Reverse Engineering Method Stereotypes

Natalia Dragan[1], Michael L. Collard[2], Jonathan I. Maletic[1]
[1]Department of Computer Science
Kent State University
Kent Ohio 44242
[2]Dept of Mathematics & Computer Science
Ashland University
Ashland, Ohio, 44805
{ndragan, collard, jmaletic}@cs.kent.edu

Abstract

An approach to automatically identify the stereotypes of all the methods in an entire system is presented. A taxonomy for object-oriented class method stereotypes is given that unifies and extends the existing literature to address gaps and deficiencies. Based on this taxonomy, a set of definitions is given and method stereotypes are reverse engineered using lightweight static program analysis. Classification is done solely by programming language structures and idioms, in this case C++. The approach is used to automatically re-document each method by annotating the original source code with the stereotype information. A demonstration of the accuracy and scalability of the approach is given.

1. Introduction

The work presented here investigates the problem of reverse engineering method stereotypes. Very few software systems have this information explicitly documented in the source code and while this may be simple to do manually for a small number of methods it is very costly to do for an entire (large) system. We feel method stereotype information forms the basis for supporting more sophisticated types of design recovery.

Given accurate information about method stereotypes, a number of things can be deduced/inferred in the context of a class or interacting classes. For instance, determining method stereotypes is the first step in identifying the stereotype of a class, say boundary, entity, or control. Knowing class stereotypes allows us to determine architectural importance for automated layout of class diagrams or architectural level understanding.

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

As such, we feel this is a very important, yet unexamined, area of OO design recovery. This work has three main contributions. The first is a taxonomic description of axiomatic object-oriented method stereotypes. This is the first comprehensive investigation on this topic with respect to reverse engineering and design recovery. The second contribution is demonstrating the use of a lightweight static analysis method in the identification of method-design features. The lightweight nature of our approach makes it very efficient and scalable and at the same time shows little loss of accuracy due to the tradeoff in lack of deep analysis. The final contribution is the evaluation of the approach that can serve as a benchmark for further studies and investigations.

We restrict our discussion to a single programming language for simplicity of presentation. While our discussion on the taxonomy of method stereotypes is generalizable to a large degree, the realization is specifically for the C++ language. C++ is widely known to be difficult to parse and extract facts from so we feel that this choice best supports the usefulness of our approach.

The paper is organized as follows. In the next section the literature on method stereotype classification is described. We then present a taxonomy of method stereotypes derived from these previous classifications with detailed definitions of each method type in section 3. This is followed by a section (4) describing definitions (rules) for C++. Our tool, *StereoCode*, that automatically identifies method stereotypes using lightweight static analysis methods and the srcML infrastructure is detailed in section 5. We consider the HippoDraw and Qt applications for the validation of the approach (section 6). Here *StereoCode* is applied to the systems and an assessment of our stereotype classification is performed by an experienced developer. Results of this study are given there. Conclusions and future work are presented in the final section.

22nd IEEE International Conference on Software Maintenance (ICSM'06)
0-7695-2354-4/06 $20.00 © 2006 IEEE

2. Method Stereotypes

While the concept of method stereotypes is widely discussed, there is surprisingly little literature on the subject and no formal in-depth studies. This reflects the fuzzy nature of the concept – a stereotype is a high-level description of the role of a method. A stereotype designation gives a clear picture of what a method does and its responsibilities within the class.

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

Most work concerning method classification, for stereotyping, has been with respect to distinguishing the internal state of objects. The focus is the type of access a method has to data members rather than the primary purpose of the method. This is reflected in the naming of accessor methods (a.k.a., *query*, *inspector*, *get*, *getter*, or *getting* method) and mutator methods (a.k.a., *modifier*, *command*, *set*, *setter*, or *setting* method). Typically *get* and *set* methods are considered atomic methods which respectively return a value of a data member or store a value into a data member. We feel that a focus on the internal state is important (while not sufficient) and include this focus in our taxonomy of stereotypes. Accessors and mutators are known by a few different variations however these two terms along with *get* and *set* are the most widespread and appropriate in our opinion. We will stick with these terms and note any variations when appropriate.

The following is a review of the literature that defines method stereotypes. The first group is mainly focused on defining stereotypes by classifying methods for design and development purposes. A later group of literature defines stereotypes with some particular application in mind.

2.1. Stereotype Definitions for Development

Fowler [8] classifies methods at the design level (i.e., UML class) concentrating on the object's state with categories *getting*, *setting*, *query* (accessor), and *modifier* or *command* (mutator). However, details about the classification within accessor and mutator groups are not provided.

