

THE EMERGENT LAWS OF METHOD AND CLASS STEREOTYPES IN
OBJECT ORIENTED SOFTWARE

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Natalia Dragan

December 16, 2010

Dissertation written by

Natalia Dragan

B.S., Moldovian State University, Moldova, 1983

M.S., Kent State University, USA, 2005

Ph.D., Kent State University, USA, 2010

Approved by

Dr. Jonathan Maletic

Chair, Doctoral Dissertation Committee

Dr. Paul Wang

Members, Doctoral Dissertation Committee

Dr. Ruoming Jin

Dr. Michael Collard

Dr. Robin Selinger

Accepted by

Dr. Jonathan Maletic

Chair, Department of Computer Science

Dr. Timothy Moerland

Dean, College of Arts and Sciences

TABLE OF CONTENTS

TABLE OF CONTENTS	III
LIST OF FIGURES	IX
LIST OF TABLES	XIII
ACKNOWLEDGEMENTS	XV
CHAPTER 1 INTRODUCTION.....	1
1.1 Research Focus.....	4
1.2 Research Goal	6
1.3 Contributions.....	6
1.4 Organization	7
1.5 Publication Notes	8
CHAPTER 2 METHOD STEREOTYPES – NANO PATTERNS OF SOFTWARE	
DESIGN	9
2.1 Related Work.....	10
2.1.1 Stereotype Definitions for Development.....	12
2.1.2 Stereotype Definitions for Applications.....	13
2.2 Method Stereotype Taxonomy	14
2.3 Reverse Engineering Method Stereotypes	18
2.3.1 Identification Rules	18
2.3.2 XPath Queries to Automatically Identify Method Stereotypes.....	21
2.4 Evaluation of the Approach.....	33

CHAPTER 3 CLASSIFYING SOFTWARE USING METHOD STEREOTYPES. 35

3.1	Overview and Motivation.....	36
3.2	The Method Stereotype Distribution of Systems	37
3.2.1	Stereotype Perspective	40
3.2.2	Stereotype Category Perspective.....	43
3.3	System Classification Findings	46
3.3.1	Mutator & Mutator-Data Storage Patterns	48
3.3.2	Mutator-Collaborator Patterns.....	48
3.3.3	Accessor, Mutator-Collaborator Patterns	49
3.3.4	Accessor-Collaborator Pattern	50
3.3.5	Accessor-Mutator-Controller Pattern	50
3.3.6	Controller-Collaborator Pattern.....	51
3.4	Automatic Clustering	51
3.4.1	Hierarchical Clustering	52
3.4.2	Partitioning.....	54
3.5	Threats to Validity.....	56
3.6	Related Work.....	56
3.7	Discussion	58
CHAPTER 4 AUTOMATIC IDENTIFICATION OF CLASS STEREOTYPES.... 60		
4.1	Overview and Motivation.....	61
4.2	Class Signature	63
4.2.1	Method Stereotypes.....	63

4.2.2	Method Stereotype Distributions	64
4.3	Taxonomy of Class Stereotypes	67
4.4	Automatically Identifying Stereotypes.....	73
4.4.1	Rules for Class Stereotype Identification.....	73
4.4.2	Implementation.....	79
4.5	Evaluation.....	80
4.6	Empirical Study.....	83
4.7	Threats to Validity.....	86
4.8	Related work	87
4.9	Conclusions	91
CHAPTER 5 COMMIT CATEGORIZATION – HIGH-LEVEL PERSPECTIVE		
 OF THE SYSTEM CHANGES OVER THE HISTORY		93
5.1	Overview and Motivation.....	94
5.2	Defining Commit Signatures.....	96
5.2.1	Method Stereotypes.....	97
5.2.2	Commit Signature	98
5.3	Commit Categorization	100
5.4	Reverse Engineering Commit Types.....	107
5.4.1	Design Changes during Evolution.....	108
5.4.2	Rules to Identify Commit Types	109
5.4.3	Implementation.....	111
5.5	The Case Study.....	112

5.6	Applying the Approach – Commit Labeling.....	116
5.7	Threats to Validity.....	118
5.8	Related work	119
5.9	Conclusions	121
CHAPTER 6 EVOLUTION OF METHOD STEREOTYPES.....		123
6.1	Motivation	123
6.2	Case Study.....	124
6.3	Results and Observations	125
6.3.1	Evolution of Stereotype.....	126
6.3.2	Stereotype Distribution Stability.....	130
6.3.3	Stereotype Distribution vs. Release Types.....	133
6.4	Patterns discovered.....	134
6.4.1	HippoDraw	134
6.4.2	QuantLib.....	135
6.5	Discussion	137
CHAPTER 7 CONCLUSIONS.....		139
7.1	Contributions.....	139
7.2	Future work	141
APPENDIX A METHOD STEREOTYPES DISTRIBUTIONS IN THE		
CLASSIFIED SYSTEMS.....		143
A.1	Distributions for systems in Mutator Pattern	143
A.1.1	Stereotype Category Perspective.....	143

A.1.2	Stereotype Perspective	144
A.2	Distributions for systems in Mutator-Data Storage Pattern	145
A.2.1	Stereotype Category Perspective.....	145
A.2.2	Stereotype Perspective	146
A.3	Distributions for systems in Mutator-Collaborator Pattern.....	147
A.3.1	Stereotype Category Perspective.....	147
A.3.2	Stereotype Perspective	148
A.4	Distributions for systems in Non-void-Mutator-Collaborator Pattern	149
A.4.1	Stereotype Category Perspective.....	149
A.4.2	Stereotype Perspective	150
A.5	Distributions for systems in Mutator-Accessor-Collaborator Pattern.....	151
A.5.1	Stereotype Category Perspective.....	151
A.5.2	Stereotype Perspective	152
A.6	Distributions for systems in Accessor-Mutator-Collaborator Pattern.....	153
A.6.1	Stereotype Category Perspective.....	153
A.6.2	Stereotype Perspective	154
A.7	Distributions for systems in Accessor-Collaborator Pattern.....	155
A.7.1	Stereotype Category Perspective.....	155
A.7.2	Stereotype Perspective	156
A.8	Distributions for systems in Accessor-Mutator-Controller Pattern.....	157
A.8.1	Stereotype Category Perspective.....	157
A.8.2	Stereotype Perspective	158

A.9	Distributions for systems in Controller-Collaborator Pattern	159
A.9.1	Stereotype Category Perspective.....	159
A.9.2	Stereotype Perspective	160
APPENDIX B SOURCE CODE OF HIPPODRAW CLASSES REDOCUMENTED		
 BY THE <i>STEREOCODE</i> AND <i>STEREOCLASS</i> TOOLS.....		
B.1	Entity - class Range.....	161
B.2	Minimal Entity - class Point.....	165
B.3	Data Provider (and Entity) - class BinnerAxis.....	166
B.4	Commander (and Boundary) - class DrawBorder.....	169
B.5	Boundary (and Data Provider) - class DataView	171
B.6	Factory - class QtViewFactory.....	175
B.7	Controller - class DisplayController	176
B.8	Pure Controller (and Factory) - class BinnerAxisXML.....	176
B.9	Pure Controller (and Small) - class AxisTickXML.....	177
B.10	Large Class (and Boundary) - class FunctionController.....	178
B.11	Lazy Class - class BinsBase	178
B.12	Degenerate Class – class AxisRep2D	180
B.13	Data Class - class AxisTick.....	181
RERERENCES		182

LIST OF FIGURES

Figure 1. Research focus.....	5
Figure 2. XPath query for the <i>get</i> method stereotype.	22
Figure 3. Rules to automatically identify the <i>get</i> method stereotype.	22
Figure 4. XPath query for the <i>predicate</i> method stereotype.	23
Figure 5. Rules to automatically identify the <i>predicate</i> method stereotype.	23
Figure 6. XPath query for the <i>property</i> method stereotype.	24
Figure 7. Rules to automatically identify the <i>property</i> method stereotype.....	25
Figure 8. XPath query for the <i>void-accessor</i> method stereotype.....	25
Figure 9. Rules to automatically identify the <i>void-accessor</i> method stereotype.	26
Figure 10. XPath query for the <i>set</i> method stereotype.....	26
Figure 11. XPath query for the <i>set</i> method stereotype.....	27
Figure 12. XPath query for the <i>command</i> method stereotype.	27
Figure 13. Rules to automatically identify the <i>command</i> method stereotype.....	28
Figure 14. XPath query for the <i>non-void-command</i> method stereotype.....	29
Figure 15. XPath query for the <i>factory</i> method stereotype.....	29
Figure 16. Rules to automatically identify the <i>factory</i> method stereotype.	30
Figure 17. XPath query and the rules for the <i>collaborator</i> method stereotype.	30
Figure 18. XPath query for the <i>controller</i> method stereotype.....	31
Figure 19. Rules to automatically identify the <i>controller</i> method stereotype.	31

Figure 20. XPath query for the <i>incidental</i> method stereotype.	32
Figure 21. Rules to automatically identify the <i>incidental</i> method stereotype.	32
Figure 22. XPath query and the rules to automatically identify the <i>empty</i> method stereotype.....	33
Figure 23. The stereotype distributions for the systems <i>Qt</i> , <i>wxWidgets</i> , <i>ACE</i> . <i>Qt</i> and <i>wxWidgets</i> have a similar distribution with significant percentage of <i>command</i> , <i>property</i> , and <i>get</i> methods while <i>ACE</i> has a very different distribution e.g., <i>non-void-command</i> , <i>command</i> , and <i>property</i> are the most numerous methods.....	42
Figure 24. Stereotype category distribution shows different patterns: <i>Code::Blocks</i> and <i>KDevelop</i> are a Mutator-Collaborator driven IDE, while <i>HippoDraw</i> is an Accessor-Mutator-Controller driven application.....	45
Figure 25. Hierarchical clustering performed by COBWEB algorithm. Overall, the clustering produced the same similarity between systems as the manual classification.	53
Figure 26. Distribution of stereotypes for the classes <i>DataSource</i> and <i>DisplayController</i> signatures (from <i>HippoDraw</i>).....	65
Figure 27. Distribution of categories for the <i>DataSource</i> and <i>DisplayController</i> signatures (from <i>HippoDraw</i>).....	66
Figure 28. Class stereotypes and their signatures for 18 <i>HippoDraw</i> 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	70
Figure 29. Commit signature, i.e., the distribution of method stereotypes, for commit #496124 from Kate with 13 added/deleted methods. The numbers in the rectangles show counts of methods stereotypes participating in the commit.	99
Figure 30. Commit signature for the large commit #669042 from KSpread with 118 added/deleted methods.	100
Figure 31. The Commit Label for commit #496124 of Kate presents both a high-level and detailed view of the commit. The Commit Type and Commit Signature are shown for the entire commit. The Participating Classes are the individual classes that contributed to the Commit Signature. For each of these Participating Classes, the individual Class-Change Signatures are given.	117
Figure 32. The evolution of the projects size in terms of the number of methods through the releases analyzed.	126
Figure 33. Evolution of the <i>predicate</i> stereotype in the HippoDraw system.....	128
Figure 34. Evolution of of the <i>property</i> stereotype in the QuantLib system.	130
Figure 35. Method stereotypes distribution for the first-and-last releases of HippoDraw (* stands for the stereotype <i>collaborator</i>).	132
Figure 36. Method stereotypes distribution for the first-and-last releases of QuantLib (* stands for the stereotype <i>collaborator</i>).....	132
Figure 37. Method stereotypes distribution in two <i>bug fixes</i> releases of HippoDraw...	133

Figure 38. Stereotype Category distribution for the first releases of QuantLib (0.1.1). 137

Figure 39. Stereotype Category distribution for the last releases of QuantLib (0.9.7).. 137

LIST OF TABLES

Table 1. A taxonomy of method stereotypes.	15
Table 2. Rules for method stereotype identification.....	20
Table 3. An overview of the software systems examined ordered by the number of methods.....	38
Table 4. Distribution of method stereotypes across the 21 systems. Values are percentage of each method stereotype. Combinations of primary & secondary stereotypes are separate (e.g., <i>get</i> and <i>get-collaborator</i>).....	39
Table 5. Classification of systems based on signatures derived from the stereotype distribution. For each signature the classification includes the systems and the architecture types/domain of those systems.	47
Table 6. Systems frequently clustered together as found by X-Means over all 24 runs..	55
Table 7. Class stereotypes.....	68
Table 8. 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).	82
Table 9. An overview of the software systems evaluated in the empirical study. Ordered by the number of classes.....	84
Table 10. Distribution of class stereotypes across 5 open-source systems.....	85
Table 11. Commit types. Accessors are shown in green colors, mutators – in blue, factory – in tan, collaborative - in rose and turquoise, degenerate – in grey. The	

method stereotype has a grey shadow effect if ‘collaborator’ is a secondary stereotype for this method.	103
Table 12. Identification rules for commit categorization.....	110
Table 13. An overview of the software systems evaluated in the empirical studies. Ordered by the number of commits.	113
Table 14. Distribution of commit types across 4 open-source systems.	114
Table 15. The overview of the projects, HippoDraw and QuantLib, in the case study.	125

ACKNOWLEDGEMENTS

I am deeply grateful to my advisor, *Dr. Jonathan Maletic*, for his support and help, always being a great mentor, and allowing me the time and independence in finding my way of reaching the dissertation goal. He taught me how to develop and present ideas, encouraged me when I was stuck, and was always there when I needed his advice and assistance.

I would like to thank *Dr. Michael Collard* who I collaborated with in my research. I have learnt a great deal from him in conducting research and writing papers. I would like to thank all my colleagues from the <SDML> lab for discussing and sharing new ideas and research thoughts.

I am indebted to my Dissertation Committee for the valuable feedback and everyone in the Department of Computer Science at Kent State University who contributed towards the completion of the dissertation and whose names I did not reveal.

This thesis would not have been possible without great support and patience from my husband, *Feodor*, and my kids, *Nick* and *Maria*. I would like to express special gratitude towards my parents, *Nadejda and Nicolai*, for teaching me how to be persistent, patient and motivated in achieving important goals.

Natalia Dragan

December 2010, Kent, Ohio

CHAPTER 1

INTRODUCTION

A wealth of standard abstractions, in the form of generic solutions, idioms, and patterns, predicated good Object-Oriented (OO) software design and modeling. This is particularly apparent in the concept of *design patterns* [Gamma et al. 1995], which articulate well-known good solutions to common OO design problems. Design patterns give names to these standard solutions and help form a vocabulary of OO design. A developer emboldened with the knowledge of design patterns (along with other well-known OO abstractions) can construct well-designed OO software much easier.

The work in this dissertation is focused on understanding an OO design abstraction but at a much lower level (than design patterns). Specifically, the concept of method and class stereotype is investigated. *Stereotypes* are generalizations that reflect some intrinsic or atomic behavior of a method or class. The notion of stereotype for OO modeling was first introduced by Wirfs-Brock [Wirfs-Brock 1993]. Initially, the main purpose was to support the classification of objects with respect to their roles and responsibilities in a software system. With the introduction of the Unified Modeling Language (UML) in the late 1990's, stereotypes became a powerful semantic extension mechanism, helping to increase the comprehensibility of UML diagrams.

Class/object identification is the central component of object-oriented modeling. In the conceptual object model, called the *analysis model*, three different standard class stereotypes can be used: *boundary*, *control*, and *entity* [Booch, Jacobson, Rumbaugh

1999]. A *boundary* class is used to model interaction between the system and its actors (i.e., users or external systems). An *entity* class represents the persistent information tracked by a system. A *control* class is used to model the dynamics of the system, and represent the tasks performed by a system: coordination, transactions, complex calculations, and business logic. In theory, every class in a system can be labeled with one of these three basic stereotypes.

However, in practice it is difficult to identify such clear-cut distinctions. A class must reflect nonfunctional requirements along with aspects of the solution domain. Moreover, they are completely specified and implemented using specific programming language syntax. Several approaches of object and class identification have been defined for requirements analysis and initial design phases. Examples that are particular to forward engineering include heuristics for object identification [Bruegge, Dutoit 2000], grammatical analysis [Abbott 1983], CRC methods [Beck, Cunningham 1989], and robustness analysis [Rosenberg, Scott 1999]. However, here a mechanism for reverse engineering that allows the identification of class's stereotypes in an existing software system is desired. The comprehension and understanding of classes (along with their methods and attributes) is a significant activity during the maintenance and evolution of software [Mayrhauser, Vans 1995], [Bennett, Rajlich, 73-87 2000] and is essential for many reverse engineering and design recovery research avenues.

Method stereotypes widely recognized by the development and maintenance communities and used in a number of well-known programming and data-structure textbooks [Deitel, Deitel 2001], [Salvitch 1999], [Stroustrup 2000], [Weiss 1999] include

constructor, *destructor*, *accessor*, *get*, *set*, *predicate*, and *mutator*. These are decades old terms that are commonly accepted. However, our empirical investigations of open-source software systems have demonstrated that diverse additional patterns of design at the method-level exist, i.e., many other method stereotypes are emergent. Additionally, as in the case with class stereotypes, a mechanism for reverse engineering method stereotypes from an implemented software system is wanted.

In practice, methods and classes are rarely documented with stereotypes, yet this information can be used to help infer the context of a class and how classes interact in a system. Having explicit knowledge of method and class stereotypes supports sophisticated types of design recovery and forms a foundation for a range of approaches based on stereotypes. For example, metrics based on class and method stereotype have a much more fine-grained perspective than commonly-used metrics (such as the number of methods in the class, the number of attributes in the class, the number of lines of code in the method, weighted methods per class, number of children, etc. [Chidamber, Kemerer 1991], [Lorenz, Kidd 1994], [Lanza 1999]) and include additional structural and semantic information at little cost. Changes to stereotypes (e.g., the method stereotype *get* in the version 0.1 became *predicate* in the version 0.2), automatically identified during evolution of a software project, may indicate major design changes to a class or system. Additionally, knowledge of class stereotypes in an implemented software system allows us to determine architectural importance for things such as automated layout of class diagrams or architectural-level understanding.

