes.

This work has the following contributions. First, a taxonomy of class stereotypes is proposed. This taxonomy is derived from an empirical examination of 21 open source software systems. The second contribution involves an approach to automatically label a given class with its corresponding derived stereotype. Here we limit our study to one programming language, namely C++.

Our approach starts by automatically identifying and labeling all methods in a system with their stereotype. This information is then collected and a distribution of method stereotypes for each class is calculated. Class stereotypes are derived from this distribution via a set of rules that map method stereotype distribution characteristics to the class stereotype taxonomy. The approach is evaluated against human experts and through an empirical study.

The paper is organized as follows. The next section (II) contains a brief description of our method stereotype taxonomy. This represents the main result of our previous work [7, 8]. Additionally, how we compute the method stereotype distributions is described. The result is a class signature that forms the input for our automatic classification scheme. In section III we present a taxonomy of class stereotypes. Section IV describes our approach to automatically identify class stereotypes from existing C++ code. Section V is an evaluation of the approach as compared to experts, followed by an empirical study in Section VI. This is followed by a discussion of the threats to validity, related work and conclusions.

II. CLASS SIGNATURES

Here we define a *class signature* [8] as a frequency distribution of method stereotypes for a class. We use the class signature to infer a class's stereotype. In this section, we summarize our previous work on defining and automatically identifying method stereotypes as this forms the basis for the signature. Specifics of the class signature are then presented.

A. Method Stereotypes

The aggregates for class signature identification, method stereotypes (see Table I), were first presented in [7] and we

refer the reader there for more complete details and examples. That work presented an approach to automatically identify method stereotypes. Additionally, a taxonomy for method stereotypes is given that unifies and extends the previous literature on stereotypes to address a number of gaps and deficiencies. Based on this taxonomy, method stereotypes can be reverse engineered using static program analysis. We constructed a tool, *StereoCode*, that re-documents source code with the stereotype information for each method. The assessment of this work demonstrated two things. First, that the given method stereotype classification covered a very large percentage of the methods studied. That is, almost all methods could be labeled by the classification scheme. Second, that the tool redocumented systems correctly. By correctly, we mean that an experienced developer agreed 90% of the time with our labeling of the method. The discrepancies typically involved very poorly written or convoluted methods.

The taxonomy of method stereotypes (Table I) is organized by the main role of a method while simultaneously emphasizing its creational, structural, and collaborational aspects with respect to a class's design. *Structural* methods provide and support the structure of the class. For example, accessors read an object's state, while mutators change it. *Creational* methods create or destroy objects of the class. *Collaborational* methods characterize the communication between objects and how objects are controlled in the system. *Degenerate* are methods where the structural or collaborational stereotypes are limited. The name is based on the mathematical term for a case for which a stereotype cannot be any simpler.

TABLE I. TAXONOMY OF METHOD STEREOTYPES

Stereotype Category	Stereotype	Description
Structural ***Accessor***	get	Returns a data member.
	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.
	command	Performs a complex change to the object's state.
	non-void-command	
Creational	constructor, copy-const, destructor, factory	Creates and/or destroys objects.
Collaborational	collaborator	Works with objects (parameter, local variable and return object).
	controller	Changes only an external object's state (not *this*).
Degenerate	incidental	Does not read/change the object's state.
	empty	Has no statements.

Also, a method may have more than one stereotype. Methods have a single stereotype from any category and may have secondary stereotypes from the collaborational and degenerate categories. For example, a two-stereotype method *get collaborator* returns a data member that is an object or uses an object as a parameter or a local variable. We now describe how the method stereotypes are used for defining the class signatures.

B. *Method Stereotype Distributions*

In [8] we introduced the idea of system signatures and examined the frequency of method distributions for one open source system. From this study we learned that these distributions of method stereotypes seemed to be indicators of system architecture. Here we extend this concept to a class signature.

We found it useful to present the distribution data in both a detailed and summarized manner. In the detailed view we give the distribution counts for each individual stereotype (e.g., get, set, command, factory, etc). In the summarized view we present counts of whole stereotype categories (e.g., all the accessors, all the collaborational, etc).

The stereotype distribution highlights the role of a method in the class. It deemphasizes, to a large degree, interaction with other classes. An example of a detailed view for two classes from the open source system *HippoDraw* is given in Fig. 1. The class `DataSource` is largely composed of different types of accessors and mutators while class `DisplayController` primarily constitutes factory and controller methods, i.e., performs most of its work on other classes.

The stereotype category distribution aggregates the data and highlights the degree of coupling and collaboration among classes in a system. It also includes some internal coupling (cohesion) of a class through the main categories of method stereotypes. Additionally, parts of the system not yet implemented (degenerate) are reflected. As can be seen in Fig. 2, the class `DataSource` collaborates (structurally) very little with other classes and has a small percentage of degenerate accessors and mutators. In contrast all methods of class `DisplayController` are collaborational and there are no degenerate methods.

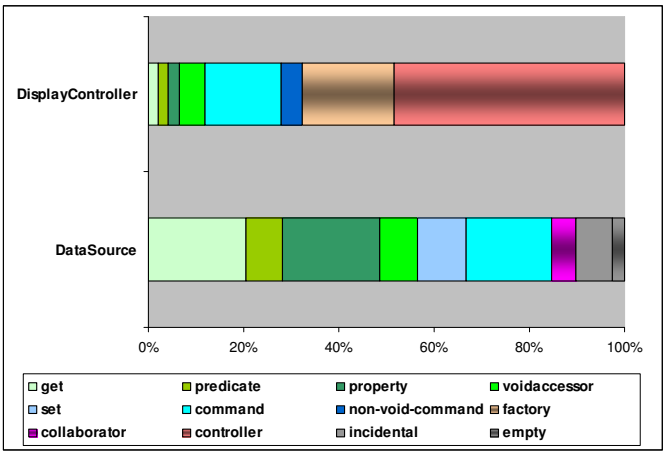

Figure 1. Distribution of stereotypes for the classes DataSource and DisplayController signatures (from HippoDraw).