Method stereotypes have been proposed to assist in program development. Stroustrup [15] classifies methods (operations) with the goal of helping developers design a class interface in C++. His classification includes the categories described above, *inspector* (accessor) and *modifier* (mutator), and additionally *conversion* (produces an object of a different type based on the applied object), *iterator* (traverses container), and *foundation operator* (constructor, copy constructor, and destructor). A number of well-known programming and data-structure textbooks (e.g., [7], [14], [16], and [17]) propose similar categories. Deitel additionally presents the notion of *predicate* and *utility* (or *helper*) methods. Predicates test the truth or falsity of conditions, and utility methods serve class' public methods and are not part of the class' interface.

Also to assist in program development, Riehle [13] classifies methods in C++ programs based mainly on the read/write type of access to data members. The proposed categories are *query*, *mutation*, and *helper* with fine-grained subcategories. However, their classification does not consider any types of collaborations between classes, identification is not explicitly mentioned, and only a naming convention for the categories is given.

In order to describe the behavior of methods within the class hierarchy, the stereotypes *template* and *hook* [9] have been proposed and used. Template methods perform self calls to abstract methods, while hook methods are designed to be overridden in subclasses.

2.2. Stereotype Definitions for Applications

In general, the previously-discussed work assumes a forward-engineering approach. The developer manually inserts the classification into the source code or defines it at the design level. The stereotype information must then be manually maintained.

However, other work uses stereotypes as a basis for problem solving. In the investigations by Workman [18], a method taxonomy for Java is considered as a base for class categorization to detect plagiarism. The eventual goal is to use the taxonomy for the program-identification problem in comparison analysis. Some use of the collaboration between methods is considered, however no means for identification is given.

Clarke et al., [3] presents a taxonomy of classes for the identification of changes in object-oriented software. Their approach takes into consideration the properties and features of the class which are based on the relationships between classes within the inheritance hierarchy and types of data associated with the class. Any other types of collaborations between classes or at the method level are not considered in the taxonomy.

Visualization approaches to support method and class understanding are proposed in [1] and [11]. Arevalo et al. [1] propose X-Ray Views which group methods based on the state usage (state access), external/internal calls (self and super calls), and behavioral skeleton (client access) using concept analysis. This approach considers collaboration between groups of methods and attributes of a single class in terms of the direct or indirect accessors; however no differentiation whether the method reads or updates the class attributes is done. Lanza et al. [11] consider categorization of classes based on the class blueprint, i.e., visual representation of the class as a set of four method's layers: initialization, interface, implementation and accessor, and an attribute layer. This approach provides semantic information on the method level, but collaborations between methods of different classes are limited to generalization relationships.

Stereotypes are also used as a powerful extension mechanism in the UML [10]. There are two basic ways of using stereotypes in UML: to emulate metamodel extensions and to support the classification of objects in terms of assigning them certain features and properties [2]. As an extension mechanism stereotyping allows us to introduce new semantics to an existing model.

All of the stereotype definitions given in this section 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 the method-stereotype classification, presents a full taxonomy, and presents a method to automatically extract, and re-document, this information from the source code.

3. A Taxonomy of Stereotypes

We now unify the literature on method stereotypes by integrating the different perspectives given in the previous section while simultaneously addressing a number of deficiencies. Our taxonomy is based on a method's main role and duties while emphasizing the creational, structural, and collaborative aspects with respect to a class's design.

Methods can be viewed from a number of different perspectives however we categorized them broadly by how they access data (i.e., a method changes the objects state or leaves it constant) and their behavioral characteristics that is, creational, structural, or collaborational. *Creational* methods are responsible for creating or destroying objects of the class. *Structural* methods allow one to set or get data member (attribute) values, hence provide and support the structure of the class. *Collaborational* methods help define the communication between objects and how objects are controlled in the system. Taking into consideration both perspectives we present definitions of the following method categories at the implementation level: creational, structural (accessor, mutator) and collaborational. The overview of our method stereotype taxonomy is given in Table 1.

Of the stereotypes presented in Table 1, factory, command, and collaborator are the only ones not typically discussed in the context of method stereotype literature. Factory and command are two variants on well know stereotypes that we feel are important distinctions for design recovery. Collaborators are methods that connect one object with other types of objects. Typically, these are put under the general stereotype of accessor or mutators and no distinction is made for interacting with external objects. We feel this is a major deficiency in previous method stereotype descriptions.

Definitions of method categories and their subtypes will now be presented. This is illustrated by considering the class `DataSource` and `DisplayController` from the HippoDraw application given in Figure 1 and Figure 2. Note that we do not consider the full C++ interfaces but only the methods and attributes that are pertinent. The class `DataSource` supplies one or more arrays of data. The class `DisplayController` is an interface between a graphic user interface and the displays. These two classes will be used throughout as examples for describing stereotypes. Necessary information about particular method definitions will be described as needed. We now discuss each category (each column in Table 1) of method stereotypes.