The dissertation proposes a technique to generate the knowledge of method and class stereotypes from an existing object-oriented software system. Initially, taxonomies of method (section 2.2) and class stereotypes (section 4.3) derived from an empirical investigation of a large number of open-source software systems are proposed. Based on the taxonomies, an approach is presented that allows one to analyze the structure and properties of each method and class in the system and automatically identify method and class stereotypes of the entire system (sections 2.3, 4.4). This knowledge of stereotypes permits the categorization of software at the system-level (Chapter 3). Additionally, method stereotypes are used to describe structural and behavioral changes of an object-oriented system during software evolution (Chapter 5, Chapter 6).

1.1 Research Focus

The dissertation focuses on automatically characterizing and classifying software at different abstraction levels - method, class, and system - into method stereotypes, class stereotypes, and system patterns. This understanding is monotonic, that is by knowing stereotypes of methods, one can integrate and abstract this to characterize class stereotypes, and likewise from the class to the system level. As can be seen in Figure 1, the foundational layer of the research is method stereotypes. The ability to automatically, and efficiently, reverse engineer method stereotypes from object oriented source code (specifically C++) has been demonstrated [Dragan, Collard, Maletic 2006].

OO design from the perspective of the source code is being empirically investigated. This bottom-up empirical investigation is focused on uncovering emergent patterns and relationships between stereotypes and high-level design. A means to automatically

identify method stereotypes given a taxonomy was developed and validated empirically [Dragan, Collard, Maletic 2006].

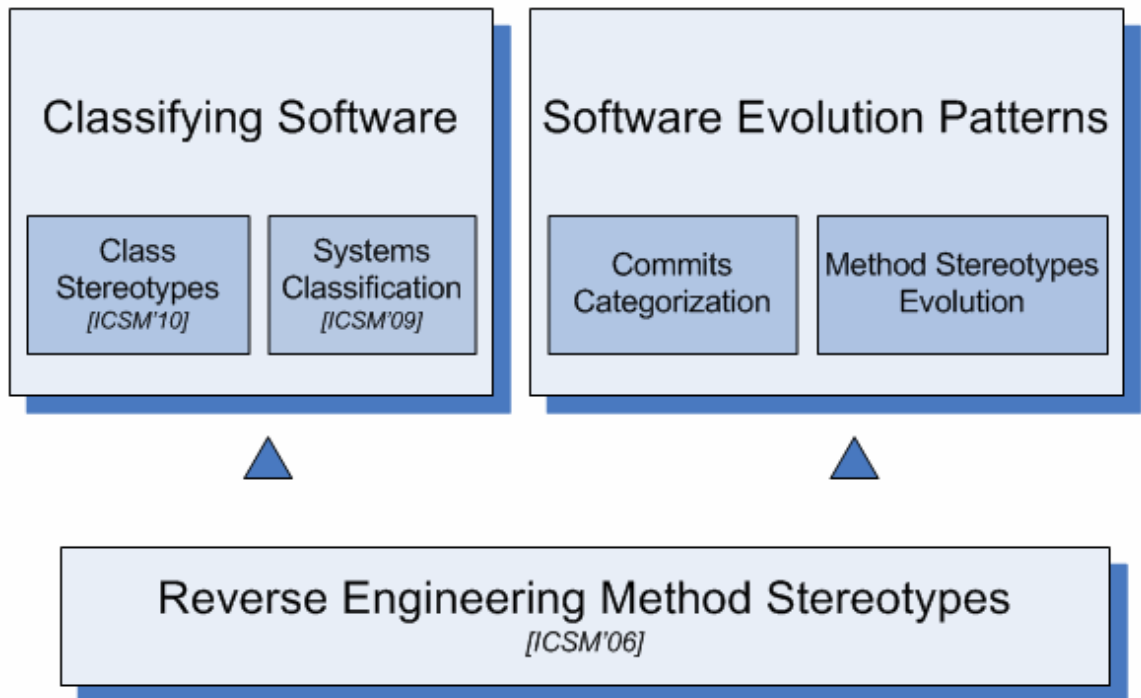


Figure 1. Research focus.

This work led to further exploration of stereotypes. The main questions of the research are as follows. Are method stereotypes indicative of class, system design, or design patterns? Are there emergent class stereotypes akin to the case of method stereotypes? Are there evolutionary patterns related to stereotypes? To answer these questions we have proposed methods and techniques which can be applied in maintenance, design recovery, and program comprehension.

1.2 Research Goal

The goal is to examine and develop new reverse-engineering approaches to uncover emergent laws of class and method stereotypes in object-oriented source code and their relationships to design, and to classify software at three different levels of abstraction: method, class, and system-level. This will bring benefits to re-documentation of source code, increase the common vocabulary of developers, and finally, assist in program comprehension and design recovery. Additionally, the information about stereotypes can be used to improve automated layout of UML class diagrams, and to construct metrics for the evaluation of design quality, testing efforts, design changes and their cost during software evolution.

1.3 Contributions

The general research contribution of this work is uncovering emergent laws of method and class stereotypes that were derived by analyzing data in existing object-oriented systems. The laws are as follows:

1. *Existence of stereotypes in practice.* Method and class stereotypes proposed in the literature exist not just in theory but in real life object-oriented applications.
2. *Increasing diversity of stereotypes.* The set of method and class stereotypes derived from empirical investigations includes not only stereotypes previously identified in the literature, but a much larger variety of stereotypes.
3. *Indicators of software design.* The stereotypes are indicative of particular software design solutions and architectures, and can be used to characterize and classify software at the method, class, and system-level.

4. *Descriptors of software evolution.* The stereotypes can describe design changes during software development and give a higher-level perspective of a system's evolution.

The first contribution is the taxonomic description of object-oriented method stereotypes [Dragan, Collard, Maletic 2006] and class stereotypes [Dragan, Collard, Maletic 2010]. These are the first comprehensive investigations on the topic of method and class stereotypes that reflect role, behavior, collaboration, and control features of methods and classes with respect to reverse engineering and design recovery. The second contribution is the extension of these approaches for method stereotype extraction and implementation of techniques for source code redocumentation, identification of descriptors for software systems and their classifications [Dragan, Collard, Maletic 2009], development of a tool for reverse engineering class stereotypes [Dragan, Collard, Maletic 2010], and implementation of a tool for the semantic categorization of commits. The final contribution is the evaluation of the approach by performing empirical studies on historical data for a wide range of open source object-oriented C++ software systems that can serve as a benchmark for further investigations and studies.

1.4 Organization

The dissertation is logically organized into three components: background information on reverse engineering method stereotypes, classification of classes and systems, and analysis of stereotypes evolution and semantic commit categorization. The dissertation is organized in the following manner. Chapter 2 gives an overview of reverse engineering method stereotypes. Chapter 3 presents classification of software at

the system level. Chapter 4 describes automatic identification of class stereotypes. Following that is the commit categorization in Chapter 6. The stereotype evolution is presented in Chapter 7. Conclusions are given in Chapter 8.

1.5 Publication Notes

Parts of this dissertation are extended versions of previously published papers. Chapter 2 is an extension of the Master Thesis [Dragan 2005] and portions of the chapter were published at the 22nd International Conference on Software Maintenance (ICSM 2006) [Dragan, Collard, Maletic 2006]. Partial results presented in Chapter 3 have been published at the 25th International Conference on Software Maintenance (ICSM 2009) [Dragan, Collard, Maletic 2009]. Chapter 4 is published at the 26th International Conference on Software Maintenance (ICSM 2010) [Dragan, Collard, Maletic 2010].

CHAPTER 2

METHOD STEREOTYPES – NANO PATTERNS OF SOFTWARE DESIGN

This chapter presents previous work dealing with method stereotypes [Dragan, Collard, Maletic 2006]. Method stereotypes are patterns of software design at a low level of abstraction - method level (i.e., nano patterns of design), and represent atomic blocks to design software. Automatic identification of method stereotypes forms the basis for much of the work in this dissertation, including reverse engineering class stereotypes, classifying software, and semantic commit categorization. The identification process is based on a proposed taxonomy of method stereotypes. This taxonomy is organized by main purpose and role of an object-oriented method while simultaneously emphasizing its creational, structural, behavioral and collaborational aspects with respect to a class's design.

Before the taxonomy is presented, the related work on method stereotypes is described in section 2.1. Then the taxonomy of method stereotypes is given, followed by section (2.3) on reverse engineering method stereotypes from C++ source code. The rules for method stereotypes identification along with their realizations as XPath queries are detailed in sections 2.3.1 and 2.3.2. Evaluations of the approach are presented in the final section.

2.1 Related Work

The notion of stereotype in object oriented modeling was first introduced by Wirfs-Brock to support the classification of objects in terms of assigning them certain features and properties [Wirfs-Brock 1993], [Wirfs-Brock, B., Wiener 1994]. Later, with the introduction of UML, stereotypes became a powerful extension mechanism in the UML for introducing new semantics to an existing model [Gogolla, Henderson-Sellers 2002], [Atkinson, Kuhne, Henderson-Sellers 2002], while increasing the comprehension of UML diagrams.

Work on UML class diagrams based on class stereotypes [Andriyevska et al. 2005], [Yusuf, Kagdi, Maletic 2007a], [Sharif, Maletic 2009] showed that layouts with additional semantic information in regard to the design were most effective, and the use of class stereotypes plays a significant role in comprehension of these diagrams. Here are a few more works on using stereotypes in UML: in comprehension of sequence diagrams [Genero et al. 2008]; in comprehension of class diagrams documented with Conallen's [Conallen 2002] stereotypes [Ricca et al. 2010].

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; 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 in a data member. We feel that a focus on the internal state is important (although 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.

Two directions of stereotype usage are described below (Sections 2.1.1 and 2.1.2). 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.

2.1.1 Stereotype Definitions for Development

Fowler [Fowler 2000] classifies methods at the design level (i.e., UML class) by concentrating on the object's state, with categories such as *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 [Stroustrup 2000] 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., [Deitel, Deitel 2001], [Salvitch 1999], [Tremblay, Cheston 2001], and [Weiss 1999]) 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's public methods and are not part of the class's interface.

Also to assist in program development, Riehle [Riehle, Berczuk 2001] 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* [Gamma et al. 1995] have been proposed and used. Template methods perform self-calls to abstract methods, while hook methods are designed to be overridden in subclasses.

2.1.2 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 [Workman 2002], 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. Visualization approach to support method understanding is proposed in [Robbes, Ducasse, Lanza 2005]. 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 by introducing three types of microprints, but the general characterization of a method with respect to its main role is not provided.

All of the stereotype definitions 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 previous work on the method stereotypes filled this gap in the method-stereotype classification, presented the taxonomy and an approach to automatically extract, and re-document, this information from the source code. We applied stereotypes to support the method's classification at the implementation level, i.e., annotating source code with precise method descriptors. Now, we investigate the emergent laws of class stereotypes and their benefits for reverse engineering, design recovery and program comprehension, as well as classifying software systems and categorizing commits based on nano patterns of software design, i.e., method stereotypes.

2.2 Method Stereotype Taxonomy

We unified the literature on method stereotypes by integrating the different perspectives given in the related work section while simultaneously addressing a number of deficiencies. The taxonomy (Table 1) is based on a method's main role and duties while emphasizing the creational, structural, behavioral and collaborative aspects with respect to a class's design. We categorized methods by the data access type (i.e., a method changes the objects state or leaves it constant) and their functionality, that is, their creational, structural, behavioral and collaborative characteristics.

Structural methods provide and support the structure of the class. For example, accessors read an object's state while mutators change it. Note that simple accessor and mutator methods `get` and `set` are only structural methods but other accessors/mutators additionally implement behavior of the class and are also characterized as *Behavioral*. *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* methods are where the behavioral or collaborative stereotypes occur in a minimal form. The name is based on the mathematical term for a case for which a stereotype cannot be any simpler, and indicate methods that are incomplete. For additional details and examples we point the reader to [Dragan, Collard, Maletic 2006].

Table 1. A taxonomy of method stereotypes.

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

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

Accessors are methods that do not change an object state (e.g., the `const` specifier in C++). The different stereotypes in the accessor category include:

- `Accessor::get` returns a data member.
- `Accessor::predicate` returns a Boolean value that is not a data member.
- `Accessor::property` returns some information about data members, and the method's return type is not Boolean.
- `Accessor::void-accessor` returns some information about data members through a parameter (method's return type is void).

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

- `Mutator::set` changes one data member and has a return type of void or Boolean
- `Mutator::command` performs a complex change to the object's state, e.g., more than one data member is changed and has a return type of void or Boolean. In the code we check the return type, and look for multiple assignments to data members.
- `Mutator::non-void-command` performs a complex change to the object's state and returns a value (i.e., is not void) of a non-boolean type. In the code we check the return type, and look for assignments to data members.

Creational methods include the stereotypes Constructor, Copy-Constructor, and Destructor that match the standard C++ language features. We restrict the consideration

of creational methods to the factory method because constructor, copy constructor, and destructor methods are well-known and are fairly easy and straightforward to identify. In fact most languages have specific syntax for these special-purpose methods and C++ is no exception. The remaining stereotype `Creational::factory` returns an object created in the method's body.

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

Collaborational methods work on an external object (of a different type) that is either a parameter or a local variable. Collaborational stereotypes include:

- `Collaborational::collaborator` is a method which works with an object that is a parameter, a local-variable, or a return-type. This is a secondary or primary stereotype.

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.

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

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

- `Degenerate::incidental` does not read or change an object's state, neither directly nor indirectly: it is a utility, an exception handler, or a candidate for overriding. This is a secondary or a primary stereotype.
- `Degenerate::empty` has no statements at all and perhaps is created with the eventual goal of overriding. This is a secondary or primary stereotype.

We now briefly discuss the tool developed to automatically extract method stereotypes. Note, in what follows we will omit the category name in the method stereotype name.

2.3 Reverse Engineering Method Stereotypes

Our tool, *StereoCode*, reverse engineers method stereotypes using lightweight static analysis and an infrastructure based on srcML (SouRce Code Markup Language) [Collard, Maletic, Marcus 2002], an XML representation supporting 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.

2.3.1 Identification Rules

Based on the 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 open source small simulation of a LAN network to illustrate good object-oriented design, Java, 20 classes), HotDraw [HotDraw 1999] (an open source two-dimensional graphics framework for structured drawing editors, Java, about 150 classes), and HippoDraw [HippoDraw] (an open source data analysis environment, C++, over 200 classes).

We now provide (Table 2) detailed rules for automatic identification of method stereotypes (in C++).

Table 2. Rules for method stereotype identification.

Accessors	
Structural	
get	<ul style="list-style-type: none"> • method is <code>const</code> • returns a data member • return type is primitive or container of a primitives
Behavioral	
predicate	<ul style="list-style-type: none"> • method is <code>const</code> • returns a Boolean value that is not a data member
property	<ul style="list-style-type: none"> • method is <code>const</code> • does not return a data member • return type is primitive or container of primitives • return type is not Boolean
void-accessor	<ul style="list-style-type: none"> • method is <code>const</code> • does not return a data member • return type is <code>void</code>
Mutators	
Structural	
set	<ul style="list-style-type: none"> • method is not <code>const</code> • return type is <code>void</code> or Boolean • only one data member is changed
Behavioral	
command	<ul style="list-style-type: none"> • method is not <code>const</code> • return type is <code>void</code> or Boolean • complex change to the object's state is performed e.g., more than one data member was changed
non-void-command	<ul style="list-style-type: none"> • method is not <code>const</code> • return type is not <code>void</code> nor Boolean • complex change to the object's state is performed e.g., more than one data member was changed
Creational	
factory	<ul style="list-style-type: none"> • returns an object created in the method's body
Collaborational	
collaborator	<ul style="list-style-type: none"> • returns <code>void</code> 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
controller	<ul style="list-style-type: none"> • no data members are written • no calls on data members • no calls <code>foo()</code>, <code>self::foo()</code>
Degenerate	
incidental	<ul style="list-style-type: none"> • no data members used • no calls (except for <code>std</code>) • includes at least one non-comment statement
empty	<ul style="list-style-type: none"> • no statements except for comments

2.3.2 XPath Queries to Automatically Identify Method Stereotypes

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 `bool` with `type='bool'`. For matching specific parts of a type the individual names can be used, e.g., matching "NTuple" in the type `const Ntuple&` can be matched with the expression `type/name='NTuple'`. This is true if at least one element `name` is equal to `NTuple`. All of the rules 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. Examples of the XPath queries that match function's definitions for all the method stereotypes are given below.

Get

The figure below gives the XPath realization of the detection rules for the method stereotype *get*.

```

src:function
[
  src:specifier='const' and
  not(src:type[src:name='void']) and
  src:return()
  [
    (
      count(*)=1 and src:name or count(*)=2 and
      *[1][self::op:operator='*'] and *[2][self::src:name]
    ) and
    src:primary_variable_name(src:name)[src:is_data_member()]
  ][1]
]

```

Figure 2. XPath query for the *get* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype get

(1) method is const
(2) return type is not void
(3) contains at least one return statement which is:
    a single variable
    pointer to a variable
    has no calls
    variable is a data member

-->

```

Figure 3. Rules to automatically identify the *get* method stereotype.

The third condition in Figure 3 checks whether the return statement is of the form ‘return dm’ or ‘return *dm’, where dm is a data member.

Predicate

The figure below gives the XPath realization of the detection rules for the method stereotype *predicate*.