The methods in the taxonomy are categorized by the data access type (i.e., read or write to the object's state) and by functionality, which is given in the creational, structural, behavioral and collaborational characteristics. These two

perspectives are reflected in the two distributions, stereotype and stereotype category, which complement each other and highlight different aspects of a class's design. The detailed view presents the class's internal structure and responsibilities in terms of types of methods, i.e., we can identify what part of the class is responsible for its creational, structural, behavioral, and control tasks. The summarized view contrasts readers of object's state (accessors) versus writers (mutators) as well as simple readers or writers versus readers or writers that use external objects (e.g., accessor versus accessor collaborator). Additionally, it hightlights the accessors and mutators that are not yet implemented (degenerate). Most likely, there is some plan to complete these in the future. Note that in Fig. 1 and Fig. 2 the class `DataSource` presents two very different distributions. This difference between the stereotype and category view is true for a majority of classes. The charts for the class `DisplayController` are more similar because it has no degenerate methods and all accessors and mutators are collaborational.

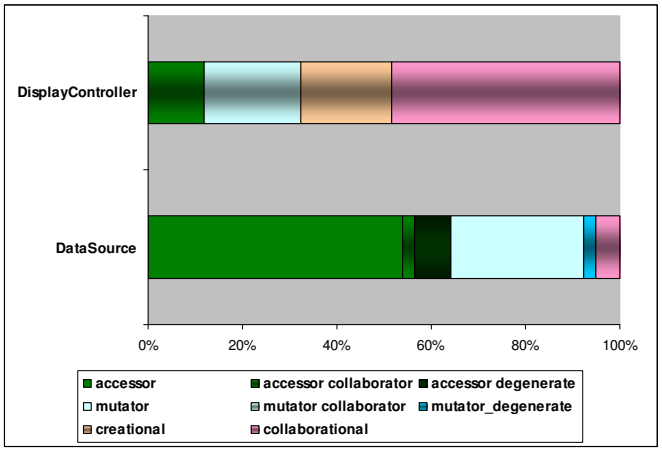

Figure 2. Distribution of categories for the DataSource and DisplayController signatures (from HippoDraw).

These two distributions make up the class signature and provide us with a basis for the automatic identification of class stereotypes.

III. TAXONOMY OF CLASS STEREOTYPES

The process of creating the taxonomy of class stereotypes involved multiple steps. The first step was creating the taxonomy of method stereotypes. We manually examined 150 of the HippoDraw classes in detail and found many patterns of design at the method and class level. The validation of the method's taxonomy on further systems gave us additional evidence of the existence of these patterns of design abstractions.

The next step was to classify software at the system level based on the method stereotypes. Automatic hierarchical (COBWEB) and partitional (X-Means) clustering was used to classify 21 open-source C++ systems listed in Table II. The clusters found are characterized by the frequency and distribution of method stereotypes. The results showed that these distributions are a good indicator of system architecture/design. Additionally, we observed more patterns of the method stereotype distributions at the class level by examining about 250 classes of the systems that were clustered together (Qt and WxWidgets) and separately (HippoDraw, QuantLib, ACE, and Doxygen).

That led to a more thorough investigation of the patterns of design at the class level. We continued the exploration of these patterns by considering the diverse types of features that a class may have with respect to the method's taxonomy and method stereotype distribution. The detection rules were implemented and then we meticulously checked the HippoDraw system and a random set of classes (about 100) in the systems listed in Table II. Some of the rules were refined and improved after this manual verification.

To summarize, the creation of the taxonomy of class stereotypes started with an empirical investigation that led to formulation of the rules for the identification of class stereotypes. The rules were validated on open source systems that led to the rules refinement and further validations of the class's taxonomy.

The list of class stereotypes is presented in Table III. The actual class names are not used in the categorization. While the name can be a good source of information it can also be misleading and we leave this aspect of the investigation for future work.

TABLE II. AN OVERVIEW OF THE SOFTWARE SYSTEMS EXAMINED TO DEVELOP THE TAXONOMY OF CLASS STEREOTYPES. ORDERED BY THE NUMBER OF METHODS.

System	Domain	Methods
C++Fuzzy 0.61	fuzzy logic library	313
CppUnit 1.12.1	framework for unit testing	1335
CEL 1.2.1	game engine	2798
SmartWin++ 2.0.0	GUI and SOAP library	2882
Ivf++ 1.0.0	visualization framework	3032
HippoDraw 1.21.3	data analysis environment	3315
QuantLib 0.9.7	finance library	4235
ClanLib 0.8.1	game SDK	4427
PPTactical 0.9.6	game engine	4887
OpenWBEM 3.2.2	management of systems	4963
ICU 4.0.1	components for Unicode	5984
FlightGear 1.9.1	flight stimulator	6036
Ice 3.3.0	internet communications engine	6952
ACE 5.6.8	communication environment	7867
CGAL 3.4	library of geometric algorithms	11365
Code::Blocks 8.02	IDE	11586
KDevelop 3.5.4	IDE	11799
CrystalSpace 1.2.1	SDK for real-time 3D graphics	12839
Doxygen 1.5.8	documentation system	13445
wxWidgets 2.8.9	GUI framework	34907
Qt 4.4.3	GUI framework	59535
Total		**214502**

Our initial taxonomy included the standard set of overarching stereotypes of *entity*, *boundary* and *control* class stereotypes [9]. We expanded this simple taxonomy as necessary to cover recurring stereotypes that emerged from our empirical investigation. We tried to adopt naming conventions from literature on such things as method stereotypes [7] and bad smells [10]. The list of class stereotypes uncovered is given in Table III. A given class may take on one or more of these stereotypes. That is, a class may have the characteristics of more than one of these stereotypes in certain cases.