Table 1. A taxonomy of method stereotypes.

Structural		Collaborational	Creational
Accessor	**Mutator**		
Get	Set	Collaborator-Accessor	Constructor
Predicate	Command	Collaborator-Mutator	Copy Constructor
Property			Destructor
			Factory

3.1. Structural Methods: Accessors

An *accessor* is a read-only method that returns information about the data members of an object. It does not change the state of the object, i.e., the value of any data member. There are three types: get, predicate, and property.

A *get* method is an accessor that returns the value of a data member. The purpose of this method is very simple and primitive. In Figure 1 the class `DataSource` has the get methods `getName()` and `rows()`, which return the values of the data members `m_ds_name` and `m_rows` respectively. Although it may be seen as a predicate

method, the method `isNull()` is classified as a get method since it directly returns the value of the data member `m_is_null`.

```
class  DataSource :public Observable
{
private:
  string m_ds_name;
  vector<string> m_labels;
  bool m_is_null;
protected:
  mutable vector<double> m_array;
  int m_rows;

public:
  /** @stereotype get */
  bool isNull() const;
  /** @stereotype get */
  virtual int rows() const;
  /** @stereotype get */
  const string& getName() const;

  /** @stereotype predicate */
  bool isValidLabel(const string& label) const;

  /** @stereotype property */
  virtual double sum(int column) const;
  /** @stereotype property */
  int columns() const;
  /** @stereotype property */
  virtual int indexOfMinElement(int index)const;
  /** @stereotype set */
  void setName(const string& name);
  /** @stereotype set */
  void setLabels(const vector<string>& v);

  /** @stereotype command */
  virtual void clear();
  /** @stereotype command */
  virtual void reserve(int count );
};
```

Figure 1. The HippoDraw C++ class DataSource after re-documenting with the method stereotypes.

A *predicate* method is an accessor that returns a Boolean result. The result is not the value of a data member but instead computed based on a data member(s). The condition may be directly based on the data member's values, or indirectly based using other get, predicate, or property methods. In Figure 1 the class `DataSource` has the predicate method `isValidLabel()` which returns whether the parameter label is a valid label for a column. This information is not directly stored in a data member, but must be determined by traversing the vector data member `m_labels`.

A *property* method is an accessor that returns information about an object based on data member values. It can compute a result, compare two objects, or convert a class object. As in the case of the predicate method, the returned value computed is based on the data members or by calling other accessor methods. The main feature of this type of method is that it derives some characteristics of the object from the data members' values. In Figure 1 the class `DataSource` has the property method `sum()` which returns the sum of all the elements in the sequence of the column specified by the parameter `column`, the method `columns()` which returns the number of columns or data arrays available from `DataSource` by calling the method `size()` on the vector, and method `indexOfMinElement()` which returns the row index of the minimum element in a column for the given column using the property method `columns()`.

```
class DisplayController
{
private:
  static DisplayController* s_instance;
  string m_null_string;
  vector<string> m_null_vector;

public:
  /** @stereotype collaborator,factory */
  PlotterBase* createDisplay(const string& name)

  /** @stereotype collaborator-property*/
  const vector<string>& getDisplayTypes() const;

  /** @stereotype collaborator-predicate*/
  bool hasControlPoints
             (const PlotterBase* plotter) const;

  /** @stereotype collaborator-command*/
  void setIntervalCount
      (const PlotterBase* plotter, int count);
};
```

Figure 2. The HippoDraw C++ class DisplayController after re-documenting with the method stereotypes.

3.2. Structural Methods: Mutators

A *mutator* is a method that changes the state of the object to which it belongs. A widespread convention is that mutators (a.k.a. *command* by Fowler) do not return a value to the client. In general we can assume that methods that return a value are accessors (queries). Meyer [12] refers to this as the Command-Query separation principle. We divide these types of methods into two classes: *set* and *command*.

A *set* method is a mutator that changes the value of a data member. This involves a direct change of a data member, i.e., some new value is assigned to the data member. The new value is typically given to the method as a parameter. In Figure 1 the class `DataSource` has the set method `setName()` which assigns the name of the ntuple (row), i.e., the data member `m_ds_name`, and the method `setLabels` which assigns the label to each column from the vector of strings `v`, i.e., the data member `m_labels`.

A *command* is a mutator that executes a complex change of the object's state. The change may involve several data members. It may change the data members either directly or indirectly using another mutator. Typically, the command method does not return any values or the only return value is a status or error code. In Figure 1 the class `DataSource` has the command method `clear()` which clears the data source, and the

method `reserve()` that for each column, reserves enough memory for the row to grow to size `count`.