```

src:function
[
  src:specifier='const' and
  src:type/src:name='bool' and
  (
    src:data_members() or
    (
      src:real_call()
      [
        src:is_pure_call() and
        (
          not(src:name/op:operator='::') or
          src:name/src:name[1]=src:class_name()
        ) or
        src:calling_object()[src:is_data_member()]
      ]
    )
  ) and
  src:return()[1] and not
  (
    src:return()
    [
      not(
        (
          *[2][self::op:operator] or src:call[1] or
          count(src:name)!=1 or
          src:primary_variable_name(src:name)
          [
            src:is_declared()
          ]
        )
      )
    ]
  )
]
]

```

Figure 4. XPath query for the *predicate* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype predicate

(1) method is const and
(2) return type includes bool and
(3) data members are used or there is a pure call or
    call on data members
(4) at least one return expression contains:
    false or
    true or
    call to another method or
    no variable, or more than one variable or
    there is variable plus an operator or
    one of the variables is not a data member

-->

```

Figure 5. Rules to automatically identify the *predicate* method stereotype.

The third condition checks if data members are used or calls on the class's methods are performed (the pure call in a form of `foo()` or the call on a data member in a form of `dm.foo()`). The fourth condition checks different situations to verify that the returned variable is not a data member, i.e., the method is not a *get* method.

Property

The figure below gives the XPath query for the method stereotype *property*.

```

src:function
[
  src:specifier='const' and
  not(src:type[src:name='void' or src:name='bool']) and
  (
    src:data_members() or
    (
      src:real_call()
      [
        src:is_pure_call() and
        (
          not(src:name/op:operator='::') or
          src:name/src:name[1]=src:class_name()
        ) or
        src:calling_object()[src:is_data_member()]
      ]
    )
  ) and
  src:return()[1] and not
  (
    src:return()
    [
      not(
        (
          *[2][self::op:operator] or src:call[1] or
          count(src:name)!=1 or
          src:primary_variable_name(src:name)
          [
            src:is_declared()
          ]
        )
      )
    ] [1]
  )
]

```

Figure 6. XPath query for the *property* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype property

(1) method is const
(2) return type is not void or bool
(3) data members are used or there is a pure call or
    call on data members
(4) return expression contains one of the following:
    more then one variable, or no variables
    a call
    single variable with an operator
    single variable that is not a data member

-->

```

Figure 7. Rules to automatically identify the *property* method stereotype.

The third and fourth conditions in Figure 7 check different situations to verify that data members are read and the returned variable is not a data member, i.e., the method is not a *get* method.

Void-accessor

The figure below gives the XPath realization of the detection rules for the method stereotype *void-accessor*.

```

src:function
[
  src:specifier='const' and
  (
    src:data_members() or
    (
      src:real_call()[src:is_pure_call() and
        (
          not(src:name/op:operator='::') or
          src:name/src:name[1]=src:class_name()
        ) or
      src:calling_object()[src:is_data_member()]]
    )
  ) and
  src:type/src:name='void'
]

```

Figure 8. XPath query for the *void-accessor* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype void-accessor
    specifier is const
    data members are used or there is a pure call or
        call on data members
    return is void
-->

```

Figure 9. Rules to automatically identify the *void-accessor* method stereotype.

The conditions are similar to the *property* method stereotype, except for the return type that is required to be `void`.

Set

The figure below gives the XPath realization of the detection rules for the method stereotypes *set*.

```

src:function
[
  not(src:specifier='const') and
  (
    src:type[src:name='void' or src:name='bool'] or
    count(src:return())=count
    (
      src:return()
      [
        count(*)=2 and *[1][self::op:operator='*'] and
        *[2][self::src:name='this']
      ]
    )
  ) and
  not(src:real_call()[2]) and
  count(src:data_members_write()) = 1
]

```

Figure 10. XPath query for the *set* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype set
    (1) method is not const
    (2) return type is void or bool, or return the object (for
chaining), i.e., 'return *this'
    (3) number of real calls in expression statements is at most 1
    (4) number of data members written to in expression
        statements is 1
-->

```

Figure 11. XPath query for the *set* method stereotype.

For this method stereotype we allow only one call (otherwise it is the *command* method) and the third condition checks this requirement.

Command

The figure below gives the XPath realization of the detection rules for the method stereotype *command*.

```

src:function
[
  not(src:specifier='const') and
  src:type[src:name='void' or src:name='bool'] and
  src:union(src:data_members_write(), src:type)
  [
    last()>2 or last()=2 and src:real_call()[2] or
    last()=1 and src:real_call()
    [
      src:is_pure_call() and not(src:is_static()) or
      src:calling_object()[src:is_data_member()][1]
    ][1]
  ][1]
]

```

Figure 12. XPath query for the *command* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype command
(1)method is not const
(2)return type contains void or bool
(3)for expression statements at least one of the following holds:
    -more then one data member is written to
    -exactly one data member is written to and the number of
      calls in expression statements or returns is at least 2
    -no data members are written to and there is a call
      -not in a throw statement that is a simple real call
        (not a constructor call)
      -or a complex call for a data member
-->

```

Figure 13. Rules to automatically identify the *command* method stereotype.

The third condition checks if less or more than one data member is written (if one - then it is the set method stereotype) and if *real* calls in the form of `foo()` or `dm.foo()` exist (where `dm` is a data member, i.e., no calls like `throw()`, `assert()`, `new()`, `static_cast()`, `dynamic_cast()`, or `const_cast()`). Note, a set is formed by `src:union` with the written data members and the `src:type`. This way the predicate is always evaluated, even if there are no data members written. The actual number of data members written is one less than `last()`. So, "`last()=1` and `...`" is evaluated when there are no data members written.

Non-void-command

The figure below gives the XPath realization of the detection rules for the method stereotype *non-void-command*.

```

src:function
[
  not(src:specifier='const') and
  not(src:type[src:name='void' or src:name='bool']) and
  src:union(src:data_members_write(), src:type)
  [
    last()>2 or
    last()=2 and src:real_call()[2] or
    last()=1 and src:real_call()
    [
      src:is_pure_call() and not(src:is_static()) or
      src:calling_object()[src:is_data_member()][1]
    ][1]][1]
]

```

Figure 14. XPath query for the *non-void-command* method stereotype.

The XPath query above checks the same conditions as for the *command* method stereotype except that the return value is not `bool` or `void` (could be a flag of the type `integer`).

Factory

The figure below gives the XPath realization of the detection rules for the method stereotype *factory*.

```

src:function
[
  src:type
  [
    type:modifier='*' and
    src:name[src:is_object()]
  ] and
  src:return()
  [
    op:operator='new' or
    src:primary_variable_name(src:name)[src:is_declared()]
  ][1]
]

```

Figure 15. XPath query for the *factory* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype factory

    return type includes a pointer to the object
    a return statement includes
        the 'new' operator or
        a variable which is a parameter or
        a local variable
-->

```

Figure 16. Rules to automatically identify the *factory* method stereotype.

The XPath query in Figure 15 checks if an object created is returned.

Collaborator

The figure below gives the XPath realization of the detection rules for the method stereotype *collaborator*.

```

<!-- stereotype collaborator

    A type name is an object, but not of this class
-->
src:function
[
    src:all_type_names_nonclass_object(../src:type/src:name,
        src:class_name())[1]
]

```

Figure 17. XPath query and the rules for the *collaborator* method stereotype.

This XPath query checks if an object, but not of this class's type, is used as a parameter, local variable or return type.

Controller

The figure below gives the XPath realization of the detection rules for the method stereotype *controller*.

```

src:function
[
  not(src:one_data_members_write()) and not
  (
    src:real_call()
    [
      src:is_pure_call() and
      (
        not(src:name/op:operator=':::') or
        src:name/src:name[1]=src:class_name()
      ) or
      src:calling_object()[src:is_data_member()]
    ]
  ) and
  (
    src:one_real_call() or
    src:expr_name()
    [
      src:is_written() and
      src:primary_variable_name(.)
      [
        not(src:is_data_member())
      ]
    ]
  ] or
  src:block//src:decl
  [
    src:type/src:name[src:is_object()]
  ][src:init]
  )
]

```

Figure 18. XPath query for the *controller* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```

<!-- stereotype controller

(1) method is not const
(2) no data members are written
(3) (
    (one or more calls:
        -no pure calls, a() a::b()
        -no calls on data members)
    or parameter or local variable is written
  )
-->

```

Figure 19. Rules to automatically identify the *controller* method stereotype.

The third condition checks that the calls performed are not class's method calls (*pure calls*) or calls on data members. Calls allowed are in form $f \rightarrow g()$, where f is not a data member, or `new f()` (which is not a *real call*).

Incidental

The figure below gives the XPath realization of the detection rules for the method stereotype *incidental*.

```
src:function
[
  src:block/*[not(self::src:comment)][1] and
  (
    count(src:block/src:return) +
    count(src:block/src:throw) + count
    (
      src:block/src:expr_stmt
      [./src:expr/src:call/src:name='assert']
    )
  ) =
  count(src:block/*[not(self::src:comment)]) and
  not(src:block/src:return//src:name) and
  not(src:data_members())
]
```

Figure 20. XPath query for the *incidental* method stereotype.

The XPath query above checks for the following conditions to be met in source code:

```
<!-- stereotype incidental
  (1) includes at least one non-comment statement
  (2) no real calls (including new calls)
  (3) no data members used
-->
```

Figure 21. Rules to automatically identify the *incidental* method stereotype.

The second condition does not allow any calls except for `assert()` or `throw()`.

Empty

The figure below gives the XPath realization of the detection rules for the method stereotype *empty*.

```

<!-- stereotype empty
      no statements, except for comments
-->

src:function
[
  not (
    src:block/*[not(self::src:comment)][1]
  )
]

```

Figure 22. XPath query and the rules to automatically identify the *empty* method stereotype.

This XPath query checks if the method has only comments in its body.

2.4 Evaluation of the Approach

The initial evaluation of the taxonomy and the *StereoCode* tool was performed for one system and reported in [Dragan 2005]. Later, more systems were added, and the taxonomy and the tool was validated [Dragan, Collard, Maletic 2006]. The developer assessment demonstrated that our stereotype classification along with our tool for automatically identifying and re-documenting method stereotypes was both sound and efficient. Our results were very good as an experienced developer agreed 90% of the time with our classification. *StereoCode*, which is based on a lightweight static program analysis approach, was 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.

Overall, the assessment of this work was performed on 28 open-source systems (in all the case studies performed for the dissertation work). The initial taxonomy and the identification rules [Dragan 2005] for reverse engineering method stereotypes were refined and expanded. The validations performed have 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, the tool redocumented systems correctly.

We now leverage this work to support a number of more complex design recovery. In the following chapter software at the system level is analyzed. The frequency and distribution of method stereotypes in an object-oriented system are used to classify software systems into architectural categories. The hypothesis that the relative frequency of particular method stereotypes within a system is any indication of a system's design/architecture is investigated by examining 21 open source systems along their method stereotype profile.

CHAPTER 3

CLASSIFYING SOFTWARE USING METHOD STEREOTYPES

In this chapter, we present an approach to classify software systems into architectural categories based on the frequency and distribution of method stereotypes in an object-oriented system. Having the stereotype for each method in a system, we now seek to understand how these design nano patterns are used to build software systems and how we can characterize software at the system level.

First the stereotype for each method is determined using the taxonomy defined in Chapter 2. The counts of the different stereotypes form a *signature* of the system and these signatures are used as input for a clustering algorithm. Determining method stereotypes is done automatically and is based on language (C++) features, idioms, and the main role (purpose) of a method. An empirical study of 21 open source systems is used for the evaluation of the approach. The results show that the distribution of method stereotype is an indicator of system architecture.

In the next section reasons and objectives of software classification are presented. In section 3.2 we present the distributions of method stereotypes for the 21 systems. Section 3.3 presents analysis of this data and a manual classification of the systems based on all the knowledge we had on hand. Following that are the results of applying two clustering algorithms to the data with threats to validity given in 3.4. Related work and conclusions follow.

3.1 Overview and Motivation

We demonstrated the ability to automatically, and efficiently, reverse engineer method stereotypes [Dragan, Collard, Maletic 2006] from object oriented source code (specifically C++). In practice, methods are rarely documented with stereotypes (e.g., get, set, predicate, etc), yet we feel this information can be used to help infer the context of a class and how classes interact. That is, good method abstraction is typically a requirement for good class abstraction. Our hypothesis is that we need to understand the methods before we can infer the role and design of a class.

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

In order to verify our hypothesis, we investigate if the relative frequency of particular method stereotypes within a system is any indication of a system's design/architecture. The work presented here examines 21 open source systems along their method stereotype profile. We developed a tool to automatically label each method with its corresponding stereotype and then calculate the total number of each for the entire system. The distribution of this data forms a method stereotype signature for the system. These signatures are used as input into clustering algorithms, and the results are compared against a manual classification.

Our findings indicate that method stereotypes and how they are distributed within a system is a good indicator of the type of design/architecture being used by the software. The empirically derived result supports, to a large degree, our hypothesis and the importance of method stereotypes.

3.2 The Method Stereotype Distribution of Systems

We now examine the frequency of each different stereotype that occurs within a system. We will then use this distribution to compare and classify systems.

Twenty-one C++ open-source software systems are used in the study and listed in Table 3, ordered by the number of methods for each system. The systems were chosen to represent a wide range of a sizes, problem domains, and architectures. Half of the systems occur in Bjarne Stroustrup's list of interesting C++ applications¹, while others were taken from sourceforge.net. The main categories of the systems are: GUI library (*wxWidgets*, *Qt*, *SmartWin++*); Game Programming library and SDK (*FlightGear*, *PPTactical*, *CEL*, *CrystalSpace*, *ClanLib*); Graphics library, 2D & 3D Engines, Image Drawing (*Ivf++*); Mathematical and Finance libraries (*CGAL*, *QuantLib*, *C++Fuzzy*); Development and Communication Environments (*KDevelop*, *Code::Blocks*, *ACE*, *Ice*); Testing, Management and Unicode frameworks (*CppUnit*, *ICU*, *OpenWBEM*); and

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

complete applications (*Doxygen*, *HippoDraw*). For the most part, these systems can be considered good examples of object oriented design.

Table 3. An overview of the software systems examined 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

For each system we automatically determined labeled the stereotype of each method using the StereoCode tool. The time to extract the stereotypes from a system ranged from a few seconds for C++Fuzzy to a bit less than five minutes for Qt. The resulting

distribution of method stereotypes for each system is given in Table 4. The degenerate secondary stereotypes for all the primary stereotypes are combined with the accessor/mutators due to the very small percentages they represent.

Table 4. Distribution of method stereotypes across the 21 systems. Values are percentage of each method stereotype. Combinations of primary & secondary stereotypes are separate (e.g., *get* and *get-collaborator*).

	get	get collaborator	predicate	predicate collaborator	property	property collaborator	void-accessor	void-accessor collaborator	accessor degenerate	set	set collaborator	command	command collaborator	non-void-command	non-void-command collaborator	mutator degenerate	factory	collaborator	collaborator degenerate	controller	controller degenerate	unclassified
ACE	1.1	1.7	0.5	0.1	3.3	7.6	2.0	0.8	0.2	1.4	2.1	5.0	10.2	9.1	36.9	1.5	4.0	2.7	2.0	5.3	0	2.6
C++Fuzzy	1.9	8.9	0	0.6	12.8	9.9	0	4.2	3.2	3.5	4.5	3.8	24.9	2.2	5.1	0	2.6	0.6	0	8.6	0	2.6
CEL	3.4	5.7	0.9	0.7	0.5	3.9	0.1	0.2	1.4	5.1	3.6	10.0	36.3	0.1	9.6	2.9	3.6	0	3.2	6.9	1.4	0.2
CGAL	1.8	7.0	2.3	7.4	1.4	34.7	0.7	2.1	1.1	1.7	2.3	5.6	12.4	0.7	7.7	0.6	0.4	2.9	0.7	5.6	0	1.0
ClanLib	3.1	2.0	3.0	1.5	6.4	6.5	0.3	0.8	2.0	4.5	1.5	18.0	23.1	3.1	7.0	2.4	1.9	2.2	1.3	6.8	0	2.6
Code:Blocks	3.8	5.0	1.4	1.3	1.0	5.5	0.1	0.6	0.5	2.4	2.9	12.5	39.0	2.2	9.8	1.5	2.9	1.3	1.6	3.8	0	0.8
CppUnit	3.0	3.5	1.4	0.6	2.5	2.8	0.7	0.6	0.5	2.9	2.0	22.5	33.1	0.9	7.3	3.6	1.2	1.6	2.4	4.4	0.1	2.3
CrystalSpace	3.6	7.8	0.7	1.1	1.1	5.1	0.2	1.2	1.3	2.4	3.8	8.0	32.6	1.4	11.3	2.0	2.9	2.8	4.0	6.1	0	0.8
Doxygen	1.1	3.7	1.5	0.3	0.9	2.7	0.2	0.2	0.1	2.4	1.6	5.5	9.9	0.4	16.2	1.7	0.9	1.0	39.9	9.4	0	0.3
FlightGear	14.2	5.7	0.7	0.4	3.8	1.5	0.1	0.2	0.3	10.3	4.7	19.6	18.0	3.3	5.1	2.5	1.2	1.3	0.2	5.8	0	1.0
HippoDraw	5.1	3.3	2.1	2.1	7.0	8.2	1.3	1.8	5.9	3.4	0.9	11.2	20.8	1.7	4.3	2.1	9.5	0.2	1.5	7.1	0	0.5
Ice	3.4	6.0	1.2	3.2	1.7	7.0	1.1	3.7	1.4	1.8	3.2	8.9	33.5	0.6	10.5	1.4	2.0	1.1	2.0	5.6	0.1	0.5
ICU	0.6	9.8	0	0	0.5	16.1	0.2	3.2	1.6	0.5	2.1	2.9	32.1	1.3	14.9	0.3	7.9	1.8	0.6	3.4	0	0.4
Ivf++	6.6	4.0	0	0.4	0	0	0.1	0.4	0	8.1	5.4	25.8	20.0	3.0	3.1	5.3	0.2	0.2	2.2	13.2	0	1.8
KDevelop	2.9	6.4	1.2	1.1	0.8	5.3	0.1	0.4	0.7	2.8	2.6	12.6	45.7	0.6	7.4	1.0	1.8	1.2	1.4	3.4	0	0.5
OpenWBEM	1.0	3.5	1.9	2.0	1.0	11.1	0.2	7.2	0.6	1.8	2.7	4.7	34.6	1.7	13.4	2.9	1.6	0.9	2.7	3.7	0	0.8
PPTactical	6.0	5.2	0.2	0.1	0.9	0.7	0	0	0.1	5.7	6.4	13.6	35.4	3.5	8.7	2.6	1.4	0.5	3.0	4.8	0	1.3
Qt	2.3	5.4	3.4	2.3	3.2	16.6	0.3	1.2	1.9	2.1	2.6	9.9	28.3	1.1	7.0	1.2	3.0	3.0	1.2	3.6	0	0.5
QuantLib	0.9	20.0	0.8	2.6	1.5	33.8	1.4	3.7	3.3	1.3	3.2	2.4	14.3	0.6	4.5	0.3	2.2	1.1	0.4	1.4	0	0.4
SmartWin++	0.5	8.0	0.6	4.7	1.4	18.9	0.1	2.0	1.7	0.7	3.3	4.3	21.8	0.8	18.9	0.2	0.3	5.2	0.5	5.2	0	0.9
wxWidgets	3.6	7.5	3.5	2.3	2.0	7.8	0.7	1.0	2.0	3.1	3.1	11.8	31.1	1.2	7.1	2.4	3.0	0.9	2.0	3.0	0	0.6
Min	0.5	1.7	0	0	0	0	0	0	0	0.5	0.9	2.4	9.9	0.1	3.1	0.0	0.2	0.0	0.0	1.4	0	0.2
Max	14.2	20.0	3.5	7.4	12.8	34.7	2.0	7.2	5.9	10.3	6.4	25.8	45.7	9.1	36.9	5.3	9.5	5.2	39.9	13.2	1.4	2.6
Mean	3.3	6.2	1.3	1.7	2.6	9.8	0.5	1.7	1.4	3.2	3.1	10.4	26.5	1.9	10.3	1.8	2.6	1.5	3.5	5.6	0.1	1.1
Stdev	3.0	3.8	1.1	1.8	3.0	9.6	0.5	1.8	1.4	2.4	1.3	6.6	10.1	1.9	7.4	1.3	2.3	1.2	8.4	2.6	0.3	0.8