For the remainder of the section, each of the class stereotypes is presented along with an explanation of the role and responsibilities of such a class. Additionally, examples of each class stereotype are presented visually along with a specific class and its signature from the HippoDraw system. Due to the space limits these class signatures are shown in a combined view from which the detailed and summarized views can be inferred. An *Entity* is a class that encapsulates data and behavior. It is the keeper of the data model and/or business logic (e.g., the Subject in the Observer pattern). Examples of entity classes are the classes `Range`, `DataSource`, `Rect`, and `BinnerAxis` (see Fig. 3). As can be seen by their signatures, they typically contain accessors and mutators in various proportions and might have a variable percentage of collaborational methods (up to 2/3). They do not have controller methods.

A *Minimal Entity* is a special case of Entity that has only get/set and command methods. It encapsulates very trivial entities (e.g., `Point`). It is considered separately because it is a very simple class that does not require much effort to comprehend. It can also be considered as a Lazy Class (described below).

TABLE III. A TAXONOMY OF CLASS STEREOTYPES

Class Stereotype Name
Entity
Minimal Entity
Data Provider
Commander
Boundary
Factory
Controller
Pure Controller
Large Class
Lazy Class
Degenerate
Data Class
Small Class

.

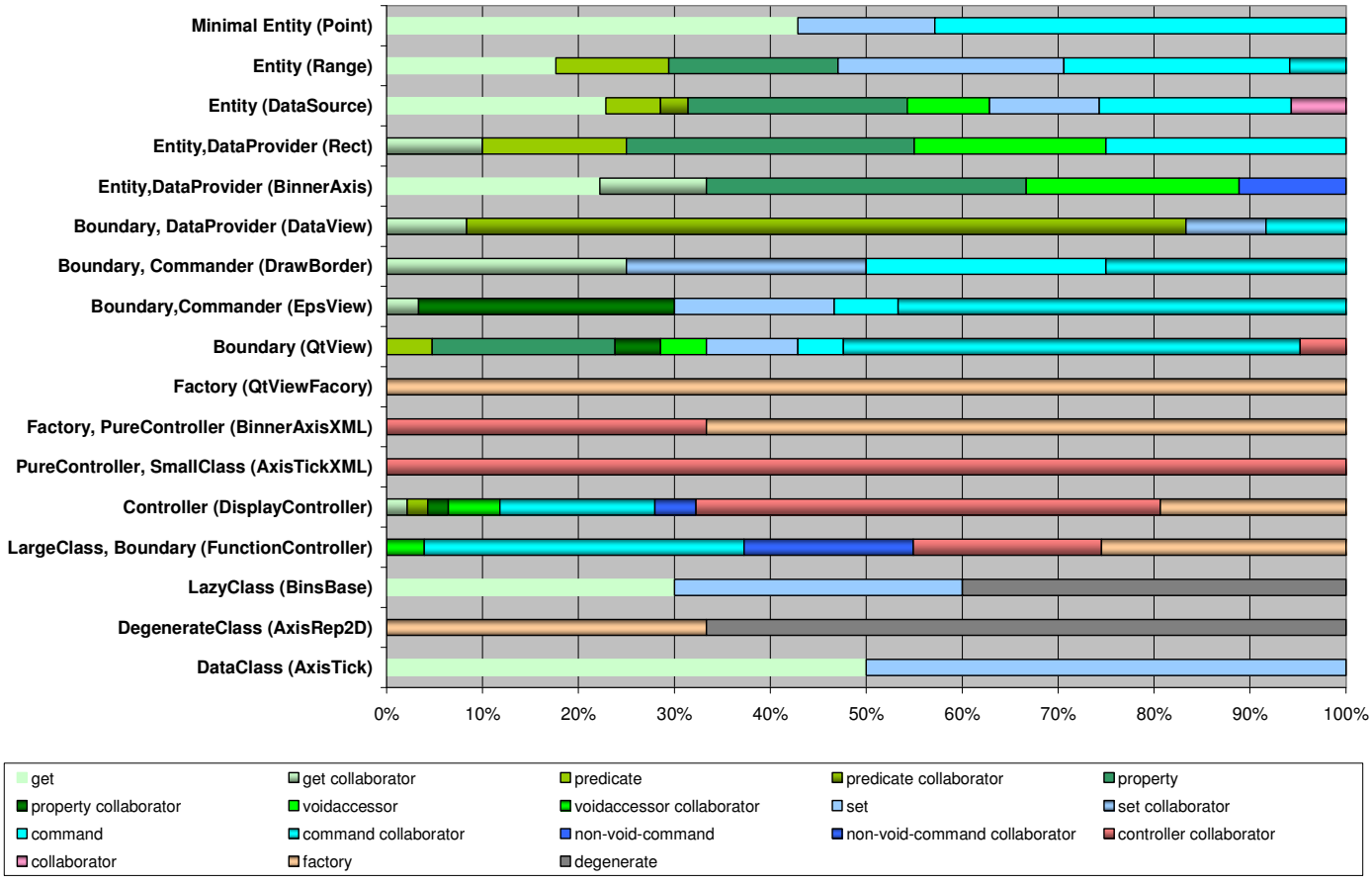

Figure 3. Class stereotypes and their signatures for 17 HippoDraw classes. Each row is labeled with the class stereotype(s) and in parentheses the name of the example class whose data is shown in the row. Each stereotype is automatically identified based on the signatures using the detection rules. Accessors are shown in green colors, mutators – in blue, factory – in tan, collaborational - in rose and turquoise. The method stereotype has a grey fill effect if 'collaborator' is a secondary stereotype for this method.

A *Data Provider* is a class that encapsulates data and consists mainly of accessors. For example, classes `Rect` and `BinnerAxis` have two stereotypes: Entity and DataProvider - more than 75% of their methods are accessors.

A *Commander* is a class that encapsulates behavior and mainly consists of mutators. A large part of the logic for the class's behavior is implemented in command and non-void-command methods. These methods execute complex changes of an object's state. The changes may also involve objects of different classes. The `DrawBorder` and `EpsView` classes are examples of the Commander class. More than 70% of their methods are mutators.