3.3. Collaborational Methods

A *collaborator* is a method that works on objects of classes different from itself. It normally doesn't change the data members of the method's object. These other objects are typically parameters or local variables, however they may also be accessed indirectly through a data member that is a pointer/reference or may contain a pointer/reference (e.g., a vector of object pointers).

Collaborator methods work outside the class of which they are part. This category is different because collaborators can additionally be any subtype of accessor or mutator. Collaborator-accessors methods read and return objects of other classes. Collaborator-mutators methods change the state of the external objects with which they have relations. In Figure 2 the class `DisplayController` has both collaborator-accessors and collaborator-mutators. The collaborator-property method `getDisplayTypes()` returns the types of displays available from a local object. The collaborator-predicate method `hasControlPoints()` returns true if the object is created using the parameter `plotter`. The collaborator-command method `setIntervalCount()` sets the interval count on an object of class `NTuple`.

3.4. Creational Methods

In the work presented here, we restrict the consideration of creational methods to the factory method because constructor, copy constructor, and destructor methods are well-known and previously defined in the general literature on object-oriented development. Additionally, these former methods 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 *factory* stereotype is a method that creates an object and returns it to the client. It is also called an object-creation method. The factory stereotype for methods is akin to what is described in the factory method design pattern. Fowler mentions this stereotype under the general category of utility methods. These types of methods work outside of the class that they are part and change the state of the external objects with which they have relations. In Figure 2 the class `DisplayController` has the factory `createDisplay()` which creates an instance whose type is a derivation of the class `PlotterBase`.

4. Rules for Stereotype Identification in C++

Based on our taxonomy that was derived from the literature we now identify the main features to support reverse engineering method stereotypes from source code written in C++. These features include: access type to data members, a method's return type, and the type and multiplicity of parameters. However, in the context of C++ these main features are not sufficient to classify a method's stereotype. We identified additional features to support the automatic identification, including an indicator if the method changes the object state (i.e., a method can be const or non-const) and local variable types.

The rules were further refined by an examination of a number of C++ systems (for idioms). Specifically, among others, we examined LAN simulation system (an opensource small simulation of a LAN network to illustrate good object-oriented design, Java, 20 classes), HotDraw (an opensource two-dimensional graphics framework for structured drawing editors, Java, about 150 classes), and HippoDraw (an opensource data analysis environment, C++, over 200 classes).

We now provide detailed rules for automatic detection of method stereotypes (in C++), show that most of the categories are disjoint, and explain which multiple stereotypes are possible.

To identify the stereotype *Accessor::Get* the following conditions need to be satisfied:

- method is `const`
- returns a data member
- return type is primitive or container of a primitives

To identify the stereotype *Accessor::Predicate* the following conditions need to be satisfied:

- method is `const`
- returns a Boolean value that is not a data member

To identify the stereotype *Accessor::Property* the following conditions need to be satisfied:

- method is `const`
- does not return a data member
- return type is primitive or container of primitives
- return type is not Boolean

To identify the stereotype *Mutator::Set* the following conditions need to be satisfied:

- method is not `const`
- return type is `void` or Boolean
- only one data member is changed

To identify the stereotype *Mutator::Command* the following conditions need to be satisfied:

- method is not `const`
- return type is `void` or Boolean
- complex change to the object's state is performed e.g., more than one data member was changed

To identify the stereotype *Collaborator* one of the following statements needs to be satisfied:

- returns void and at least one of the method's parameters or local variables is an object

- returns a parameter or local variable that is an object

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

- returns an object created in the method's body

The rules of the accessors, mutators, and factory will result in a method only having a single stereotype from these categories. A method may have a second stereotype of *collaborator* if it has a parameter or a local variable that is an object (which are rules for collaborators). In this case multiple stereotypes are assigned to this method, e.g., *collaborator, property* or *collaborator, command*, etc.

More analysis is required to further refine whether a collaborator method also can be labeled with a subtype of accessor or mutator. For example, if a method changes a few data members of a different class then this method is a *collaborator-command*. In the experiments performed here we did not differentiate between different subtypes of a collaborator and leave these considerations as future work. In the next section we describe how these rules for the automatic identification of methods are implemented to reverse engineer stereotypes from existing C++ source.