This distribution represents the signature of a system. It describes the degree of prevalence of static structures such as collaboration and state change. To better interpret

the data visually, we organize it at two different levels to highlight the logical structure of the code, i.e., the method and class levels. At the method level, *stereotype perspective*, we grouped by primary stereotype. This reflects the role of a method as it ignores, to a large degree, interaction with other classes. The alternative class level perspective, *stereotype category perspective*, highlights the degree of coupling and collaboration among classes in a system, along with some internal coupling (cohesion) of a class through accessors/mutators. Additionally, parts of the system not yet implemented (degenerate) are reflected. These two different perspectives complement each other and highlight different aspects of a system's design/architecture. That is, it presents a view of the method alone and a view of how the methods collaborate inter- and intra-class. These two perspectives (or slices of the data) will be discussed in more details now.

3.2.1 Stereotype Perspective

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

As an example, the distributions of three systems, *Qt*, *wxWidgets*, and *ACE*, are given in Figure 23. As can be seen, *Qt* and *wxWidgets* have a very similar stereotype distribution in this method level perspective, while *ACE* exhibits a very different distribution. We generated pie charts for all the 21 systems and observed a number of trends. All 21 charts are presented in Appendix A and are constructed from Table 4. The

main observations are discussed here while an analysis of the trends is given in section 3.3.

As can be calculated from Table 4 get-methods are prevalent in all 21 systems with distribution varying from 2.8% to 20.9%. The systems *FlightGear* and *QuantLib* have the highest percentage of *get* methods (~20%). The majority of the systems have a larger percentage of the get-methods than set-methods (2.6% to 15%). Few systems contain many predicates.

Command methods occur in of all studied systems, and the systems KDevelop, Code::Blocks, CppUnit, CEL, Pptactical, and Ivf++ contain a very large percentage (around 50%). ACE is the only system with a significant percentage of non-void-command methods. Collaborators occur infrequently in all systems except for Doxygen.

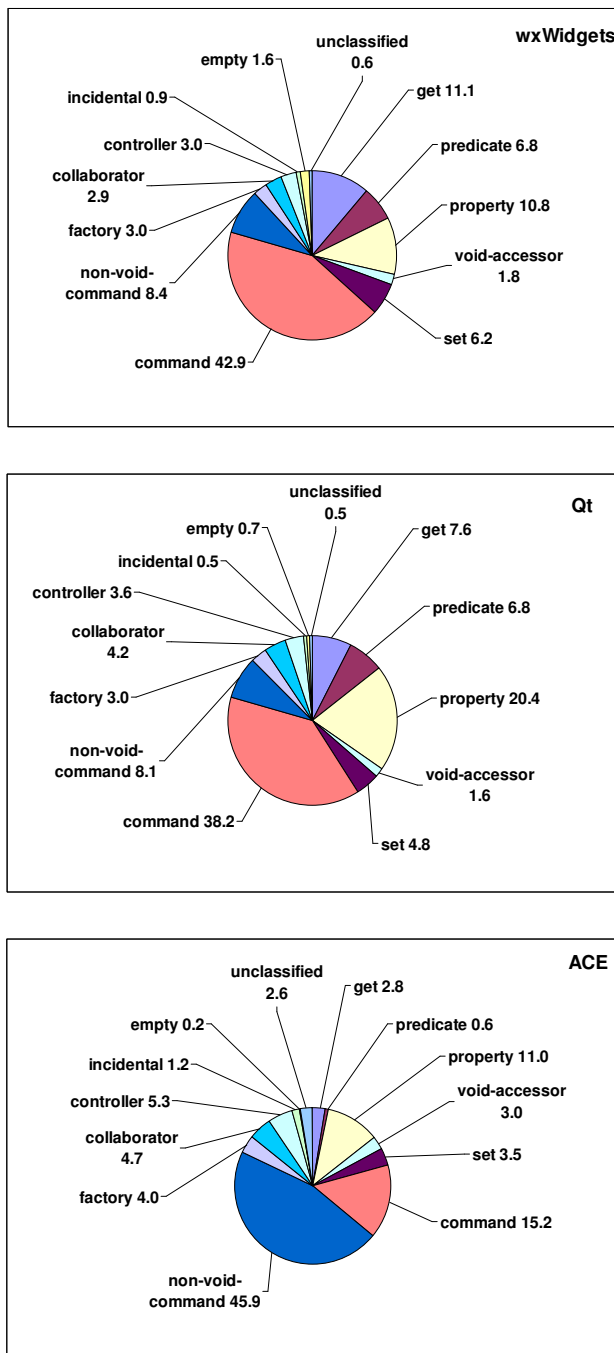


Figure 23. The stereotype distributions for the systems *Qt*, *wxWidgets*, *ACE*. *Qt* and *wxWidgets* have a similar distribution with significant percentage of *command*, *property*, and *get* methods while *ACE* has a very different distribution e.g., *non-void-command*, *command*, and *property* are the most numerous methods.

Approximately 40% of the system is actually collaborator degenerate methods (i.e., do not read or change an object's state).

Controller and factory are also prevalent stereotypes with maximum values 13.2% and 9.5%, respectively. *HippoDraw* has a high percentage of both. A few systems (mostly game frameworks) have a large percentage of controller methods, namely *C++Fuzzy*, *Ivf++*, *CEL*, *ClanLib*, *CrysrtalSpace*, *FlightGear*, *Doxygen*. We now examine the stereotype category perspective in similar fashion.

3.2.2 Stereotype Category Perspective

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

An example of the stereotype category perspective for *Code::Blocks*, *KDevelop*, and *HippoDraw* is given in Figure 24. We see that *Code::Blocks* and *KDevelop* have very similar distributions, while *HippoDraw* is very different. Again we generated pie charts for all the 21 systems and present the observations here while an analysis of the trends is given in section 3.3.

We observed that most of the 21 systems have more mutators than accessors, sometimes as much as three times more. Only two systems, *QuantLib* and *CGAL*, have

more accessors than mutators. In most of the cases the systems with mutators as the most prevalent, have more collaborators than non-collaborators. However, there are exceptions, such as *ICU* with 91.8% collaborators and *HippoDraw* with 59.3%. Four systems have nearly equal non-collaborational methods.

HippoDraw has a very different distribution compared to the other systems. Accessors, mutators, controller, and factory are evenly distributed. There is also an almost equal distribution of collaborational subcategories compared to non-collaborational.

Given these two abstract perspectives we now analyze the trends and describe a set of patterns that seem to correlate with architecture and/or domain.

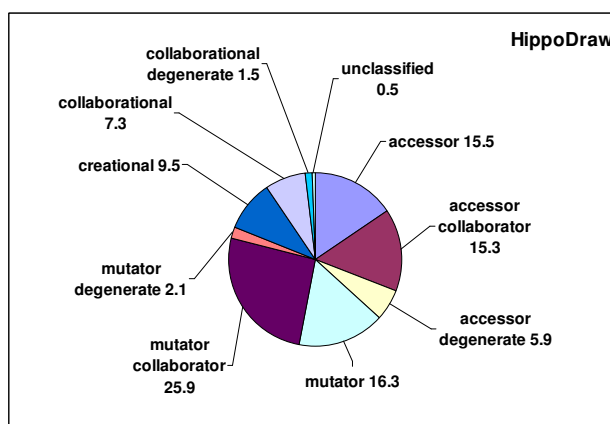
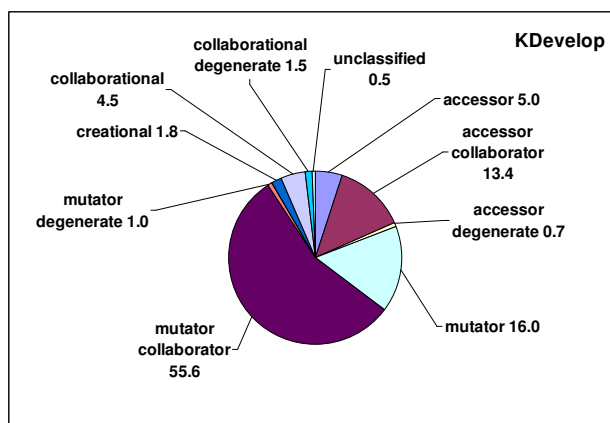
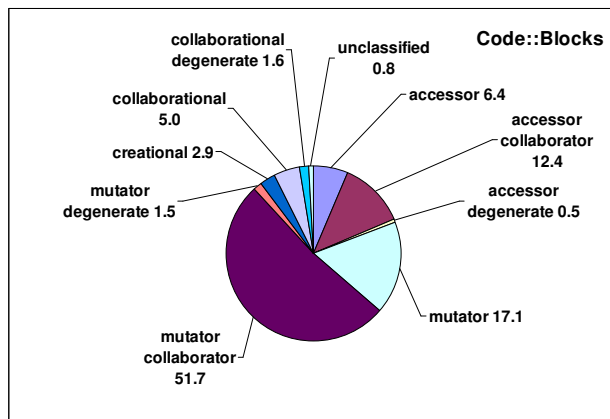


Figure 24. Stereotype category distribution shows different patterns: *Code::Blocks* and *KDevelop* are a Mutator-Collaborator driven IDE, while *HippoDraw* is an Accessor-Mutator-Controller driven application.

3.3 System Classification Findings

Given the trends we observed in the previous section we now classify systems based on these two views of the method stereotype distribution. We use knowledge of the domain and architecture/design of the 21 systems (see Table 3) and correlate this manually with common trends and patterns observed in the method and class level perspectives of the distributions. This manual categorization will be used to evaluate the results of automatic clustering techniques in the following section.

Through manual inspection of the distributions we uncovered common trends within groups of similar (with respect to domain and/or architecture) systems. We describe these trends as common patterns of stereotype distribution. The results are presented in Table 5. The pattern name relates to the most dominant attributes of the method stereotype distribution. For each pattern the systems are listed that exhibit the pattern along with the prevailing architecture and problem domain.

The architecture for each system is derived from the description provided at the project site and the purpose/role of the system. They are classified into the categories of application, framework, SDK, and library.

We now discuss each distribution pattern in the context of the distribution and the systems that exhibited the pattern.

Table 5. Classification of systems based on signatures derived from the stereotype distribution. For each signature the classification includes the systems and the architecture types/domain of those systems.

Pattern	Classification		
	Systems	Architectures	Domain
Mutator	CppUnit, ClanLib	SDK/framework	game and unit-testing
Mutator-DataStorage	Ivf++, FlightGear	framework/application	visualization and game
Mutator-Collaborator	KDevelop, Code::Blocks, PPTactical	framework/application	IDE, game
Non-void-Mutator-Collaborator	ACE	framework	communication environment
Mutator-Accessor-Collaborator	Ice, OpenWBEM, CEL, CrystalSpace	SDK/framework	internet communications engine, management of systems, game and 3D graphics
Accessor-Mutator-Collaborator	wxWidgets, Qt, SmartWin++, ICU	library/framework	GUI, Unicode component
Accessor-Collaborator	CGAL, QuantLib, C++Fuzzy	library	math and finance
Accessor-Mutator-Controller	HippoDraw	application	data analysis environment
Controller-Collaborator	Doxygen	application	documentation system

3.3.1 Mutator & Mutator-Data Storage Patterns

For these patterns mutators form a large part of the distribution. While collaborator subcategories exist, they do not dominate over the non-collaborator categories, i.e., the stereotype set-collaborator occurs less than half as often as the stereotype set. The accessors do exist, but in much smaller numbers. The systems that fall into this category are *CppUnit*, *ClanLib*, *FlightGear* and *Ivf++*. External collaborators, i.e., stereotypes factory, collaborator and controller, which work only on external objects, are typically insignificant, except for *Ivf++*.

However there are significant features which make the systems *Ivf++* and *FlightGear* different from *CppUnit* and *ClanLib*, the stereotypes get and set play a more significant role, so we denote systems *Ivf++* and *FlightGear* with the more specific Mutator-DataStorage pattern. The domain of these systems includes unit-testing, visualization and game frameworks (SDK).

3.3.2 Mutator-Collaborator Patterns

Similar to the mutator pattern previously described, this pattern has a substantial portion (65 to 75%) of mutators. However, unlike the mutator pattern, the percentage of the secondary stereotype collaborator is also very high. The percentage of predicate, property, controller, collaborator (primary), and factory is quite low in this pattern.

The systems that contain this distribution pattern are *KDevelop*, *Code::Blocks*, *PPTactical*, and *ACE*. The domains of these systems are IDEs (*KDevelop* and *Code::Blocks*), a game (*PPTactical*), and a communication environment (*ACE*). All have quite similar distributions. The only one that stands out is *ACE*, which has stereotype

non-void command in significant (about 45%) numbers, and property plays a more significant role. We put this system in the separate pattern termed Non-void-Mutator-Collaborator.

3.3.3 Accessor, Mutator-Collaborator Patterns

In this pattern, accessors and mutators are nearly equal in distribution, but mutators are still the dominant stereotype. Collaborators are more prevalent than the equivalent non-collaborators. Additionally, the external collaborators, controller and factory, are significant.

The systems that contain this pattern include *Ice*, *OpenWBEM*, *CEL*, *CrystalSpace*, *wxWidgets*, *Qt*, *SmartWin++*, and *ICU*. This covers a wide variety of different domains including GUI libraries/framework, Unicode-component library, a game SDK, 3D graphics SDK, a communication environment, and an information system framework.

One difference among them is the degree of collaboration. Most of these systems have 2-4 times larger percentages of collaborators compared to non-collaborators. However, for *ICU* and *SmartWin++* the number of collaborators is 10-20 times larger than the number of non-collaborators.

The distribution of accessors and mutators is also not the same for all systems. *Qt*, *SmartWin++*, and *ICU* have close percentages of both, while *Ice*, *OpenWBEM*, *CEL*, *CrystalSpace*, and *wxWidgets* have a lower percentage of accessors. However the last five systems belong to this group because accessors still play a more significant role than in the Mutator-Collaborator group of systems.

Overall this large group can be divided into two subgroups. In the subgroup containing *Ice*, *OpenWBEM*, *CEL*, and *CrystalSpace*, collaborators play a more essential role as well as mutators with respect to accessors. Because of a significantly larger number of mutators, we call such systems Mutator-Accessor-Collaborator. The other subgroup contains *wxWidgets*, *Qt*, *SmartWin++*, and *ICU*, where accessors and mutators are more evenly distributed, and we call this pattern Accessor-Mutator-Collaborator.

3.3.4 Accessor-Collaborator Pattern

For this pattern accessors form a significant part of the distribution. Of all the accessors, the stereotypes `get` and `property` are the most prevalent and have the largest percentage in the distribution. The systems that fall into this category are *CGAL*, *QuantLib*, and *C++Fuzzy*. They are all math and finance libraries. Note that in these systems the stereotype `collaborator` is also quite prevalent.

3.3.5 Accessor-Mutator-Controller Pattern

In this pattern, accessors, mutators, and external collaborators are evenly distributed. The term `controller` is used in the name of this pattern because external collaborators control the behavior of external classes. In addition, a comparison of `collaborator` and `non-collaborator` shows an equal distribution. The number of `controller` and `factory` stereotypes is large.