A *Boundary* is a communicator in a system and has a large percentage of collaborational methods but a low percentage of controller and not many factory methods. Alternatively this type of class could be a Data Provider when its main purpose is to get data from a model (when it has mainly accessors) or Commander when its main purpose is to send data and provides updates/output to a model (when it has mainly mutators). For Example, the `DataView` class has both stereotypes Boundary and Data Provider because all of its methods are collaborational, there are no controller methods, and additionally more than 80% of the methods are accessors. The `EpsView` class has both stereotypes Boundary and Commander – most of its methods are collaborational, there are no controller methods, and additionally, 70% of the methods are mutators.

A *Factory* is a creator of objects and has mostly factory methods. The classes `QtViewFactory` and `BinnerAxisXML` are examples of the Factory class stereotype with 100% and 67% of factory methods respectively.

A *Controller* is a class that provides functionality and processes data of external objects. It updates an entity/model working mainly outside of itself, i.e., it has almost all controller and factory methods. The `DisplayController` class is an example of this stereotype. It has about 70% Controller and Factory methods.

A *Pure Controller* is a special case of the Controller. It has 100% Controller and Factory methods and works only outside of itself. We consider this stereotype separately because it is a candidate for the bad-smell God class [11]. A God class is a large controller class that monopolizes most of the system functionality and depends on external data. Methods of the controller class work on data stored in surrounding classes. The Pure Controller class could be a God class if it is a standalone class and consists of many methods.

A *Large Class* is a class which contains too many responsibilities and "is trying to do too much" [10] . "Too much" can be interpreted in different ways using metrics such as LOC, number of attributes, number of methods, complexity metrics, etc. However, those types of metrics do not directly reflect the different roles of a class. We consider a class a Large Class not only if it has many methods, but also if it combines multiple roles, such as Data Provider, Commander, Controller, and Factory. It also could be highly collaborative. The `FunctionController` class is an example of a Large Class. It has a small percentage of accessors, about 50% mutators, 20% controller, and 25% factory. The class is also 100% collaborational.

A *Lazy Class* is a very trivial class which does "too little" [10]. The Lazy Class might occur in the context of a new or planned feature that is not yet completed. Similarly, "too little" can be interpreted using different metrics. But we consider a class as Lazy if it has get/set methods and a low percentage of other methods. The class is also considered Lazy if it has a significant number of degenerates, e.g., `BinsBase` has 40% degenerate methods besides get/set methods.

A *Degenerate Class* is when the state and behavior are degenerate. It has mainly methods that do not read/write to the object's state - half or more methods are incidental or empty. If the Degenerate class is a leaf in the hierarchy, then most likely it needs to be examined for a possible refactoring. An example of the Degenerate class is `AxisRep2D`.

A *Data Class* is a class with degenerate behavior. That is, it has only get and set methods. This type of class passively stores data and does not contain methods that operate on the data. An example of the Data Class is the `AxisTick` class with only get/set methods.

A *Small Class* is a class that only has one or two methods. If it is a standalone class then it is a bad-smell because it degenerates the state and/or behavior.

IV. Automatically Identifying Stereotypes

We developed a tool to automatically identify the class stereotypes presented in the previous section. The tool uses the class signature to assign stereotypes to a class. The rules for identification are based on an empirical investigation of the 21 open source systems in Table II. We present the identification rules to reverse engineer class stereotypes from (C++) source code and give details of a tool that automatically labels a class with its stereotype(s).

A. Rules for Class Stereotype Identification

The rules are based on both the stereotype and category distributions of the class signature. Both distributions are required to determine the stereotype except for the cases of Factory, Data, Degenerate, and Small Class, which require only stereotype or category distribution. To calculate the stereotype we use semantic fractional thresholds of method stereotype frequencies and statistical, average and standard deviation, thresholds that are proposed in [12] as a means to characterize and evaluate the design of object-oriented systems.

We use a fractional threshold of ⅔ for representign situations were a class consists mostly of stereotype A. The thresholds for the Large, Lazy, Degenerate and Small Class were determined empirically by running the rules on the systems HippoDraw and Qt.

We now introduce the notations used in the rules for class stereotype identification. The set of the stereotype is formed as follows.

Let *{stereotype}* be a set of method stereotypes of the type stereotype, e.g., *{get}* is a set consisting of get and get collaborator methods. *{methods}* is a set of all the methods in a class. The set of the stereotype category is formed as follows.

The set *{accessors}* consists of all the accessors (get, predicate, etc), accessors collaborators (get collaborator, predicate collaborator, etc) and accessors degenerate (predicate incidental, voidaccessor empty, etc). The set *{mutators}* is constructed in a similar way.

The set *{collaborators}* consists of all the collaborational methods, e.g., get collaborator, set collaborator, factory collaborator, etc. Thus, the set *{non-collaborators} = {methods} - {collaborators}.*

The set *{degenerate}* consists of accessors degenerate (predicate incidental, void-accessor empty, etc), mutators degenerate (command incidental, non-void command incidental), and collaborator degenerate (collaborator incidental, collaborator empty). We denote by |*stereotype*| the cardinality of the set *{stereotype}*.