5. Implementation of the *StereoCode* Tool

Implementation of our approach requires three main activities: static analysis, stereotype identification, and re-documentation (as can be seen in Figure 3). The driving activity, stereotype identification, which our tool *StereoCode* implements requires rules that define stereotype categories to be realized and applied to a code base. The previous section describes these rules in detail. Obviously, to implement the rules a reasonable amount of static program analysis and fact extraction must be supported. We have developed an infrastructure to support static analysis, querying, and fact extraction of C/C++ source and use this to reverse engineer method stereotypes. This same infrastructure directly supports the re-documentation (annotation) of the source code via the addition of comments.

Our infrastructure is based on srcML[1] (SouRce Code Markup Language) [6] an XML representation that supports both document and data views of source code. The format supports lightweight static program analysis using standard XML tools while at the same time preserving all original lexical information. This allows the integration of static program analysis into a transformation. 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 C++ source code [4] and transformation (refactoring) [5].

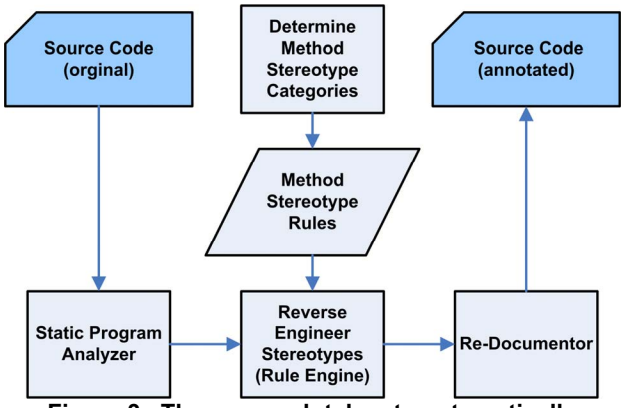

Figure 3. The approach taken to automatically identify and re-document the source code with method stereotypes.

For the purposes of method-stereotype identification we translate the source code into srcML and then *StereoCode* takes over by leverage XPath, an XML standard for addressing locations in XML. The rules described in section 4 are realized as XPath queries on srcML. This is a relatively straightforward process and the complete implementation using XPath took only a few days. As such one can define their own custom stereotype rules quite easily.

Adding the comments is also quite efficient in the context of srcML. The XPath query gives us a location of the method and we can then do a simple transformation within the srcML document to add the necessary comments. This process is fully automated and very efficient/scalable. The next two subsections describe each of these activities in more detail.

5.1. Rules as XPath Queries

The rules for the method stereotypes were converted into XPath predicates using the terminology of srcML. If all the predicates for a stereotype are true, then the function matches that stereotype. Some of the rules can be directly extracted from srcML, while others take a bit more processing. For example, because the keyword `const` is marked with an element `specifier` in srcML it is directly extractable using the XPath expression `specifier='const'`. Likewise, the return type of a function is in the element `type`. The type can be directly compared and easily determine if it is of type void with `type='void'`. For matching specific parts of a type the individual names can be used, e.g., matching "Data" in the type `const Data&` can be matched with the expression `type/name='Data'`. This is true if at least one element `name` is equal to "`Data`". A full example of

[1] Pronounced source M L.
[2] See www.sdml.info for translator tool download.

the XPath query that matches function definitions for methods with the stereotype *predicate* is:

```
function[specifier='const']
[type/name='bool']
[descendant::return/expr/name=
     descendant::decl/name
     or descendant::return/expr='false'
     or descendant::return/expr='true']
[not(contains(descendant::type,'*'))]
```

For other rules further processing is required. For example, the expressions in return statements are found directly using the expression `return/expr`. However, determining that the expressions consist of a variable that is a data member requires non-local (to the definition) information. A variable is considered a data member if it is not declared in the function definition, i.e., not a parameter or a local variable. This allowed stereotype determination based on the function definition alone without the corresponding class information (complete symbol table) or full inheritance hierarchy.

Determining the number of changed data members is done in a similar manner on expression statements. Because srcML does not explicitly mark operators, text comparison is used to determine if a data member is assigned. Then counts of the assigned data members are used. A number of our rules make a distinction between objects and variables, i.e., between user-defined classes and standard types. The srcML representation makes no distinction between primitive and non-primitive types, so a list of primitive types is maintained which are compared to the declaration names.

5.2. Automatic Re-Documentation

In our approach, method stereotypes are not only automatically classified but also used to re-document the source code of the original method. This re-documentation is one of the most flexible ways of storing the method stereotypes as it shows the method in context, allows for updates by the developer, and can be easily extracted for aggregation using the srcML platform.