The only system that falls into this group is *HippoDraw*, which is a data analysis application. However, if we take into account the method-level view, this system is

closest to GUI systems *wxWidgets* and *Qt*, and based on the overall distribution it is similar to *ClanLib*.

3.3.6 Controller-Collaborator Pattern

For this pattern the collaborators are the dominant stereotypes and the percentage of controllers is high. Of the collaborators, a large number are collaborator-degenerate. There is a low percentage of accessors (around 11%), and mutators form about 1/3 of the system. The only system that fits this pattern is *Doxygen*, which is a documentation application.

We now apply clustering algorithms to the method stereotype distribution data and compare the results with our manual classification.

3.4 Automatic Clustering

The manual classification done in the previous section took advantage of domain and architectural knowledge of the systems. Here we investigate if the method stereotype distribution (signature) alone can be used to classify systems into meaningful groups. To verify this we apply an automatic clustering technique to the raw distribution data of the 21 systems. The results of the clustering algorithms are compared to the manual classification done in the previous section.

Clustering is an unsupervised learning technique that allows grouping of similar entities or the discovery of common patterns. Degree of similarity can be calculated using a variety of distance measures. Here we use Euclidean distance because our data is already normalized as percentages. There are a number of well-studied clustering

algorithms and they are typically divided into two main groups, hierarchical and partitioning. Hierarchical clustering algorithms distribute input instances into a hierarchy of clusters while partitioning distributes into distinct clusters.

Hierarchical clustering is used to determine the similarity of systems based on their signatures, however, it does not always classify systems into distinct groups. Partitioning is used to identify the distinct groups but does not indicate the degree of similarity of systems. The combination of both types of clustering is validated against the manual classification.

3.4.1 Hierarchical Clustering

First we perform hierarchical clustering using the COBWEB [Fisher 1987] algorithm to determine the similarity between systems without having to set a predefined number of clusters. The input to the clustering algorithm is the signature of systems contained in the data from Table 4, i.e., method stereotypes distribution in percentages. The result of the clustering is shown on Figure 25.

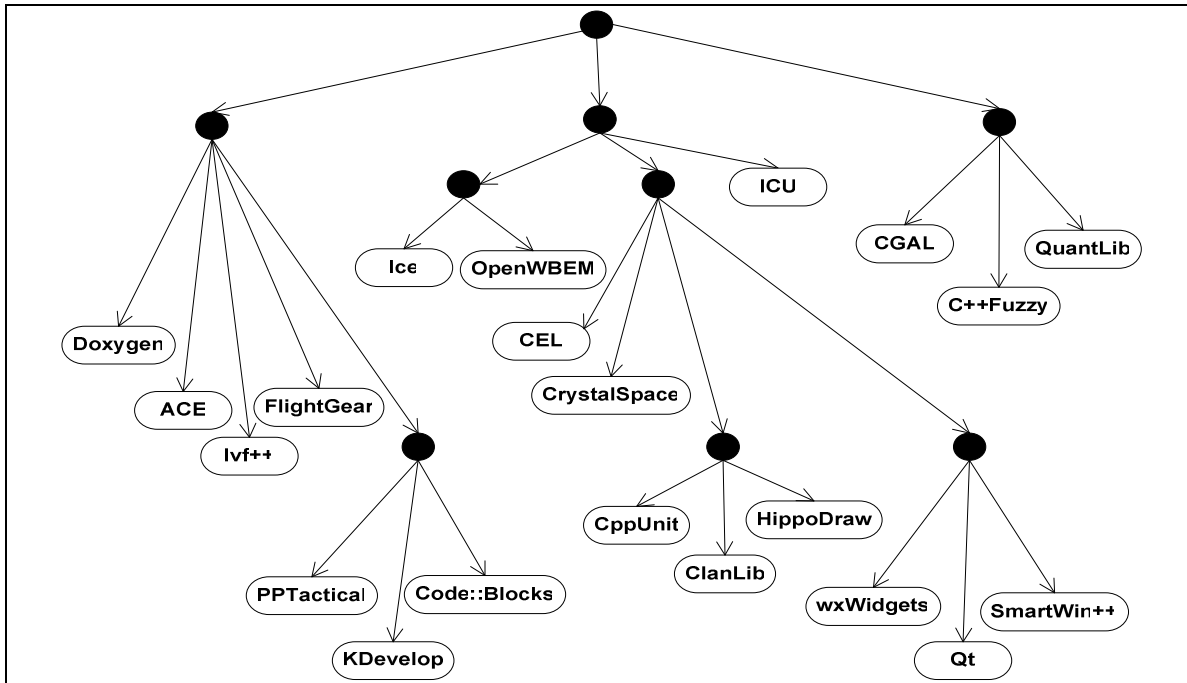


Figure 25. Hierarchical clustering performed by COBWEB algorithm. Overall, the clustering produced the same similarity between systems as the manual classification.

The hierarchical clustering produced three large clusters. The first cluster consists of the systems *Doxygen*, *ACE*, *Ivf++*, *FlightGear*, *KDevelop*, *Code::Blocks*, and *PPTactical*. The systems *KDevelop*, *Code::Blocks*, and *PPTactical* are in one sub-cluster, which implies that they have a more similar distribution than the others (e.g., *Ivf++*, *Doxygen*). This cluster consists of IDEs, games, and other systems close to an IDE in their distribution. This clustering is in agreement with the manual classification given in Section 3.3.

The second large cluster consists of the systems *Ice*, *OpenWBEM*, *CrystalSpace*, *CEL*, *ClanLib*, *CppUnit*, *HippoDraw*, *wxWidgets*, *Qt*, *SmartWin++*, and *ICU*. In this large cluster there are five sub-clusters.

All these systems fall into the same groups as the manual classification, except for *HippoDraw*. From our observations this system should stand by itself; however it has similarity to *wxWidgets*, *Qt* and, to a lesser degree, *ClanLib*. This second cluster consists of GUI libraries, a GUI framework, games, and other systems closest to GUI in their distribution. The third small cluster consists of *C++Fuzzy*, *CGAL*, and *QuantLib*. Math and finance libraries are instances of this cluster.

Overall, the automatic clustering produced very similar results to the manual classification given in Section 3.3, indicating that classification based on stereotype signature patterns is indicative of the important aspects of the distribution.

3.4.2 Partitioning

To validate the specific groups in our manual classification, we performed X-Means partitional clustering [Pelleg, Moore 2000], which is an extension of K-Means clustering. In addition to the distribution data, X-Means clustering requires as input a range for the expected number of clusters. The normal practice is to run the algorithm multiple times with different ranges and seeds, and then assess the multiple runs. We ran it 24 times with ranges of six to eight clusters and used different seeds.

Table 6. Systems frequently clustered together as found by X-Means over all 24 runs.

Systems	Clustered Together
KDevelop, Code::Blocks	100%
wxWidgets, Qt	100%
CGAL, QuantLib	83%
ClanLib, CppUnit	83%
Ice, OpenWBEM	83%
Crystal Space, CEL	67%
Ivf++, FlightGear	67%
Ice, OpenWBEM, CrystalSpace	67%
CEL, Crystal Space, KDevelop, Code::Blocks	67%
Crystal Space, Ice, OpenWBEM, KDevelop, Code::Blocks	67%

Each run produces a set of clusters. The number of times systems are clustered together is calculated across all runs. From the 24 runs 39 different clusters (more accurately item-sets) had optimal evaluation parameters (i.e., minimal distortion and maximal BIC-values). Of the 39, ten occurred in at least two-thirds of the runs with different input seeds. The percentage of times particular systems were clustered together is given in Table 6. For example, the systems *wxWidgets* and *Qt* were clustered together in every run. This is a strong indication that these two systems are very similar in their distribution, but distinct from the other systems. Other systems clustered together infrequently - less than a third of the time - e.g., *wxWidgets*, *Qt*, and *HippoDraw*, indicating that these three systems should not be classified together.

Ice, *OpenWBEM*, *CEL*, and *CrystalSpace* clustered together in two-thirds of runs with *Code::Blocks* and *KDevelop*. While their distributions are similar, the manual classification and hierarchical clustering seem to be in more agreement.

3.5 Threats to Validity

The assessment of classification using method stereotypes is subject to a number of threats to validity in both the collection of the stereotypes and the classification analysis.

For the collection of the stereotype data, our tool uses lightweight source code analysis. The conversion to the srcML format may produce incorrect markup that may cause the misidentification of a stereotype. However, the srcML format has been successfully used for querying and fact extraction of C++ source code and transformation (refactoring). The XPath expressions used to find stereotypes may misidentify a stereotype, and leave some methods unclassified. However, our previous evaluation of the tool showed high correlation to manual assignment of stereotypes.

For each system, the architecture/domain had to be determined based on a written description of the project. For automatic clustering, the choice of clustering algorithm and the chosen similarity measure can affect the results. However, the results from applying both hierarchical clustering and partitioning are complementary.

3.6 Related Work

Analysis of software with respect to architectural/design patterns on the coarse- and fine-grained levels (such as method-, class-, package- and system-level) and the evolution of these patterns have been investigated by many researchers [Gamma et al. 1995], [Gil, Maman 2005], [Lanza, Ducasse 2001b], [Arevalo, Ducasse, Nierstrasz 2003a], [Robbes, Ducasse, Lanza 2005], [Workman 2002], [Kim, Pan, Whitehead 2006], [Dong, Godfrey 2007], [Dong, Godfrey 2008].

Analysis of design patterns at more fine-grained levels, such as method- and class-level, is performed by many researchers. Method-level patterns are given in [Arevalo, Ducasse, Nierstrasz 2003a], [Robbes, Ducasse, Lanza 2005] and [Workman 2002]; class-level design patterns are presented in [Gil, Maman 2005], [Lanza, Ducasse 2001b], and [Clarke, Malloy, Gibson 2003]. The work presented by Dong et al is the closest to our work in terms of the granularity level. They present a hybrid model reverse engineered at a coarse-grained level, such as package diagrams, and identify architectural change patterns during software evolution [Dong, Godfrey 2007], [Dong, Godfrey 2008].

The main directions of applying clustering techniques in software engineering are the following: software architecture recovery [Maqbool, Babri 2007]; identification of subsystem structures by clustering procedures or methods into modules [Hutchens, Basili 1985], [Montes de Oca, Carver 1998], and modules or classes into subsystems [Anquetil, Fourier, Lethbridge 1999], [Mancoridis et al. 1999], [Mitchell, Mancoridis 2006], measuring differences between clustered objects and comparing clustering algorithms [Tzerpos, Holt 1999], [Koschke, Eisenbarth 2000].

Other reverse-engineering approaches for program comprehension have been identified in [Lanza 2003]: dynamic execution traces can be inspected, though it may take too much time, and the large volume of information about thousands of methods calls hide important facts, making it easy to get lost. The version history can be analyzed. However, this is more useful for understanding the past history of software or prediction of evolution. The source code and documentation can be examined. This is an

extensively used non-automated practice, but very often the documentation analyzed is not up to date, is inaccurate, or is not present at all, and it may take too much time.

The approach proposed in our work is to use lightweight static analysis of the source code and clustering techniques to solve the problem of classifying software, and then utilize this approach for system re-documentation. Clustering algorithms are used to group systems based on method stereotypes distribution.

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

3.7 Discussion

The results show that the frequency and distribution, across a system, of the method stereotypes described by our taxonomy is a good indicator of system architecture/design. That is, the method stereotype signature for a system can be used to automatically cluster systems with similar architectures together. Manual classification of the system using visual inspection of the signatures along with knowledge from the system documentation further supported this result.

For example, the two IDEs we studied, *Code::Blocks* and *KDevelop*, are grouped together. To explain this we surmise that there is underlying reference architecture for IDEs that both systems follow. We also saw this strong grouping with the two GUI frameworks we studied, *Qt* and *wxWidgets*. While these examples are not terribly

surprising, the result of having the methods stereotype signature reflect this so clearly is of particular interest.

We plan to extend this work by studying the change of signatures over the evolution of a system. This will investigate whether the signature of a system changes during the development of a project, and at what point the signature become stable.

This chapter presents an application of the method stereotypes to characterize and classify software at the system level. In the next chapter, we again leverage the work on method stereotypes; this time they are used to recover design information at the class level. Class stereotypes are automatically identified from source code based on the frequency and distribution of the method stereotypes in the class.

CHAPTER 4

AUTOMATIC IDENTIFICATION OF CLASS STEREOTYPES

In this chapter, we describe an approach to automatically determine a class's stereotype. Having the stereotype for each method in a system and observing patterns of design at a system level, we now try to find how the nano patterns of design are used to build object-oriented classes and how we can characterize software at the class level. Class stereotypes represent *micro patterns* [Gil, Maman 2005], i.e., design patterns at the class level. They are similar to *design patterns* [Gamma et al. 1995], but micro patterns are found at a lower level of abstraction.

The class's stereotype is based on the frequency and distribution of the method stereotypes in the class. The method stereotypes are automatically determined using the defined taxonomy. 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.

The chapter is organized as follows. The next section motivates automatic identification of class stereotypes. Section 4.2 contains a description of a class *signature*

[Dragan, Collard, Maletic 2009] which is based on method stereotypes distributions. Additionally, how we compute the method stereotype distributions is described. The class signature forms the input for our automatic classification scheme. In Section 4.3 we present a taxonomy of class stereotypes. Section 4.4 describes our approach to automatically identify class stereotypes from existing C++ code. Section 4.5 is an evaluation of the approach as compared to experts, followed by an empirical study in Section 4.6. This is followed by a discussion of threats to the validity of our approach, related work, and conclusions.

4.1 Overview and Motivation

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 simple for a small number of classes but doing so for entire systems would be 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 design patterns. A class's stereotype maybe an indicator of bad smells and gives clues for refactoring. It can be an indicator of a class's comprehensibility.

A number of studies [Andriyevska et al. 2005; Genero et al. 2008; Kuzniarz, Staron, Wohlin 2004; Ricca et al. 2010; Sharif, Maletic 2009; Staron, Kuzniarz, Wohlin 2006;

Yusuf, Kagdi, Maletic 2007b] demonstrate the benefits of using class stereotypes, which reflect semantics, in program comprehension, design, and software maintenance tasks. Using stereotype information [Andriyevska et al. 2005; Sharif, Maletic 2009; Yusuf, Kagdi, Maletic 2007b], as a factor in laying out UML class diagrams, has shown to improve the comprehensibility of the diagram. Staron et al. [Staron, Kuzniarz, Wohlin 2006] 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 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 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.

4.2 Class Signature

Here we define a *class signature* 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 how we defined and automatically identified method stereotypes as this forms the basis for the signature. Specifics of the class signature are then presented.

4.2.1 Method Stereotypes

The aggregates for class signature identification are method stereotypes (see Table 1). Method stereotypes are reverse engineered using the tool, *StereoCode*, which re-documents source code with the stereotype information for each method and calculates totals per class.

The method stereotypes are logically organized by categories: *creational*, *structural*, *behavioral* and *collaborational*. *Structural* methods provide and support the structure of the class and include accessor and mutator subcategories. Most of the methods from the structural category are also *behavioral* – they define behavior of the class (predicate, property, void-accessor, command, and non-void-command). *Creational* methods (only factory is considered here) create or destroy objects of the class. *Collaborational* methods (collaborator and controller) characterize the communication between objects and how objects are controlled in the system. *Degenerate* are methods where the structural or collaborational stereotypes are limited (incidental and empty).

The individual stereotypes and the above categories of stereotypes are used for defining the class signatures.

4.2.2 Method Stereotype Distributions

In Chapter 3 we presented the idea of a system signature and examined the frequency of method distributions for twenty-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 Figure 26. 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.

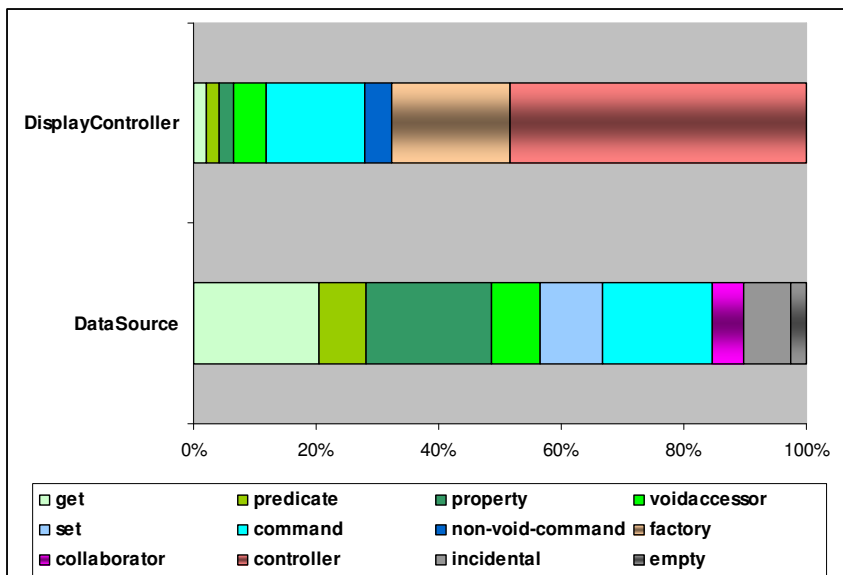


Figure 26. Distribution of stereotypes for the classes DataSource and DisplayController signatures (from HippoDraw).

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 Figure 27, 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 collaborative and there are no degenerate methods.