To identify the class stereotype ***Entity*** the following conditions need to be satisfied:

- They contain an accessor besides get and a mutator besides set
  $\{accessors\} - \{get\} \neq \emptyset \ \&$
  $\{mutators\} - \{set\} \neq \emptyset$
- The ratio of collaborational to non-collaborational methods is 2:1
  $|collaborators| / |non\text{-}collaborators| = 2$
- They can have factory methods but no controller methods
  $|controller| \neq 0$

To identify the class stereotype ***Minimal Entity*** the following conditions need to be satisfied:

- The only method stereotypes are get, set, and command/non-void-command
  $\{methods\} - (\{get\} \cup \{set\} \cup \{command\} \cup \{non\text{-}void\text{-}command\}) = \emptyset \ \& \ |get| \neq 0 \ \& \ |set| \neq 0 \ \& \ (\{command\} \cup \{non\text{-}void\text{-}command\}) \neq \emptyset$
- The ratio of collaborational to non-collaborational methods is 2:1
  $|collaborators| / |non\text{-}collaborators| = 2$

To identify the class stereotype ***Data Provider*** the following conditions need to be satisfied:

- It consists mostly of accessors
  $|accessors| > 2 \cdot |mutators|$
- Low control of other classes
  $|accessors| > 2 \ (|controller| + |factory|)$

To identify the class stereotype ***Commander*** the following conditions need to be satisfied:

- It consists mostly of mutators
  $|mutators| > 2 \cdot |accessors|$
- Low control of other classes
  $|mutators| > 2 \cdot (|controller| + |factory|)$

To identify the class stereotype ***Boundary*** the following conditions need to be satisfied:

- More collaborators then non-collaborators
  $|collaborators| > |non\text{-}collaborators|$
- Not all the methods are factory methods
  $|factory| < \frac{1}{2} \cdot |methods|$
- Low number of controller methods
  $|controller| < \frac{1}{3} \cdot |methods|$

To identify the class stereotype ***Factory*** the following conditions need to be satisfied:

- It consists mostly of factory methods
  $|factory| > \frac{2}{3} \cdot |methods|$

To identify the class stereotype ***Controller*** the following conditions need to be satisfied:

- High control of other classes
  $|controller| + |factory| > \frac{2}{3} \cdot |methods|$
- Accessor or mutator are present (not only methods that work on external objects exist)
  $|accessors| \neq 0 \vee |mutators| \neq 0$

To identify the class stereotype ***Pure Controller*** the following conditions need to be satisfied:

- Only controller and factory methods with no mutator, accessor, or collaborator methods
  $|controller| + |factory| \neq 0 \ \&$
  $|accessors| + |mutators| + |collaborator| = 0$
- There must be at least one controller method
  $|controller| \neq 0$

To identify the class stereotype ***Large Class*** the following conditions need to be satisfied:

- Categories of stereotypes (accessor with mutator) and stereotypes, factory and controller, are approximately in equal proportions
  $\frac{1}{5} \cdot |methods| < |accessors| + |mutators| < \frac{2}{3} \cdot |methods|$
  $\& \ \frac{1}{5} \cdot |methods| < |factory| + |controller| < \frac{2}{3} \cdot |methods|$
- Controller and factory have to be present
  $|factory| \neq 0 \ \& \ |controller| \neq 0$
- Accessor and mutator have to be present
  $|accessors| \neq 0 \ \& \ |mutators| \neq 0$
- Number of methods in a class is high
  $|methods| > average + stdev$
- Note, *average* and *stdev* of number of methods are calculated per system.

To identify the class stereotype ***Lazy Class*** the following conditions need to be satisfied:

- It has to contain get/set methods
  $|get| + |set| \neq 0$
- It might have a large number of degenerate methods
  $|degenerate| \ / \ |methods| > 1/3$
- Occurrence of other stereotypes is low
  $|methods| - (|get| + |set| + |degenerate|) <= 1/5$

To identify the class stereotype ***Degenerate Class*** the following conditions need to be satisfied:

- It consists of many degenerate methods
  $|degenerate| \ / \ |methods| > 1/2$

To identify the class stereotype ***Data Class*** the following conditions need to be satisfied:

- Only the simple accessor/mutators get and set are present:
  $|get| + |set| \neq 0 \ \& \ |methods| - (|get| + |set|) = 0$

To identify the class stereotype ***Small Class*** the following conditions need to be satisfied:

- Number of methods in a class is less than 3:
  $|methods| < 3$

B. Implementation

We extended our tool, StereoCode [7], to obtain class signatures and automatically identify the stereotypes. StereoCode automatically identifies method stereotypes using lightweight static analysis and an infrastructure based on srcML (SouRce Code Markup Language) [13] an XML representation that supports both document and data views of source code.

The automatic detection of method stereotypes is based on static analysis of the source code using srcML. For each stereotype, an XPath expression is used to detect that particular pattern. StereoCode then re-documents the original source code with the stereotypes with a special `@stereotype` tag in the comments. Next, for class-wide totals, these stereotype comment tags are collected and totaled to obtain the signature (both the stereotype and category distributions) for each class in a software system.

Once the class signatures are generated, they are fed into the tool *StereoClass* that determines the stereotype for a given class using the rules described previously. A class is assigned the stereotype if all conditions of a rule are satisfied. Classes may satisfy more than one rule and the assigned stereotypes are the concatenation of all matches. The part of *StereoClass* for the automatic identification of class stereotypes is implemented in C++. The tool currently works only for C++ source code as input.

V. EVALUATION

To evaluate the approach and taxonomy we compare the results of our automatic classification of a class's stereotype with that of human experts. In this section we will present the details and results of this evaluation.

The system we chose is HippoDraw, an open source application that provides a data-analysis environment. It is a wide-ranging application with parts for data-analysis processing and visualization with an application GUI interface. The source code is well written and follows a pretty consistent object-oriented style. Additionally, the application follows the Model-View-Controller (MVC) architecture that is to a great extent reflected in our class stereotypes.

Three experienced developers (subjects) manually evaluated and stereotyped classes of the HippoDraw system. The subjects are doctoral students in computer science with multiple years of academia and industry experience (OO development). The students are members of our laboratory but were not involved in the implementation and development of this research. In addition, these students were familiar with the design of HippoDraw.

Each subject was given the description of the taxonomy of class stereotypes (as given in Section III), examples of the method stereotypes, and the class signatures for 45 classes from HippoDraw. The subjects were not given the detection rules. The 45 classes were randomly picked and comprise about 15% of the system. This random sample was inspected and found to contain a wide diversity of class stereotypes.