The re-documentation is performed on the class declaration in the include file (e.g., *DataSource.h*) by the insertion of a Javadoc/Doxygen-like formatted comment before the function declaration. For example, given the original method declaration:

```
// set current status
void setStatus(const string& s);
```

we re-document it with the derived stereotype:

```
// set current status
/** @stereotype set */
void setStatus(const string& s);
```

In order to accomplish this, both the class declaration file (e.g., *DataSource.h*) and the corresponding implementation/definition file (e.g., *DataSource.cpp*) are translated into srcML. A XSLT program we built, *annotate_stereotype.xsl*, performs an identity transformation of the srcML file with special handling of the methods in class definitions. A comment with the stereotype annotation is inserted before the method declaration/definition in the class declaration file. Pure-virtual functions are not annotated. After the re-documentation is done on the srcML the file is converted back to a source-code file via a quick translation.

To automatically determine the stereotype it is necessary to examine the method body. If the method is defined inside the class then it can be determined in-place. However, if the class only declares the method then the method definition has to be found. The XSLT program is passed the name of the corresponding definition file as a parameter which it searches for the proper method definition. The program also searches for methods defined in the include file but outside of the class. The linking of the declaration to the definition is based on the name and the parameter signature, i.e., ordering and type of the parameters. Namespace prefixes on type names were not used in order keep analysis local to the definition.

All of the rules XPath expressions are applied to each function definition. The inserted stereotype is the concatenation of all matches. This determines whether the predicates given are unique. The re-documentation is applied to an entire project by repeating this process on each pair of declaration/definition files.

6. Evaluation of the Approach

In order to assess the approach we applied *StereoCode* to the medium and large-sized software systems HippoDraw and Qt. HippoDraw is an opensource application providing a data-analysis environment. It is a wide-ranging application with parts for data-analysis processing and visualization with an application GUI interface. The studied version contains approximately 60 KLOC of source code in over 400 C++ files. The system contains over 200 classes (2900 methods & free functions). Qt is a cross-platform C++ GUI framework. The 4.1.2 version contains about 1000 KLOC of source code. Over 1000 classes with about 20900 methods were annotated. The source code for both is well written and follows a pretty consistent object-oriented style.

We used *StereoCode* to re-document the original source code. Translation to srcML took less then 15 seconds for HippoDraw and 3 minutes for Qt. The identification and re-documentation both took less then 30 seconds and 6 minutes respectively. Converting back to raw source is very fast (two seconds for HippoDraw and under 30 seconds for Qt). These figures are from running our application on a desktop machine under Linux. Table 2 gives a summary of the results of the re-documentation process. Of the 2706 methods in

HippoDraw all but 220 were classified with one or more stereotypes. Additionally, 30 were empty methods (no body) and could not be classified (for a total of about 9%). Qt had a much lower percentage of unclassified and empty methods (only 2%).

Table 2. Summary results for the reverse engineering of method stereotypes from HippoDraw and Qt. Each method was labeled with one or more stereotypes.

Stereotype(s)	Occurrences		%	
	HD	Qt	HD	Qt
command	439	1281	16.2	6.1
property	361	1098	13.3	5.3
collaborator	239	3707	8.8	17.7
get	133	109	4.9	0.5
predicate	99	54	3.7	0.3
set	84	161	3.1	0.8
factory	2		0.1	
Number of methods labeled with only one stereotype	**1357**	**6410**	**50.1**	**30.7**
collaborator,command	623	8546	23.0	40.9
collaborator,factory	296	889	10.9	4.3
collaborator,property	90	2806	3.3	13.4
collaborator,set	30	819	1.1	3.9
collaborator,predicate	23	471	0.8	2.3
collaborator,get	22	378	0.8	1.8
collaborator,empty_method	14	156	0.5	0.8
property,empty_method	1		0.0	
Number of methods labeled with two stereotypes	**1099**	**14067**	**40.6**	**67.4**
unclassified	220	386	8.1	1.8
empty_method	30	8	1.1	0.04
Overall Total	**2706**	**20869**	**100**	**100**

Our approach labels a method with multiple stereotypes if it meets the constraints of multiple rules. HippoDraw has almost equal percentage of methods labeled with only one stereotype and with two stereotypes (50% and 41% respectively). Qt had a much higher percentage of methods with multiple stereotypes (67%). No methods were labeled with more than two. Command, property, and collaborator make up the largest percentage of methods. When combined with the multi-labeled methods these three stereotypes make up a significantly larger portion of the methods. Variations on collaborator make up nearly 50% of the methods in HippoDraw with a much higher percentage in Qt (85%). The other stereotypes make up a relatively small portion.

A small number of the empty methods were classified as collaborators due to the associated formal parameter list (i.e., an external object was passed to the method). We labeled these methods *collaborator, empty_method* to distinguish them from other collaborators.