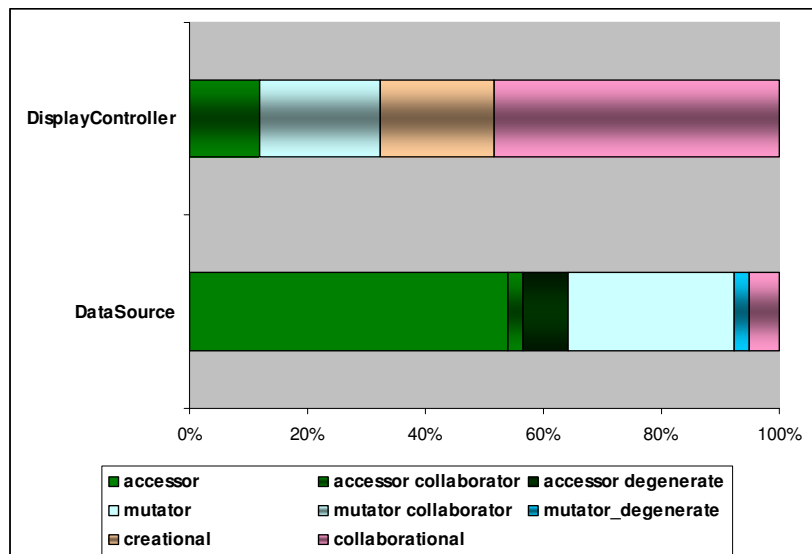


Figure 27. Distribution of categories for the DataSource and DisplayController signatures (from HippoDraw).

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 highlights 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 Figure 26 and Figure 27 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 collaborative.

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

4.3 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 3. 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 3. 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 7. The actual class names are not

Table 7. Class stereotypes.

Class Stereotype	Description	Candidate for a bad-smell class
Entity	Encapsulates data model	
Minimal Entity	Encapsulates trivial data model	
Data Provider	Encapsulates data	
Commander	Encapsulates behavior	
Boundary	Communicator in a system	
Factory	Objects ' creator	
Controller	Manager in a system	
Pure Controller	External data manager	X
Large Class	"Too much" responsibilities	X
Lazy Class	"Too little" responsibilities	X
Degenerate	Degenerate state and behavior	X
Data Class	Degenerate behavior	X
Small Class	Small number of methods	X

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.

Our initial taxonomy included the standard set of overarching stereotypes of *entity*, *boundary* and *control* class stereotypes [Booch, Jacobson, Rumbaugh 1999]. 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 [Dragan, Collard, Maletic 2006] and bad smells [Fowler 1999]. The list of class stereotypes uncovered is given in Table 7. 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 Figure 28). 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.

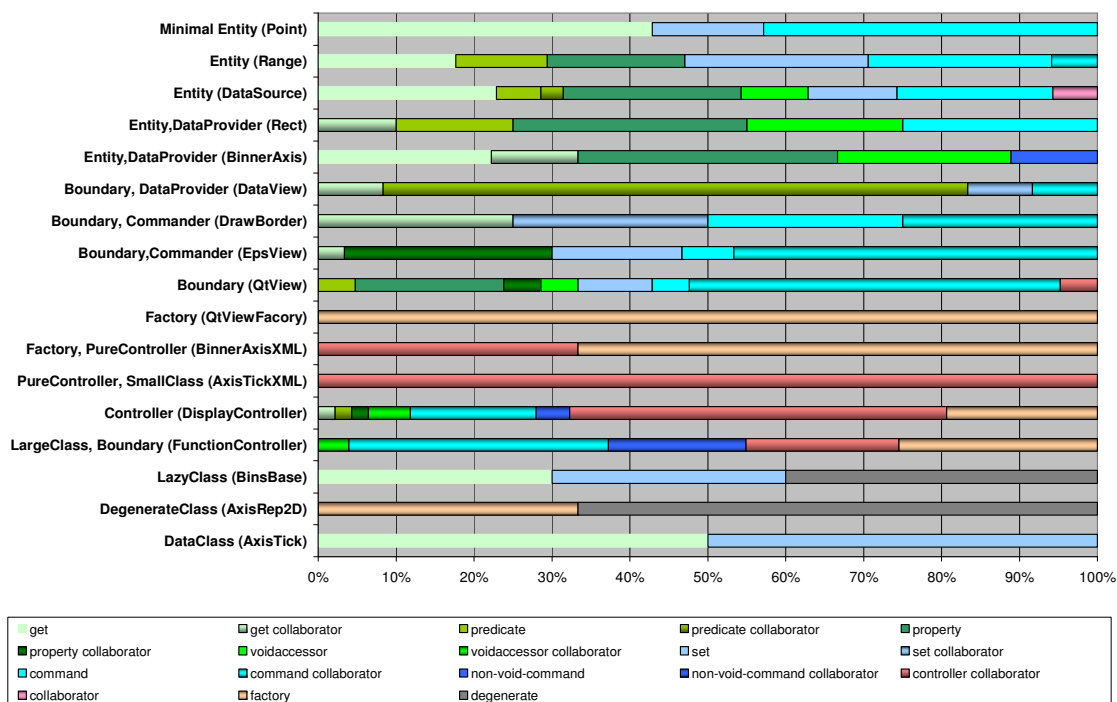


Figure 28. Class stereotypes and their signatures for 18 HippoDraw classes. Each row is labeled with the class stereotype(s) and in parentheses the name of the example class whose data is shown in the row. Each stereotype is automatically identified based on the signatures using the detection rules. 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 *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).

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 [Riel 1996]. 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” [Fowler 1999]. “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% collaborative.

A *Lazy Class* is a very trivial class which does “too little” [Fowler 1999]. 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 degenerates the state and/or behavior.

4.4 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 system in Table 3. 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).

4.4.1 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 [Lanza, Marinescu 2006] as a means to characterize and evaluate the design of object-oriented systems.

We use a fractional threshold of $\frac{2}{3}$ for representing situations where 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 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., *[Fuggetta]* 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, void-accessor empty, etc). The set $\{mutators\}$ is constructed in a similar way.

The set $\{collaborators\}$ consists of all the collaborative 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

$$|collaborators| / |non-collaborators| = 2$$

- 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

$$\{methods\} - (\{get\} \cup \{set\} \cup \{command\} \cup \{non-void-command\}) = \emptyset \ \& \ |get| \neq 0 \ \&$$

$$|set| \neq 0 \ \& \ (\{command\} \cup \{non-void-command\}) \neq \emptyset$$

- The ratio of collaborational to non-collaborational methods is 2:1

$$|collaborators| / |non-collaborators| = 2$$

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

- It consists mostly of accessors

$$|\text{accessors}| > 2 \cdot |\text{mutators}|$$

- Low control of other classes

$$|\text{accessors}| > 2 \cdot (|\text{controller}| + |\text{factory}|)$$

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

- It consists mostly of mutators

$$|\text{mutators}| > 2 \cdot |\text{accessors}|$$

- Low control of other classes

$$|\text{mutators}| > 2 \cdot (|\text{controller}| + |\text{factory}|)$$

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

- More collaborators than non-collaborators

$$|\text{collaborators}| > |\text{non-collaborators}|$$

- Not all the methods are factory methods

$$|\text{factory}| < \frac{1}{2} \cdot |\text{methods}|$$

- Low number of controller methods

$$|\text{controller}| < \frac{1}{3} \cdot |\text{methods}|$$

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

- It consists mostly of factory methods

$$|\text{factory}| > \frac{2}{3} \cdot |\text{methods}|$$

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

$$|\text{accessors}| \neq 0 \ \& \ |\text{mutators}| \neq 0$$

- Number of methods in a class is high

$$|\text{methods}| > \text{average} + \text{stdev}$$

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

$$|\text{get}| + |\text{set}| \neq 0$$

- It might have a large number of degenerate methods

$$|\text{degenerate}| / |\text{methods}| > 1/3$$

- Occurrence of other stereotypes is low

$$|\text{methods}| - (|\text{get}| + |\text{set}| - |\text{degenerate}|) \leq 1/5 \cdot |\text{methods}|$$

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

- It consists of many degenerate methods

$$|\text{degenerate}| / |\text{methods}| > 1/2$$

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

- Only the simple accessor/mutators get and set are present

$$|\text{get}| + |\text{set}| \neq 0 \ \& \ |\text{methods}| - (|\text{get}| + |\text{set}|) = 0$$

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

- Number of methods in a class is less than 3

$$|methods| < 3$$

4.4.2 Implementation

We extended our tool, StereoCode, to obtain class signatures and automatically identify the stereotypes. StereoCode automatically identifies method stereotypes using lightweight static analysis and an infrastructure based on srcML [Collard, Maletic, Marcus 2002], an XML representation of source code. StereoCode 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. The stereotype is assigned to a class if all conditions of the 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.

4.5 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 [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, 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 8.

The results obtained by the tool are shown in the first column. The tool labeled the 45 classes with 67 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 in those cases.

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. 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

Table 8. 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 ∩ Tool	S ₂	S ₂ ∩ Tool	S ₃	S ₃ ∩ Tool	(S ₁ ∪ S ₂ ∪ S ₃) ∩ Tool
Entity	13	13	8	4	3	9	7	10
Minimal Entity	3	1	1	0	0	2	0	1
Data Provider	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

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. Thus, tool support in this case will improve comprehension of class's design and role in the system.

4.6 Empirical Study

To further assess our approach we applied our tools to the five open source systems listed in Table 9. 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++ applications², 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 9. An overview of the software systems evaluated in the empirical study. Ordered by the number of classes.

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 of class stereotypes for each system is given in Table 10.

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

² www.research.att.com/~bs/applications.html

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.

Table 10. Distribution of class stereotypes across 5 open-source systems.

Stereotype	KDevelop		Code::Blocks		FlightGear		HippoDraw		QuantLib		Min (%)	Max (%)	Aver (%)	Stddev (%)
	#	%	#	%	#	%	#	%	#	%				
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
<i>Coverage</i>	98%		99%		95%		94%		99%		94	99	97	2.3

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.

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.

4.7 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.

4.8 Related work

The main objective of our work is to understand the role and main responsibilities of an object-oriented class and method in the system design. There are many dynamic, static or combined techniques used in reverse engineering which help to generate high-level views of the source code with the eventual goal of increasing comprehension. The following approaches to reverse engineer software entities have been widely used: metrics, visualization, concept analysis, clustering, information retrieval and webmining.

The number of methods in the class, the number of attributes in the class, the number lines of code in the method, the number of parameters in the method, etc. is counted using a metrics approach. A detailed list of object-oriented class and method metrics and metrics approaches for program comprehension and design quality are considered in details in [Chidamber, Kemerer 1994], [Lorenz, Kidd 1994], [Lanza 1999].

A metrics approach is generally used to assess software quality and performance, as “guidelines” for a good design or refactoring, and to predict maintainability [Fenton 1991], [Chidamber, Kemerer 1994], [Hitz, Montazeri 1995], [Henderson-Sellers 1996], [Briand, Daly, Wüst 1997], [Briand, Daly, Wüst 1999], [Demeyer, Ducasse, Nierstrasz

2000], [Demeyer, Ducasse, Lanza 1999], [Li, Henry 1993], [Basili, Briand, Melo 1996]. Antoniol et.al use metrics to automatically identify design patterns [Antoniol, Fiutem, Cristoforetti 1998a]. However, we are not able to grab the functionality and behavioral aspects of the class using this approach, i.e. taking into consideration only quantitative parameters of entities. The metrics approach is very straightforward, but alone does not work well for classification tasks or identification of method and class stereotypes in the source code.

The visualization approach helps to understand a software entity (method, class, package, system, etc) by representing entities and relationships between them graphically. Visualization tools as Rigi [Muller 1986], SeeSoft [Eick, Steffen, Summer 1992], ShrimpViews [Storey, Muller 1995], CodeCrawler [Lanza, Ducasse 2001a] are widely used in the research community.

The Concept Analysis mathematical technique is broadly used for program understanding. *Concept lattice* (or *Galois Lattice*) is constructed to represent source-code entities and their relationships. Applying concept analysis in reverse engineering and reengineering includes such research directions as: identification of modules in legacy systems [Siff, Reps 1999] and module restructuring [Tonella 2001], [Tonella 2003]; representation of the internal relationships between group of methods and attributes of a class [Arevalo, Ducasse, Nierstrasz 2003b]; location of features and concepts in the source code [Eisenbarth, Koschke, Simon 2003], [Poshyvanyk, Marcus 2007]; identification of changes in object-oriented software [Clarke, Malloy, Gibson 2003].

Understanding and reverse engineering design patterns [Gamma et al. 1995] has attracted a lot of attention from many researchers [Brown 1996], [Antoniol, Fiutem, Cristoforetti 1998b], [Antoniol, Fiutem, Cristoforetti 1998a], [Albin-Amiot et al. 2001], [Bieman et al. 2003], [Guéhéneuc, Sahraoui, Zaidi 2004], [Ng, Guéhéneuc 2007], [De Lucia et al. 2009], [Guéhéneuc, Guyomarc'h, Sahraoui 2010].

A lot of efforts have been spent to understand and detect bad-smell code at the method-, class-, and system-levels [Riel 1996], [Fowler 1999], [Brown et al. 1998], [Martin 2002], [Mäntylä 2003]. Riel defines 61 heuristics characterizing good object-oriented programming that allow engineers to assess the quality of their systems manually and provide a basis for improving design and implementation. *22 code smells* at the method- and class-level are defined in [Fowler 1999] suggesting to engineers applying refactorings. Brown et al. describe 40 *antipatterns*, i.e. general *design smells*. Mäntylä proposes a taxonomy for code smells. Automatic identification of problems in software design in general and particularly at the class level are presented in [Marinescu 2004], [Munro 2005], [Moha et al. 2008]. Note, in our work we do not detect bad-smell classes but take into consideration features of those classes for possible further refactoring.

The following work is closest to our work by the level of abstraction, i.e. understanding and reverse engineering patterns of design at the class level. A few approaches identify key or most important classes in a software system [Zaidman, Demeyer 2008], [Richner, Ducasse 2002], [Greevy, Ducasse 2005]. Zaidman et al. [Zaidman, Demeyer 2008] provide a mechanism based on dynamic coupling and

webmining to find classes with a lot of “control” within the application. Richner et al. [Richner, Ducasse 2002] present a tool based on dynamic information to support the recovery and understanding of collaborations between classes. Greevy et al. [Greevy, Ducasse 2005] 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 detailed description of roles/responsibilities for all the classes in a system and not only for “control” classes.

Gil et al. [Gil, Maman 2005] 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. [Clarke, Malloy, Gibson 2003]. Their approach does not reflect role and class responsibilities. A visualization approach to support quick class understanding is proposed by Lanza et al. [Lanza, Ducasse 2001b]. 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. Concept Analysis is used by Dekel to visualize the structure of the class in Java [Dekel, Gil 2003]. This approach describes the “context of a class“ showing relationships between methods based only on the state access.

All of the class categorizations given in the referenced work are primarily based on an 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.

4.9 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). A developers' assessment showed that our stereotype 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 [Fowler 1999; Riel 1996] 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.

The next chapter represents an extension of the approach of method stereotypes identification which characterizes commits into types based on the impact of the changes to a class (or classes). The stereotypes of the added and deleted methods form a descriptor of the change embodied by the given commit. These descriptors are then used to categorize commits. The objective is to gain a higher-level perspective of the changes to a system over its evolution history.

CHAPTER 5

COMMIT CATEGORIZATION – HIGH-LEVEL PERSPECTIVE OF THE SYSTEM CHANGES OVER THE HISTORY

In this chapter, we apply method stereotypes information to the evolution data of a software project. Individual commits to a version control system are automatically categorized based on the stereotypes of altered methods. The stereotype of each method is reverse engineered using the taxonomy and the StereoCode tool. The stereotypes of the added and deleted methods in a commit form a descriptor of the change embodied by the commit. These descriptors are then used to categorize commits, into types, based on the impact of the changes to a class or classes. The goal is to gain a higher-level perspective of the changes to a system over its history. A case study of four open-source project histories is presented to illustrate the potential benefits of this method. The case study also empirically investigates the distribution of the different commit types, common types, and the correlation with changes over the project's history.

The next section describes the benefits of commit categorization for development and maintenance activities. Section 5.2 contains a description of a commit signature that forms the input for automatic identification of commit types. In Section 5.3 we present the categorization of commits. Section 5.4 describes our approach to reverse engineer commit types from existing C++ code. Section 5.5 contains a case study of the approach.

An application of the approach in the form of commit labels is presented in Section 5.6. This is followed by a discussion of the threats to validity, related work, and conclusions.

5.1 Overview and Motivation

Version control systems, such as *Subversion*, *CVS*, *Git*, *MS Visual SourceSafe*, or *Mercurial*, are standard tools to help manage changes in documents during the development and maintenance of software systems. As changes to the system are made a new version is saved as a commit and stored by the version control system. This new version can be compared to previous versions (using tools such as *diff*) to determine what changed. These changes may be quite simple, such as fixing a spelling error in a comment, or quite complex, such as adding a new feature to the system.

Error correction (i.e., bug fixing) is most often typified by small and infrequent modifications to source code [Raghavan et al. 2004], [Kim, Whitehead Jr. 2008], [Hattori, Lanza 2008], [Hammad, Collard, Maletic 2009] as these types of changes rarely require major reworking of the design. Adding new features or altering the design of a system typically requires the addition and/or removal of classes or methods in an object oriented system. This latter class of changes often has broader implications to developers, testing plans, and project management. Here we focus on change (more specifically commits) that alters the design of a system. Furthermore, we would like to understand more about the types of different design changes taking place in a given commit and across the evolution of a system. This work proposes a means to categorize commits that impact the design of a software system.

Knowing what types of changes are occurring in a given commit would be very valuable to developers, testers, and managers. For example, if we know a commit changes the behavior of a given class, then that class would need to be re-tested and additional test cases may need to be developed or integrated into the testsuite. This would also give some notification to a developer that code using this class may be impacted. A manager could use such information to assess the cost of a given change and assess the risks of different deployment options. That is, if a particular change impacts a module or class that has historically been error prone, the risk assessment may be too great to deploy that change.

In an ideal environment, good development practice would annotate a commit with an accurate description about what is being changed. However, in reality this is often not done, or done inaccurately or incompletely.