Each subject spent approximately 90 minutes to complete the study. First they read the descriptions of the method and class stereotypes, and then labeled the classes. The subjects were not asked to check the code and made their decisions based on the class signatures.

StereoCode was run on the entire system to generate the class signatures and then StereoClass was run on the class signatures to automatically generate the class stereotypes. Running both tools took less than 2 minutes for the entire system. The results of the subjects' evaluation were compared against the tool results and are given in Table IV.

The results obtained by the tool are shown in the first column. The tool labeled the 45 classes with 70 stereotypes. Almost half of the classes (22) were labeled with one stereotype and 23 classes with two stereotypes. For Example, pairs of class stereotypes included Boundary and Data Provider, Boundary and Degenerate, Entity and Commander, Factory and Small Class.

The columns S1, S2, and S3 show the numbers of class stereotypes obtained by each subject. Two of the subjects identified the number of stereotypes close to that of the tool, while one found more: 72, 86, and 68 vs. 70 (tool). The intersection columns show how the subject's results compare to the results of the tool. Those numbers (52, 47 and 50) show that each subject did not label some stereotypes that the tool found. However, the union of all the subjects with the tool, shown in the last column, indicates that those missed stereotypes were different for each subject in almost all cases. That is, the tool and at least one of the subjects agreed.

The cases where the tool disagreed with the subjects as a whole are of particular interest because they may indicate a

problem with the approach or taxonomy. The stereotype Pure Controller was missed (not labeled) by all three subjects in one case.

TABLE IV. SUMMARY OF ASSESSMENT STUDY. 45 CLASSES FROM HIPPODRAW WERE LABELED WITH CLASS STEREOTYPES BY THE TOOL AND THEN ASSESSED BY 3 EXPERIENCED SUBJECTS (S1-S3).

	Tool	**S_1**	**$S_1 \cap$ Tool**	**S_2**	**$S_2 \cap$ Tool**	**S_3**	**$S_3 \cap$ Tool**	**$(S_1 \cup S_2 \cup S_3) \cap$ Tool**
Entity	**13**	13	8	4	3	9	7	**10**
Minimal Entity	**3**	1	1	0	0	2	0	**1**
DataProvider	8	10	7	13	8	8	6	8
Commander	7	8	6	16	6	8	6	7
Boundary	15	21	13	28	13	18	13	15
Factory	5	5	5	8	4	6	5	5
Controller	6	5	5	4	3	7	6	6
Pure Controller	**2**	1	1	1	1	0	0	**1**
Large Class	3	4	3	5	3	3	3	3
Lazy Class	2	0	0	3	2	0	0	2
Degenerate	2	1	1	1	1	1	1	2
Data Class	2	1	1	2	2	2	2	2
Small Class	2	2	1	1	1	4	1	2
Total	**70**	72	52	86	47	68	50	**64**

However, the subjects labeled the other occurrence of this same stereotype. The stereotype Minimal Entity was missed twice by all the subjects but was identified in a third instance. In the missing cases it was labeled Entity (both times) and Data Class (one time). The third class labeled correctly has very similar distribution to the missed one. The Entity stereotype was missed 3 times out of 13 cases that the tool labeled. The 10 cases where the subjects labeled the classes were very similar to the missed cases. In all three cases the class had the second stereotype Data Provider which maybe the reason for missing the Entity stereotype. In short, all the missed cases have no patterns and can be viewed as just missing a stereotype. Additionally, the stereotypes identified by the subjects but not the tool (false positives) are different for each subject and there is no case when all three subjects have the same false positive.

Through an analysis of the data (missing stereotypes and false positives) we can conclude that the tool performs better than each subject individually or combined. In 91% of the cases (64 out of 70) the subjects were in agreement with the tool. We found after careful examination that it was easy to miss aspects and make mistakes in stereotype identification during manual inspection. Tool support will improve comprehension of a class's design and role in the system.

VI. EMPIRICAL STUDY

To further assess our approach we applied our tools to the five open source systems listed in Table V, ordered by the number of classes in each system. The research questions we address here are: Do these stereotypes identified by the tool exist in nontrivial quantities in real systems? And, do most classes fit into at least one class stereotype?

The systems were chosen to represent a range of sizes, problem domains, and architectures. Some of the systems are mentioned in Bjarne Stroustrup's list of interesting C++ applications[1], while others are taken from sourceforge.net. The categories of the chosen systems are: Game Programming library and SDK (*FlightGear);* Mathematical and Finance library (*QuantLib*); Development and Communication Environments (*KDevelop, Code::Blocks*); and complete application (*HippoDraw*). For the most part, these systems can be considered good examples of object oriented design.

TABLE V. AN OVERVIEW OF THE SOFTWARE SYSTEMS EALUATED IN THE EMPIRICAL STUDY. ORDERED BY THE NUMBER OF METHODS.

System	**Domain**	**Classes**	**Methods**
HippoDraw 1.21.3	data analysis environment	308	3315
QuantLib 0.9.7	finance library	808	4235
FlightGear 1.9.1	flight stimulator	361	6036
Code::Blocks 8.02	IDE	753	11586
KDevelop 3.5.4	IDE	1023	11799
Total		**3253**	**36971**

For each system we automatically determined the stereotypes of each class using the StereoClass tool. The tool took less than 2 minutes for each system. The resulting distribution for each system is given in Table VI.

The results show that all class stereotypes occur in all of these systems. Most classes (94% to 99%) of the system fit into at least one of the class stereotypes. The Commander stereotype occurs in large number of times in some systems, but less than 20% in others. Boundary occurs at least about 40% of the time. Controller and Pure Controller stereotypes do not occur in a significant percentage for the majority of systems, except for the *HippoDraw,* which exploits the MVC architecture. Data Provider stereotype shows a wide distribution – it varies from 1.9% in the *FlightGear* to 62.8% in *QuantLib*. The stereotypes, which are candidates for bad-smell classes, i.e., Controller and Pure Controller, Lazy, Data, Small, and Large Classes, do not occur in significant numbers.