StereoCode was also applied to the opensource web development project Mozilla. The same categories as in HippoDraw and Qt occurred but a large percentage of all methods were variations on collaborators (about 95%). More analysis is required which takes into account indirect calls and the heavy use of macros in this system.

6.1. Developer Assessment

To further assess our approach an experienced developer (subject) helped to rate how well the automatically-generated annotations matched with each method. The subject is a graduate student in computer science with multiple years of industry experience (OO development). This student is a member of our laboratory but was not involved in the implementation, development, or discussions on this research.

Table 3. Summary of assessment study. 19 classes from HippoDraw were annotated with method stereotypes and then assessed by an experienced developer. The automated re-documentation of each method was rated as Very Good, Good, Fair, or Poor.

Class Name	# of Meth	VG	G	F	P
BinsFactory	2	2			
CircularBuffer	4	3			1
CutController	22	19	3		
DataRepController	8	8			
DataSource	28	23	2		3
DataSourceException	1				1
DisplayController	80	76	4		
FunctionController	45	28	5	10	2
LinearTransform	9	5		1	3
NTuple	32	29	2		1
NTupleController	17	16	1		
NTupleSorter	11	5	1	2	3
OpenGLView	30	21	7		2
OpenGLWindow	4	4			
QtView	18	13	5		
QtViewImp	28	20	6	1	1
QtViewWidget	17	13	2	1	1
ViewBase	8	4	2		2
ViewFactory	1				1
Total Methods	**365**				
Subject's Assessment		289 (79%)	40 (11%)	15 (4%)	21 (6%)
Errors due to poor design of methods			1		5
Errors due to lightweight analysis			19		5
Errors due to differences in interpretation			7	5	11

The subject was given a subset of the entire HippoDraw system comprising approximately 14% of the system. We randomly selected 19 classes that consisted of 365 methods. However, these 19 classes were inspected to assure that a wide diversity of stereotypes was represented. The assessment procedure took the subject more than four hours of constant work.

The assessment was run as follows. The subject was given the definitions of our stereotypes (as stated in Section 3) to make them familiar with the terminology. The subject was not given the rules (as stated in Section 4 and 5). Then they were asked to rate our classification of a method's stereotype on a Likert scale of 1) Very Good - completely agree, 2) Good - agree with minor issue, 3) Fair - disagree because of false positive, and 4) Poor - complete disagree. The subject also provided a short explanation for why they did not agree (Fair or Poor) with a particular method classification.

In a postmortem the subject described their method. They went through the code class by class and examined all the methods in one class at a time. The assessment of the stereotype classification was done within the context of the entire class. That is, they examined other methods in the class and the class attributes (data members). Rarely, if ever, did the subject need to extend their scope beyond that of the method's class to assess the classification. The subject quickly attempted to categorize each method broadly as a mutator or accessor via the method's signature – name, parameters, and return type. The next step would then be to inspect how the method utilized data members. Finally, if necessary, what other method calls were made inside the method were examined.

The results of the subject's assessment are given in Table 3. We've included the list of classes that were inspected and the number of methods for each. The overall ratings per method are also given as Very Good, Good, Fair, and Poor.

Based on the assessment, our system labeled 90% of the methods inspected by the subject as either Very Good or Good. In 10% of the methods the subject was in disagreement with our classification. Upon close inspection of the subject's ratings and notes, we were able to better understand these disagreements. We found that methods viewed as poorly classified were due to three issues: the method was poorly designed, the lightweight static analysis applied was insufficient, or there were differences of opinions on the stereotype definitions (between us and the subject).

The subject stated that a small number of disagreements with our classification came about due to poorly designed methods in the application under study. Examples of this were methods that used output parameters when it was clearly unnecessary. In one example a method was classified as collaborator but should have been classified as collaborator and property. That is, the property was returned via the parameter list instead of a return value, which in this case was `void`.

In a small number of cases (five methods, less than 1.5%) our system did not perform the complex analysis of the code required to get a correct result (good or very good instead of poor). Additionally, in another 19 methods more analysis would be necessary to increase the rating from good to very good.

There was a difference in the interpretation of the stereotype definitions by our subject in a small number of cases and these are noted in the table. For example, we labeled one method as collaborator and command however the subject felt it should only be stereotyped as a collaborator. Upon careful inspection of our definitions this particular method should in fact be labeled as both collaborator and command. These cases seem to represent differing opinions and views on method stereotype classifications. The definitions could be modified to address these differences.

6.2. Threats to Validity

The assessment of *StereoCode* is subject to a number of threats to validity. There was only one subject and as such no statistical analysis to the significance of the results could be employed. Only one software system was examined in detail and additional systems are warranted to better justify the results. As stated, we attempted to construct the study in an unbiased fashion however the selection of the subset of the application is a potential problem. Also, the size of the subset inspected (nearly 14% of the system) could be increased however the assessment is quite time consuming (it would require about 30 hours of the subject's work for the manual re-documentation of the entire system) and it is quite difficult securing subjects to conduct such a study.