Therefore, we feel that automated methods to augment the commit messages would be valuable. Additional knowledge can be derived from the source code and the commit, and explicitly documented to help address this problem. To accomplish this we must first develop a set of commit-categories (or types) that are meaningful to developers in assisting their understanding of what maintenance activities are taking place in a commit. Commits can be categorized with data present in the version control system or directly measured from the commit, e.g., LOC, author, etc. Commits can also be categorized based on analysis of messages via Natural Language Processing [Hattori, Lanza 2008] or information retrieval techniques [Kagdi, Poshyvanyk 2009]. Additionally, techniques have been used to categorize commits based on simple static analysis of the code

changes; our previous work on identifying design commits [Hammad, Collard, Maletic 2009] is one such approach.

However, these techniques do not provide deep insight into how these changes alter the actual code. Of course, the alternative approach is to conduct a full impact analysis of the change; however this is typically quite costly, time consuming, or impractical. Our goal is to develop an efficient approach that provides simple, yet fairly accurate, heuristics to the developers as to the overall characteristics of a given commit in the context of how it impacts the behavior or structure of classes. Other changes of interest here are those that impact the communication between classes, the access to a class, or the attributes of a class.

To accomplish this we build on our previous work that reverse engineers method stereotypes from source code. The stereotype information of methods added or deleted in a commit is used to construct a categorization of commit types. Then we define an automated approach to derive the commit type and label the commit with this meta-data. The final contribution is the evaluation of the approach on four open source systems that can serve for further studies and investigations.

5.2 Defining Commit Signatures

A commit details the changes to a software system and may represent major design changes as well as just minor edits or comment improvements. Here we provide a mechanism to automatically identify the different types of commits that impact the design of a system. Our approach of defining commit types is based on method stereotypes and how the changes impact different types of methods. Method stereotypes are

generalizations that reflect some intrinsic or atomic behavior of a method and indicate a method's role and responsibilities within a class. With stereotype information of the methods in a commit we can enrich the context of existing versioning systems with more semantics of method and class level changes.

Here, we will say a *method is in a commit* if the method is added or deleted as part of the commit. We now define the idea of a *commit signature*, which is used to identify a commit's type. The commit signature is the frequency distribution of stereotypes of methods occurring in a commit. We previously used a similar notion of signatures for the description of patterns of design at a system- and class-level. The commit signature provides information about what types of design changes are actually occurring in a commit. Here, a design change is defined as the addition or deletion of a class, a method, or a relationship (i.e., generalization, association, dependency) in the corresponding UML class diagram [Hammad, Collard, Maletic 2009].

5.2.1 Method Stereotypes

The aggregates for commit signature identification are method stereotypes (see Table 1). The stereotype of each method is reverse engineered using the taxonomy and the StereoCode tool. The stereotypes of the added and deleted methods in a commit form a descriptor of the change embodied by the commit. Let us recall that the taxonomy of method stereotypes is organized by the main role of a method, while simultaneously emphasizing its creational, structural, behavioral and collaborational aspects with respect to a class's design. Hence, the commit descriptor reflects creational, structural, behavioral and collaborational features of the design changes performed.

5.2.2 Commit Signature

The idea of a signature was exploited at a class- and system-level to characterize software (Chapter 3, Chapter 4). Here we apply a similar concept to commits to better understand design changes to a system.

A commit signature is the distribution of method stereotypes for the methods that are added or deleted in the change. They provide us with a heuristic of the structural complexity of the changes occurring in a commit. From the commit signature we can infer information of how the system was changed and whether the system gains more structural, behavioral, collaborational, or control features.

A signature is formed by determining which methods are in each commit (i.e., those methods that are added or deleted) and then reverse engineering the stereotype for each of these methods. The sum of the stereotypes in the commit is calculated. The method stereotypes counts can be shown as a bar chart ordered by method stereotype categories: accessors, mutators, creational, and collaborational. In the chart the method stereotype is given a grey shadow effect if collaborator is a secondary stereotype for the method.

An example commit signature from the open source system *Kate* is given in Figure 29. The commit #496124 consists of added/deleted accessors and mutators in almost equal proportions – five and four, respectively. Only the simplest accessor *get*, which just

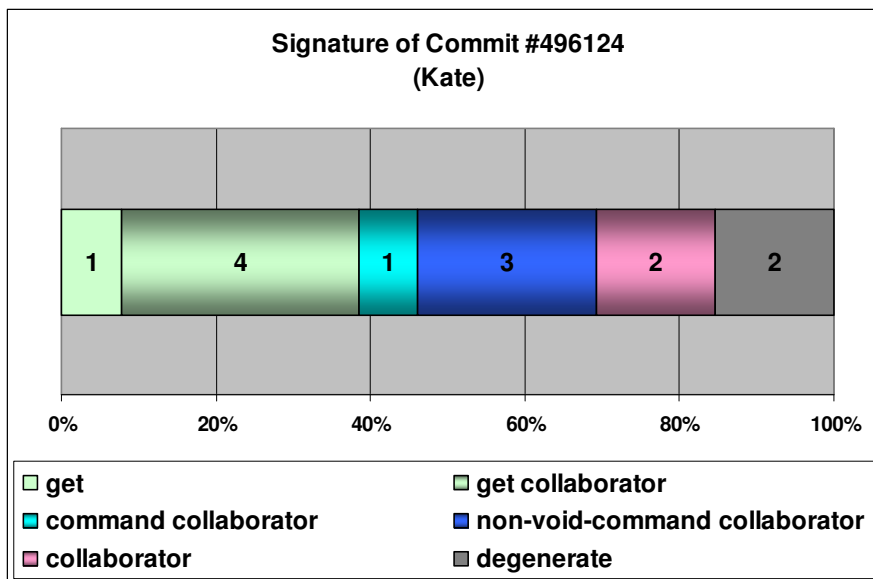


Figure 29. Commit signature, i.e., the distribution of method stereotypes, for commit #496124 from Kate with 13 added/deleted methods. The numbers in the rectangles show counts of methods stereotypes participating in the commit.

reads an object's state, participates in this design change. Methods changing object state are represented by command and non-void-command stereotypes. Degenerate, i.e., not yet implemented, methods also participate in the commit. The commit is highly collaborational, i.e., changes made included many objects. We can characterize these changes as a commit that added collaborational features along with structural and behavioral features (in equal proportions) to the existing system's functionality.

Another example signature for a larger commit is shown in Figure 30. This change includes 118 methods added/deleted and a large diversity of stereotypes is represented. Property, complex mutator, command, as well as a few control functions performed by controller methods add many of the behavioral features. Controller methods implement the class's external behavior, as they work only outside the class on objects of different type. Additionally, this commit is highly collaborational – approximately 60% of

methods added/deleted are coupled with other objects. Overall, we can characterize this commit as adding/deleting mainly behavioral and collaborational features along with a

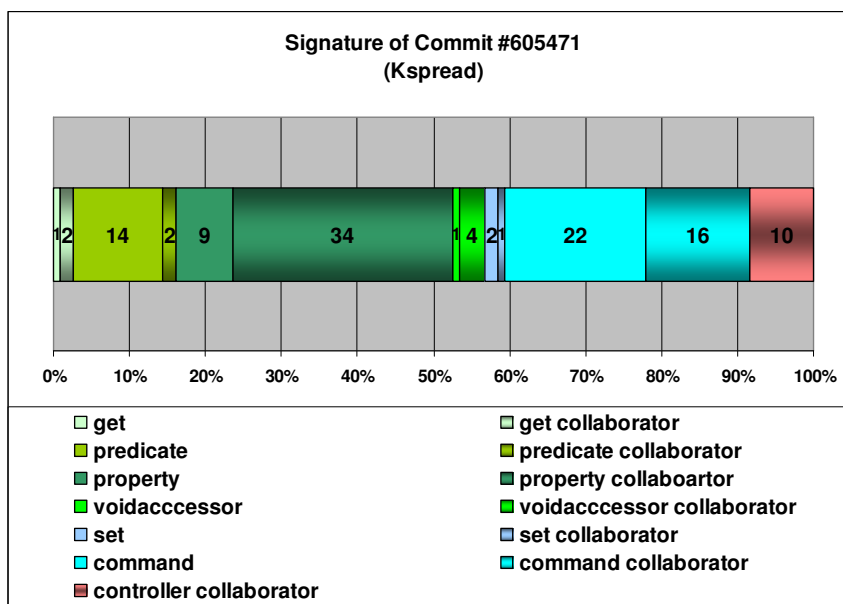


Figure 30. Commit signature for the large commit #669042 from KSpread with 118 added/deleted methods.

smaller number of structural and control features to the existing system functionality. The distribution of method stereotypes makeup the commit signature and forms a basis for determining commit types.

5.3 Commit Categorization

Commit categorization based on our empirical examination of the evolution history of a number of open source systems is presented in this section. The process of commit categorization is influenced by our previous work on uncovering patterns of design from a single-version system at different levels of abstraction: method [Dragan, Collard, Maletic 2006], class [Dragan, Collard, Maletic 2009], and system [Dragan, Collard,

Maletic 2010]. We will start with a description of how the commit categorization was created.

Initially, we created the taxonomy of method stereotypes and reverse engineered the method stereotypes from C++ software systems (e.g., HippoDraw and Qt) [Dragan, Collard, Maletic 2006]. We found a number of patterns of design at the method and class level. The validation of the method's taxonomy on additional systems gave us further evidence of the existence of these patterns of design abstractions.

We then classified software at the system level based on the method stereotype distributions using the system signature [Dragan, Collard, Maletic 2009]. Automatic hierarchical (COBWEB) and partitional (X-Means) clustering was used to classify 21 open-source C++ systems of various sizes, problem domains, and architectures (the full list of systems is given in [Dragan, Collard, Maletic 2010]). Clusters were characterized by the frequency and distribution of method stereotypes. The results showed that these distributions are an indicator of system architecture/design, and we observed additional patterns of the method stereotype distributions at the class level.

That led to a more thorough investigation of the patterns of design at the class level [Dragan, Collard, Maletic 2010]. We created a taxonomy of class stereotypes and developed an approach for automatic identification.











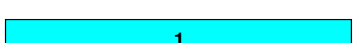
This foundation for identifying stereotypes at the method, class, and system level allowed us to hypothesize that those patterns of design, in the form of method stereotype distributions for a single-version system, also exist in multiple-version systems and could characterize design changes over the evolution history. A software system evolves

through the changes in structural, behavioral, creational, and collaborative characteristics which are implemented in methods. Each method in a commit has specific responsibilities within the class, and we characterize a commit by aggregating the responsibilities of the methods added/deleted in the change. The commit types are defined from the distributions of method stereotypes, i.e., the commit signature.

The list of commit types is shown in Table 11. A given commit may take on, i.e., have the characteristics of, more than one of these types. Examples of each commit type are presented visually in Table 11 along with an example commit and its signature from the system *Kate*. The color scheme used in Table 11 is the same one as the one found in Figure 29 and Figure 30. Clearly, categorization reflects the nature of the systems studied in [Dragan, Collard, Maletic 2006], [Dragan, Collard, Maletic 2009], [Dragan, Collard, Maletic 2010] and may not completely generalize to all domains. Also, there is some subjectivity to the categorization. However, the evidence gathered so far does support our hypothesis. We will now individually explain each commit type.

A *Structure Modifier* commit is responsible for changes related to data storage and only contains methods that perform simple access and modification to the data. It consists of only get and set methods. The example of the Structure Modifier, shown in Table 11, is commit #502478 with 3 get and 2 set methods.

Table 11. Commit types. Accessors are shown in green colors, mutators – in blue, factory – in tan, collaborational - in rose and turquoise, degenerate – in grey. The method stereotype has a grey shadow effect if ‘collaborator’ is a secondary stereotype for this method.

Commit Type Name	Signature
Structure Modifier (commit #502478)	
State Access Modifier (commit #582964)	
State Update Modifier (commit #593810)	
Behavior Modifier (commit #493147)	
Object Creation Modifier (commit #496123)	
Relationships Modifier (commit #496124)	
Control Modifier (commit #6375 QuantLib)	
Large Modifier (commit #605471 KSpread)	
Lazy Modifier (commit #859282)	
Degenerate Modifier (commit #715531)	
Small Modifier (commit #525142)	

A *State Access Modifier* commit consists of methods that provide a client with information and does not change any data members. It consists of almost all accessor methods. For example, commit #582964 has different types of accessors (get, property, property collaborator and void-accessor collaborator) for 8 out of 9 methods participating in the commit.

A *State Update Modifier* commit provides changes related to updates of an object's state. It consists mainly of mutator methods. These methods often implement complex behavior and may involve objects of different classes. Thus, a State Update Modifier commit involves both structural and behavioral changes. For example, commit #593810 has all 3 types of mutators (set, command and non-void-command), and 7 of 9 methods participating in the commit are mutators.

A *Behavior Modifier* commit is a special case of the State Update Modifier where the main characteristic is to execute complex *internal* behavioral changes within an object. It mainly consists of command and non-void-command methods. The largest part of the logic for the class's behavior is implemented in these methods. The commit #493147 is an example of the Behavior Modifier commit. About 86 % (6 out of 7) of its methods are command and non-void-command, and there are no set methods. Note that the difference between State Update Modifier and Behavior Modifier is in the percentage of set methods in a commit: State Update Modifier can have many set methods and a smaller percentage of command/non-void-command methods.

An *Object Creation Modifier* commit is responsible for changes related to the creation of objects and has mostly factory methods. The commit #496123 is an example of the Object Creator commit with 75% factory methods.

A *Relationships Modifier* commit adds or deletes methods that implement generalization, dependency and association relationships by performing calls on parameter or local variable objects. Therefore, changes to a system, performed by a commit consisting of many collaborational methods, represent modifications of relationships between classes. Alternatively, this type of commit could be a State Access Modifier when the main purpose of its methods is to get data from a model (when it has mainly accessors), or a Behavior Modifier when the main purpose of its methods is to update data (when it has mainly mutators). The commit #496124 is an example of a Relationships Modifier commit with more than 75% of methods that have a stereotype collaborator, e.g., get-collaborator, command-collaborator, etc. The commit #593810 is an example of State Update Modifier and Relationships Modifier.

A *Control Modifier* commit provides changes in the *external* behavior of the participating class, i.e., it processes data of the class's external objects. It consists mostly of controller methods (6 out of 7) that implement external class's behavior, because they work only outside the class, on objects different than *itself*. The commit #6375 (QuantLib) is an example of this stereotype. The only change is the addition/deletion of a controller method.

A *Large Modifier* commit contains a large number of responsibilities. This is a commit with a high impact on design. "Large number" can be characterized using

metrics such as number of methods, number of classes, LOC, etc. However, those types of metrics do not directly reflect the different semantics of changes. We consider a commit a Large Modifier commit if it has many methods and combines multiple roles, such as State Access Modifier, Behavior Modifier, Relationships Modifier and Control Modifier. The commit #605471 (KSpread) is an example of a Large Modifier which consists of approximately 56% accessors, 35% mutators, and 9% controller methods.

A *Lazy Modifier* commit is a very trivial commit that does “too little³”. The Lazy Modifier commit might occur in the context of a new or planned feature that is not yet completed. This is a commit with a minimal impact on design. Similarly, “too little” can be interpreted using different metrics. We consider a commit a Lazy Modifier if it has get/set methods and a low percentage of other methods. The commit is also considered Lazy Modifier if it has a large number of degenerate methods, e.g., besides the get method commit #859282 has 2/3 degenerate methods (collaborator empty and empty).

A *Degenerate Modifier* commit includes a degenerate, incidental, or empty method. If a commit contains even one degenerate method it means that adding a new feature is planned. As a maintainer we would like to know when exactly in the evolution history this will occur and how this method is changed (if at all). An example of this commit is commit #715531 that has 1 predicate-incidental and 1 empty method.

³ Fowler uses this term in the description of bad smells. We use it in a similar fashion here.

A *Small Modifier* commit has only one or two methods and does not change the system significantly. Commits #715531 and #525142 are examples of the Small Modifier type.

5.4 Reverse Engineering Commit Types

With the commit types defined based on the commit signature, we can automatically reverse engineer the commit type. To do so we perform the following steps:

1. Recover design changes from the code changes of commit by the *srcTracer* tool [Hammad, Collard, Maletic 2009].
2. Extract added/deleted methods per commit from the design changes.
3. Identify method stereotype distribution (commit signature) for the extracted methods.
4. Identify a commit type by applying rules on the commit signature.

We limit our consideration to only added/deleted methods and ignore other types of design changes such as added/deleted classes or relationships. While this is a current limitation of the approach we feel it is a good approximation of using all the information.

Changes to method stereotypes implicitly cover other design changes in many cases. For example, adding a new class will be reflected in adding a number of new methods and their corresponding stereotypes, and adding a new dependency will often be realized by the addition of a new collaborational method.

Additionally, we also ignore changes to existing methods. Again, this is a limitation of the approach but we feel little additional information will be added by its inclusion. Our main argument for this is because we are particularly interested in changes that impact the system's design. Small changes to the body of existing methods often reflect

error corrections (bug fixes) and are less likely to impact the design. For example, Raghavan et al. [Raghavan et al. 2004] showed that most bug fixes changed `if` statements. In our previous work [Hammad, Collard, Maletic 2009] we showed that most bug fixes contain a small number of changed lines and do not add or delete any methods. Clearly, additional investigation is necessary to fully understand the impact of such changes and to completely support our argument. We leave a comparison of using these other types of changes in the derivation of commit categories for future work.

A tool was developed to automatically identify the commit types presented in the previous section. The commit signature is used to assign types to a commit. The rules for identification of commit types are influenced by the rules on automatic identification of patterns of design at the class level for a single-version system [Dragan, Collard, Maletic 2010].