Based on the distribution of the class stereotypes we observe some similarities and differences between the systems. The two IDE systems *KDevelop* and *Code::Blocks* show very similar distribution of class stereotypes. *HippoDraw* and *Quantlib* have a close distribution of the Commander stereotype - it forms a small part of their distribution (18.5% and 18.9% respectively). However, in *FlightGear* this stereotype has a significant portion (84.2%). *HippoDraw* and *FlightGear* are not as much collaborative as *KDevelop, Code::Blocks* and *QuantLib.* The results also show that the frequency and distribution of the class stereotypes across a system reflect an implementation of particular design decisions and good/bad programming practices, and might be an indicator of system architecture/design. For example, the two IDEs we studied, *Code::Blocks* and *KDevelop*, showed very similar distribution of class stereotypes. To explain this we surmise that there is underlying reference architecture for IDEs that both systems follow. While these examples are not terribly surprising, the result clearly is of particular interest.

[1] www.research.att.com/~bs/applications.html

TABLE VI. DISTRIBUTION OF CLASS STEREOTYPES ACROSS 5 OPEN SOURCE SYSTEMS

Stereotype	KDevelop		Code::Blocks		FlightGear		HippoDraw		QuantLib		Min (%)	Max (%)	Aver (%)	Stdev (%)
	#	%	#	%	#	%	#	%	#	%				
Entity	42	**4.1**	23	**3.1**	31	**8.6**	46	**14.9**	20	**2.5**	**2.5**	**14.9**	**6.6**	**5.2**
Minimal Entity	10	**1.0**	6	**0.8**	7	**1.9**	5	**1.6**	0	**0.0**	**0.0**	**1.9**	**1.1**	**0.8**
Data Provider	57	**5.6**	25	**3.3**	7	**1.9**	46	**14.9**	511	**62.8**	**1.9**	**62.8**	**17.7**	**25.7**
Commander	748	**73.1**	608	**80.7**	304	**84.2**	57	**18.5**	154	**18.9**	**18.5**	**84.2**	**55.1**	**33.5**
Boundary	743	**72.6**	573	**76.1**	139	**38.5**	120	**39.0**	700	**86.0**	**38.5**	**86.0**	**62.4**	**22.2**
Factory	10	**1.0**	9	**1.2**	5	**1.4**	38	**12.3**	1	**0.1**	**0.1**	**12.3**	**3.2**	**5.1**
Controller	8	**0.8**	3	**0.4**	6	**1.7**	19	**6.2**	2	**0.2**	**0.2**	**6.2**	**1.9**	**2.5**
Pure Controller	18	**1.8**	6	**0.8**	0	**0.0**	18	**5.8**	14	**1.7**	**0.0**	**5.8**	**2.0**	**2.3**
Large Class	**2**	**0.2**	**2**	**0.3**	0	**0.0**	5	**1.6**	4	**0.5**	**0.0**	**1.6**	**0.5**	**1.9**
Lazy Class	4	**0.4**	0	**0.0**	2	**0.6**	8	**2.6**	0	**0.0**	**0.0**	**2.6**	**0.7**	**2.6**
Degenerate Class	11	**1.1**	12	**1.6**	5	**1.4**	5	**1.6**	1	**0.1**	**0.1**	**1.6**	**1.2**	**0.6**
Data Class	12	**1.2**	6	**0.8**	2	**0.6**	8	**2.6**	6	**0.7**	**0.6**	**2.6**	**1.2**	**0.8**
Small Class	365	**35.7**	166	**22.0**	75	**20.8**	96	**31.2**	339	**41.6**	**20.8**	**41.6**	**30.3**	**8.9**
Coverage	**98%**		**99%**		**95%**		**94%**		**99%**		**94**	**99**	**97**	**2.3**

The chi-square test was performed to investigate the link of class stereotypes in different software systems. The null hypothesis is that the distribution of class stereotypes in different software is a random phenomenon and the alternative hypothesis is that there is a link between class stereotypes and software systems. Chi-square reports a p-value <0.0001 with 95% confidence and 48 degrees of freedom that lets us reject the null hypothesis. The critical and observed values are 65.171 and 2143.018 respectively.

VII. THREATS TO VALIDITY

The assessment of class stereotypes identification and the *StereoClass* tool is subject to a number of threats to validity. The rules for stereotype identification are subjective and thresholds might vary depending on differences in subject's interpretations. The manual inspection of the results includes one software system and additional examples may be warranted. We attempted to construct the study in an unbiased fashion however the selection of the subset of the system is a potential problem. Also, the size of the subset inspected (nearly 15% of the system) could be increased however the assessment is very time consuming for the subjects.

The approach was only applied to C++ systems. However, the srcML format supports Java and rules for method stereotype identification could be adapted for Java. The class stereotype rules are valid for other object-oriented languages and we believe that our approach is extensible to other languages.

VIII. RELATED WORK

The notion of stereotype for object-oriented modeling was first introduced by Wirfs-Brock to support the classification of objects in terms of assigning them certain features and properties [14]. Later, with the introduction of UML, stereotypes became a powerful extension mechanism in UML for introducing new semantics to an existing model while increasing the comprehension of UML diagrams [15], [16]. Work on UML class diagrams based on class stereotypes [4], [5], [6] showed that layouts with additional semantic information about the design were most effective, and the use of class stereotypes plays a significant role in comprehension of these diagrams.

A few approaches identify *key or most important classes* in a software system [17], [18]. Zaidman et al. [17] provide a mechanism based on dynamic coupling and webmining to find classes with a lot of "control" within the application. Orla Greevy et al. [18] identify the key classes and methods which provide functionality for individual features. However, importance of a class is defined by the specific tasks or activities during software maintenance. Our approach provides a description of roles/responsibilities for all the classes in a system and not only for "control" classes.