7. Conclusions

We presented a tool, *StereoCode*, for reverse engineering method stereotypes of an entire system. Our assessment demonstrates that our stereotype classification along with our tool for automatically identifying and re-documenting method stereotypes is both sound and efficient. Our results were very good as an experienced developer agreed 90% of the time with our classification. *StereoCode*, based on a lightweight static program analysis approach, is very efficient and usable – while still giving very good results. While our system incorrectly labeled 10% of the methods for HippoDraw, only a very small number of methods (less than 1.5 % of those assessed) were not correctly classified in the lightweight approach. Even if this

number proved to be larger for different systems, the cost trade-off is hard to compete with.

The approach is obviously limited by the definitions of the stereotypes and the underlying need for information about the method. The implementation is specifically for C++ so the usefulness of the taxonomy applied to other languages is still open. Knowledge about the problem/solution domains and programming idioms can play an important role in the quality of the results. As can be seen for the stereotype categories we presented, much can be done with fairly simple assumptions and heuristics. In addition the re-documentation allows for the developer to improve the quality by further (manual) refinement. Even the rules used in *StereoCode* can be easily modified and customized. Poorly written code and the lack of standard idioms also pose serious limitations. Systems of this quality give little hope for design recovery - manual or automatic.

A very small percentage of the incorrectly labeled methods are due to the poor programming style (5 methods, less than 1.5%). In general, the identification of property and predicate methods might be affected by the poor programming style. However, the statistics presented give hope that for any arbitrary software system all methods will be labeled, but possibly with less accuracy if there are many poor-designed methods exist.

While our tools are specifically for C++ the approach can be easily extended to other object-oriented programming languages (ex., Java, C#). For example, the rule 'if the method is `const`' can be substituted with the more general one 'if the method changes the data member'. Other rules satisfy the object-oriented programming language structures and idioms.

We feel that this work forms the basis for a number of avenues of research in design recovery. First is the construction of design-quality metrics based on stereotype classification. In the initial phases of this investigation we attempted to apply classical object oriented metrics to the problem of method-stereotype classification. However, these metrics are too coarse grained and were poor predictors of stereotype. Knowing method stereotypes may also assist in identifying design patterns. However, our goal is to extend this approach to automatically reverse engineer class stereotypes. Our current investigations have shown that information about the method stereotypes is a necessary requirement to adequately address this problem.

8. References

[1] Arevalo, G., Ducasse, S., and Nierstrasz, O., " XRay Views: Understanding the Internals of Classes", in Proceedings of 18th IEEE International Conference on Automated Software Engineering, 2003, pp. 267-270.

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

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

[4] Collard, M. L., Kagdi, H. H., and Maletic, J. I., "An XML-Based Lightweight C++ Fact Extractor", in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03), Portland, OR, May 10-11 2003, pp. 134-143.

[5] Collard, M. L. and Maletic, J. I., "Document-Oriented Source Code Transformation using XML", in Proceedings of 1st International Workshop on Software Evolution Transformation (SET'04), Delft, Nov. 9 2004, pp. 11-14.

[6] Collard, M. L., Maletic, J. I., and Marcus, A., "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9 2002, pp. 34-41.

[7] Deitel, H. M. and Deitel, P. J., C++ How to Program Third Edition, Prentice Hall, 2001.

[8] Fowler, M., UML Distilled 3rd Ed. A Brief Guide to the Standard Object Modeling Language, Addison-Wesley, 2000.

[9] Gamma, E., et al, Design Patterns Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

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

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

[12] Meyer, B., Object-Oriented Software Construction, Prentice-Hall, 2000.

[13] Riehle, D. and Berczuk, S., "Types of Member Functions in C++", http://www.riehle.org/computer-science/industry/publications.html, 2001.

[14] Savitch, W., Problem Solving with C++ The Object of Programming Second Edition, Addisson-Wesley ed., 1999.

[15] Stroustrup, B., The C++ Programming Language, Addison-Wesley, 2000.

[16] Tremblay, J.-P. and Cheston, G. A., Data Structures and Software Development in an Object-Oriented Domain Eiffel Edition, Prentice Hall, 2001.

[17] Weiss, M. A., Data Structures & Algorithm Analysis in C++, Addisson-Wesley, 1999.

[18] Workman, D., "A Class and Method Taxonomy for Object-Oriented Programs", Software Engineering Notes, vol. 27, no. 2, 2002, pp. 53-58.