Gil et al. [19] introduce class-level traceable patterns for Java code (called micro-patterns) with the eventual goal of design assessment. The approach slightly touches upon association and dependency relationships by considering classes that do not propagate calls. A taxonomy of classes to identify changes in object-oriented software based on generalization relationships and the types of data associated with the class is presented by Clarke et al. [20]. Their approach does not reflect role and class responsibilities. A visualization approach to support quick class understanding is proposed by Lanza et al. [21]. The internal structure of a class is presented as a set of a few method layers and an attribute layer. This approach provides semantic information at the class level, but collaborations between different classes are limited to generalization relationships. Another visualization approach to

support method understanding is proposed in [22]. Robbes et al. present microprints, pixel-based representations of methods enriched with semantic information such as state access, control flow, and invocation relationship. This approach provides fine-grained information about the method's internals but not a general characterization.

All of the class categorizations given in the referenced works are primarily based on the access type to the data members. Collaborations between classes (if they are used at all) are limited to inheritance relationships, while association and aggregation relationships are not taken into consideration. Our work fills this gap in class categorizations and identifies stereotypes with respect to a class's architectural importance in the entire system.

IX. CONCLUSIONS

We present a taxonomy of class stereotypes that was derived from an empirical investigation of 21 open source systems written in C++. Additionally, a tool was implemented that automatically reverse engineers a class's stereotype and redocuments the class. The tool can analyze an entire system and redocument it efficiently (in approximately two minutes for Hippodraw). A developers' assessment showed that our classification and the tool accurately describe a class's stereotype.

We feel automatic identification of class stereotypes can support better program comprehension and design recovery. Using both class and method stereotype information a developer should be able to quickly grasp the high level role of the class without reading the source code in detail. Our approach forms a foundation for a number of applications based on class stereotypes. For example, the class stereotypes allow us to determine architectural importance for automated layout of class diagrams or architectural level understanding. It introduces new measures of class's control and can be used to improve existing coupling metrics. Additionally, the stereotypes can be used for mapping to class stereotypes in analysis models, to design pattern roles, and to detect bad-smell classes for refactoring.

The proposed stereotypes could be used not only to characterize design and implementation solutions, they may be used to evaluate and improve design or used as indicators of bad design in need of refactoring. Controller and Pure Controller, Lazy, Data, Small, and Large Classes are candidates for refactoring in particular situations and represent bad smell [10, 11] and we leave this for future work. Our plans are to extend the empirical study to more systems. We also plan to extend the detection rules to Java classes.

REFERENCES

[1] M. Staron, L. Kuzniarz, and C. Wohlin, "Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments," *Journal of Systems and Software,* vol. 79, pp. 727-742, 2006.

[2] M. Genero, J. A. Cruz-Lemus, D. Caivano, S. M. Abrahão, E. Insfrán, and J. A. Carsí, "Does the use of stereotypes improve the comprehension of UML sequence diagrams?," in *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08)*, Kaiserslautern, Germany, 2008, pp. 300-302.

[3] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato, "Influence Web Application Comprehension Tasks Supported by UML Stereotypes: A Series of Four Experiments," *IEEE Transactions on Software Engineering,* vol. 36, pp. 96-118, 2010.

[4] 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)*, Budapest, Hungary 2005, pp. 14-20.

[5] S. Yusuf, H. Kagdi, and J. I. Maletic, "Assessing the Comprehension of UML Diagrams via Eye Tracking " in *15th IEEE International Conference on Program Comprehension (ICPC 2007)*, Banff, Canada, 2007, pp. 113-122. .

[6] B. Sharif and J. I. Maletic, "The Effect of Layout on the Comprehension of UML Class Diagrams: A Controlled Experiment," in *IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'09)*, Edmonton, Canada 2009, pp. 11-18.

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

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

[9] G. Booch, I. Jacobson, and J. Rumbaugh, *The Unified Software Development Process*: Addison-Wesley, 1999.

[10] M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[11] A. J. Riel, *Object-Oriented Design Heuristics*: Addison-Wesley, 1996.

[12] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*: Springer, 2006.

[13] M. L. Collard, J. I. Maletic, and A. Marcus, "Supporting Document and Data Views of Source Code," in *ACM Symposium on Document Engineering (DocEng'02)*, McLean VA, 2002, pp. 34-41.

[14] R. Wirfs-Brock, "Stereotyping: a technique for characterizing objects and their interactions," *Object Magazine,* vol. 3, pp. 50-53, 1993.

[15] M. Gogolla and B. Henderson-Sellers, "Analysis of UML Stereotypes within the UML Metamodel," in *UML*, 2002, pp. 84-99.

[16] C. Atkinson, T. Kuhne, and B. Henderson-Sellers, "Stereotypical Encounters of the Third Kind," in *UML*, 2002, pp. 100-114.

[17] A. Zaidman and S. Demeyer, "Automatic Identification of Key Classes in a Software System Using Webmining Techniques," *Journal of Software Maintenance and Evolution: Research and Practice,* vol. 20, pp. 387-417 2008.

[18] O. Greevy and S. Ducasse, "Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces " in *6th International Workshop on Object-Oriented Reengineering (WOOR'05)*, 2005.

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

[20] P. J. Clarke, B. A. Malloy, and J. P. Gibson, "Using a Taxonomy Tool to Identify Changes in OO Software," in *7th European Conference on Software Maintenance and Reengineering*, 2003, pp. 213-222.

[21] M. Lanza and S. Ducasse, "A Categorization of classes based on the visualization of their Internal Structure: the Class Blueprint," in *16th ACM Conference on Object-Oriented Programming, Systems. Languages and Applications (OOPSLA ' 01)*, 2001, pp. 300-311.

[22] R. Robbes, S. Ducasse, and M. Lanza, "Microprints: A pixelbased semantically rich visualization of methods," in *ESUG 2005 (13th International Smalltalk Conference - Academic Track)* 2005, pp. 172 - 188.