First, we briefly describe our approach on identifying design changes within a commit. Then we present the rules to reverse engineer commit types from (C++) source code and give details of a tool that automatically labels a commit with its type(s).

5.4.1 Design Changes during Evolution

In [Hammad, Collard, Maletic 2009], an approach and a tool `srcTracer` (Source Tracer) was developed to automatically identify code changes that break traceability links between code and design. A *design change* is defined as the addition or deletion of a class, a method, or a relationship (i.e., generalization, association, dependency) in the corresponding UML class diagram of the code. These types of changes impact the structure of the class diagram in a meaningful way with respect to the abstract design.

Any commit that causes the addition/deletion of a class or method, or addition/deletion in a class relationship, is considered a design impact commit.

The approach of identifying design changes begins by examining a single code change within a commit. First, the source code of the two revisions is translated to srcML [Collard, Maletic, Marcus 2002], an XML format that supports the static analysis required. Second, the code changes are represented with additional XML markup in srcDiff [Maletic, Collard 2004] that supports syntactical analysis on the differences. Lastly, the changes that impact the design are identified from the code changes via a number of XPath queries. The design changes are identified by querying the differences (in srcDiff), and added/deleted classes, methods and relationships (generalizations, associations, and dependencies) are reported (for more complete details see [Hammad, Collard, Maletic 2009]).

Here this approach is applied to the analysis of the code changes in commits over a period of system evolution to identify added/deleted methods from commits. As we mentioned previously, other types of design changes are ignored.

5.4.2 Rules to Identify Commit Types

The identification rules (Table 12) for commit types are based on the method stereotype distribution of the commit signature. Fractional thresholds, natural number thresholds with generally accepted meanings, average, and standard deviation are used in rule definitions. These thresholds are proposed in [Lanza, Marinescu 2006] as a means to

Table 12. Identification rules for commit categorization.

Type	Description	Rule
Structure Modifier	Only the simple accessor and mutator, get and set, are present	$ get + set \neq 0 \ \& \ methods - (get + set) = 0$
State Access Modifier	Consists mostly of accessors	$ accessors > \frac{2}{3} \cdot methods $
State Update Modifier	Consists mostly of mutators	$ mutators > \frac{2}{3} \cdot methods $
Behavior Modifier	Consists mostly of command and non-void-command methods	$ command + non-void-command > \frac{2}{3} \cdot methods $
Object Creation Modifier	Consists mostly of factory methods	$ factory > \frac{2}{3} \cdot methods $
Relationships Modifier	More collaborators than non-collaborators Not all the methods are factory methods Low number of controller methods	$ collaborators > non-collaborators $ $ factory < \frac{1}{2} \cdot methods $ $ controller < \frac{1}{3} \cdot methods $
Control Modifier	Many control features Controller is present	$ controller + factory > \frac{2}{3} \cdot methods $ $ controller \neq 0$
Large Modifier	Categories of stereotypes (accessor with mutator) and (factory with controller) have to participate in distributions not in small proportions Controller or factory have to be present Number of methods in a commit is high	$ accessors + mutators > \frac{1}{5} \cdot methods $ $ factory > \frac{1}{10} \cdot methods \vee controller > \frac{1}{10} \cdot methods $ $ accessors \leq \frac{1}{2} \cdot methods \vee mutators \leq \frac{1}{2} \cdot methods $ $ factory \neq 0 \vee controller \neq 0$ $ methods > average + stdev$
Lazy Modifier	Has to contain get/set methods It might have a large number of degenerate methods Occurrence of other stereotypes is low	$ get + set \neq 0$ $ methods - (get + set - degenerate) \leq \frac{1}{3} \cdot methods $ $ degenerate > \frac{1}{3} \cdot methods $
Degenerate Modifier	Has at least one degenerate method	$ degenerate > 1$
Small Modifier	Number of methods in a class is less than 3	$ methods < 3$

characterize and evaluate the design of object-oriented systems. First we introduce the notation used in the rules for commit type identification.

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

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

The set $\{degenerate\}$ consists of $accessors$ $degenerate$ ($predicate$ $incidental$, $void$ - $accessor$ $empty$, etc), $mutators$ $degenerate$ ($command$ $incidental$, non - $void$ $command$ $incidental$), and $collaborator$ $degenerate$ ($collaborator$ $incidental$, $collaborator$ $empty$). $|\{stereotype\}|$ is the cardinality of the set $\{stereotype\}$. Note, $average$ and $stdev$ of number of methods are calculated per system.

5.4.3 Implementation

Our tool, *StereoCode*, was extended to obtain commit signatures and reverse engineer the commit types. StereoCode reverse engineers method stereotypes using an infrastructure based on srcML (SouRce Code Markup Language) [Collard, Maletic,

Marcus 2002], an XML representation that supports both document and data views of source code.

After the generating commit signatures, they are fed into the tool *StereoCommit* that determines the type of each commit using the rules described previously. A commit is assigned the type if all conditions of a rule are met. A commit may satisfy more than one rule and the assigned type is the concatenation of all matches. The stereotype identification part of *StereoCommit* is implemented in C++. The tool currently works only for C++ source code as input.

5.5 The Case Study

The evolutionary histories of four C++ open source projects (Table 13) over specific time durations were analyzed. The main questions we address here are the following: Do the commit types identified by the tool exist in the evolution histories of real systems? Do most commits fit into at least one commit type? What are the most common types? What kinds of changes are prevalent in the evolution history?

The systems selected include the KDE editor *Kate*⁴, the KOffice spreadsheet *KSpread*⁵, the quantitative finance library *QuantLib*⁶, and the cross-platform GUI library

⁴ See kate-editor.org

⁵ See www.koffice.org/kspread

⁶ www.quantlib.org

*wxWidgets*⁷. The systems were chosen to represent a range of sizes, problem domains, and architectures. These projects are written in C++, well documented, have a large evolutionary history, and vary in their purposes.

Table 13. An overview of the software systems evaluated in the empirical studies. Ordered by the number of commits.

System	Time Period	Total Commits	Commits with Design Changes
Kate	3 years (1/1/2006–12/31/2008)	1592	403
KSpread	3 years (1/1/2006–12/31/2008)	2389	686
QuantLib	3 years (1/1/2006–12/31/2008)	2701	748
wxWidgets	3 years (1/1/2005–12/31/2007)	11438	1531

For each system we automatically determined the types of each commit using the StereoCommit tool. The resulting distribution of commit types for each system is given in Table 14.

The results show that all commits fit into at least one of the commit types and all commit types occur in all of these systems. The most common type is Relationships Modifier that occurs in between 63% to 87% of the commits in the four systems. The State Update Modifier and Behavior Modifier types occur in at least 46% in three of the

⁷ www.wxwidgets.org

systems, but less than 20% in one system - *QuantLib*. The State Access Modifier type occurs frequently (about 70%) for *QuantLib*, but the occurrence is low in the three other systems (from 14.4% to 24.9%). The Degenerate Modifier type occurs in about 10% of the commits for 3 systems and 7.3% for *KSpread*. Control Modifier and Object Creation Modifier types do not occur in significant numbers in any of the systems (maximum of 4.7% and 3% respectively). Lazy Modifier varies significantly (from 5.4% to 44.8%) and Large Modifier occurs from 4.2% to 10.1%. The Structure Modifier commit type has very low numbers (maximum of 2%).

Table 14. Distribution of commit types across 4 open-source systems.

Commit type	Kate		KSpread		QuantLib		wxWidgets		Min (%)	Max (%)	Avg (%)	Stdev (%)
	#	%	#	%	#	%	#	%				
Structure Modifier	4	1.0	1	0.1	5	0.7	4	0.3	0.1	1.0	0.5	0.4
State Access Modifier	58	14.4	171	24.9	520	69.5	229	15.0	14.4	69.5	30.9	26.2
State Update Modifier	263	65.3	354	51.6	145	19.4	986	64.4	19.4	65.3	50.2	21.4
Behavior Modifier	227	56.3	315	45.9	120	16.0	880	57.5	16.0	57.5	43.9	19.3
Object Creation Modifier	7	1.7	9	1.3	22	2.9	46	3.0	1.3	3.0	2.2	0.9
Relationships Modifier	254	63.0	465	67.8	647	86.5	1068	69.8	63.0	86.5	71.8	10.2
Control Modifier	19	4.7	22	3.2	14	1.9	52	3.4	1.9	4.7	3.3	1.2
Large Modifier	17	4.2	69	10.1	45	6.0	122	8.0	4.2	10.1	7.1	2.5
Lazy Modifier	60	14.9	37	5.4	335	44.8	207	13.5	5.4	44.8	19.6	17.3
Degenerate Modifier	39	9.7	50	7.3	72	9.6	152	9.9	7.3	9.9	9.1	1.2
Small Modifier	236	58.6	382	55.7	352	47.1	1004	65.6	47.1	65.6	56.7	7.7
<i>Coverage</i>	99.5%		96.5%		99.3%		98.2%					

Based on the distribution of the commit types we observed some similarities and differences between the systems. The two systems, *Kate* and *wxWidgets*, show a similar distribution for about half of the commit types. *KSpread* and *wxWidgets* are close in distribution of Structure Modifier (0.1% and 0.3%), Relationships Modifier (67.8% and

69.8%), and Control Modifier (3.2% and 3.4%). *QuantLib*'s pattern is opposite to the other systems – a high number for State Access Modifier (69.5%) and a low number for the State Update Modifier and Behavior Modifier types (19.4% and 16%). This system has a very high percentage of relationships updates and a very low percentage of changes of control features. However, *QuantLib* and *Kate* are close in percentages for Degenerate Modifier commit type.

The chi-square test of independence was performed between the commit types and the four software systems. The null hypothesis is that the distribution of the commit types in different systems is a random phenomenon; the alternative hypothesis is that there is a link between commit types and software systems. Chi-square reports a p-value <0.0001 with 95% confidence and 30 degrees of freedom that lets us reject the null hypothesis. The critical and observed values are 43.7732 and 1389.114 respectively.

The results also show that the frequency and distribution of the commit types across a system reflects an implementation of particular design decisions, the underlying architecture, and good/bad design changes. For example the finance library has the highest percentage of Relationships Modifier and State Access Modifier, which is not surprising given the domain — these types of commits will be typically impacting calculations. Another example is that commit type Degenerate Modifier could represent a bad practice. Degenerate methods participating in this commit type should be tracked through the evolution history. If they are left unchanged during the evolution then we can assume that they are candidates for refactoring. Behavior Modifier, State Access Modifier, and Large Modifier are candidates for added/deleted (or edited) features or

concepts while that commit type of Structure Modifier does not include adding or deleting new features. However, a full investigation whether commit types reveal particular design/architecture, good/bad design changes and added/deleted (or edited) features and concepts is needed.

5.6 Applying the Approach – Commit Labeling

We now apply our approach of commit categorization by introducing a *commit label*. This is a direct and practical way to enrich the current versioning systems with additional information about code changes (i.e. commits). Previously, a commit labeling concept was described [Hammad, Collard, Maletic 2010], but it was limited to listing the exact design changes of commits. Here we use the commit label to brand a commit with its type. An example of a commit labeling is given in Figure 31. First, the commit type characterizes the commit at a high level. Then the commit signature presents a detailed view of changes performed in the overall commit. Finally, class-change signatures show detailed views of changes at the class level for all classes that participated in the commit.

The commit label in Figure 31 is for commit #496124 of Kate. Overall, this commit is labeled with the commit types Relationships Modifier and Degenerate Modifier. Looking further at the participating classes, we see that the commit involved added/deleted methods of four different classes. In particular, the class change signatures show that the class `KateScriptConfigPage` is responsible for the commit type Degenerate Modifier, and the other three classes are responsible for the commit type Relationships Modifier.

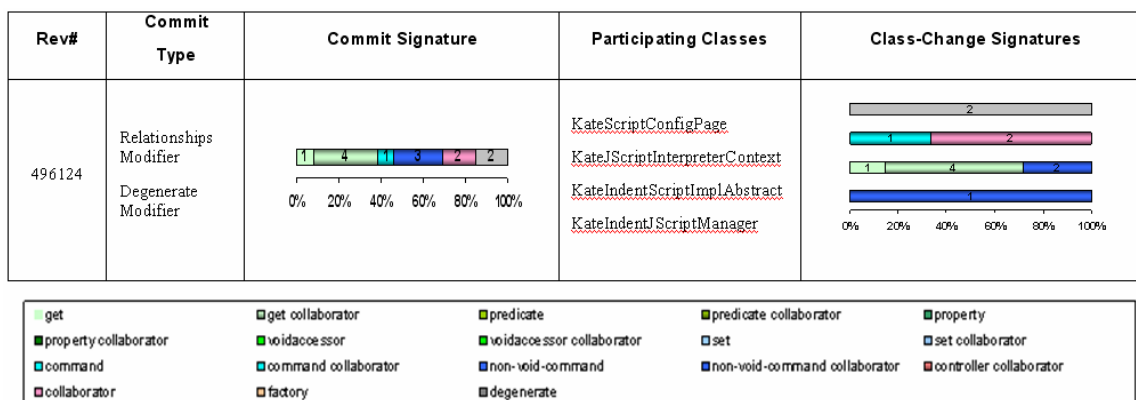


Figure 31. The Commit Label for commit #496124 of Kate presents both a high-level and detailed view of the commit. The Commit Type and Commit Signature are shown for the entire commit. The Participating Classes are the individual classes that contributed to the Commit Signature. For each of these Participating Classes, the individual Class-Change Signatures are given.

Commit labels could provide support for understanding the direction in which a software system as well as individual classes have been developing, and to assist in the identification of the most important commits/changes with respect to design changes. By analyzing the commit label at a particular point of time in the history of a project, one could learn what and to which degree it has been modified. That is, was the class's structure, behavior, collaborations or control functionality modified? For example, by using commit labels, the monthly/weekly distribution of commit types can be analyzed for the following purposes:

1. Determining the direction of the evolution of the system during a specific time period. E.g., months with the commit types of Structure Modifier and State Access Modifier indicate that the focus of changes to the system is to modify the access (reading) of object state.

2. Assistance in planning maintenance activities using the commit type to determine which classes need further design work and refactoring.
3. Determining how features and bug fixes were implemented, what commit types correspond to these activities, and finding if there was any specific ordering of commit types for these activities.

5.7 Threats to Validity

Commit type identification and the *StereoCommit* tool are subject to a number of threats to validity. The rules for stereotype identification are subjective in part, and thresholds might vary depending on differences in interpretation. The results of automatic identification of design changes are based on applying *srcTracer* which has its own threats to validity [Hammad, Collard, Maletic 2009].

The case study is limited to only four open source projects with one continuous time period for each project. Results may vary for other projects or variable and different time durations. However, the extracted data and time periods are not small or seriously limited.

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

5.8 Related work

Recently, more attention has been focused on the discovery of evolutionary change from historical data. Evolutionary information of complex software systems at different levels is a focus in software engineering research, and is a solution for the problems of increasing complexity and decreasing software quality [Gall, Lanza 2006].

Automatic classification of large changes in software systems into various categories of maintenance tasks - corrective, adaptive, perfective, feature addition, and non-functional improvement - using machine learning techniques is given in [Hindle et al. 2009]. Hattori et al. [Hattori, Lanza 2008] propose commit classification with respect to the size that is based on the number of files. Additionally, they classify commits by the types of development (forward engineering) and maintenance (reengineering, corrective engineering, and management) activities based on the content of the comments.

Evolution of the object-oriented software system at a coarse-grained level (such as package-level) is analyzed in [Dong, Godfrey 2008]. In their exploratory study, seven releases of Apache Ant, a Java-based build tool, are analyzed using hybrid models proposed by the approach. Design patterns on the class-level are investigated in [Kim, Pan, Whitehead 2006] to develop results which are common across projects or releases and help maintainers capture and better comprehend architectural evolution. Three open-source projects written in Java are analyzed to identify common kinds and frequencies of micro patterns as well as bug-prone patterns. Analysis of changes on a method-level, i.e., function signature changes through evolution such as their kind, frequency, correlation

with changes in LOC and number of function body modifications, and evolution patterns, is performed in [Kim, Whitehead, Bevan 2006].

Evolution patterns of change types such as “new features added”, “bugs fixed”, and “consistency of coding rules re-established” are presented in [Fluri, Giger, Gall 2008]. Source code changes of one commercial and two open-source software systems are extracted, and then the agglomerative hierarchical clustering is applied. The results show that the control flow, the exception flow, or the API is affected by the changes.

Our work is distinguished by identifying key characteristics of commits, such as changes to the class structure, class behavior, changes related to the communication, creation and control of other objects, and type of access to class’s data members (read/write) used in the commit altered methods.

Gîrba et al. [Gîrba et al. 2007] proposed the usage of formal concept analysis to identify groups of entities with similar properties that change in the same way and at the same time. Using predefined conditions, specific change patterns are detected. For classes, the goal is to identify bad smells. Robillard [Robillard 2005] proposed a technique based on structural dependencies to automatically discover the program elements (including classes) relevant to a change task. The method depends on static analysis of the source code and does not take into account the history of changes.

Aversano et al. present an empirical study on evolution of design patterns in three open source systems and analyze which patterns tend to change more frequently [Aversano et al. 2007]. Vaucher et al. [Vaucher, Sahraoui, Vaucher 2008] proposed techniques to discover patterns of evolution in large object-oriented systems. To locate

patterns, they use clustering to group together classes that change in the same manner at the same time. We do not study the internal or external evolutionary patterns of classes; instead we match the semantic information about a group of added/deleted methods to a set of classes.

Static design patterns at the class-, package- and system-level have been investigated by many researchers. Class-level design patterns are presented in [Gil, Maman 2005] and [Lanza, Ducasse 2001b]. A number of approaches identify key classes in a software system [Zaidman, Demeyer 2008], [Greevy, Ducasse 2005]. We look at the dynamics of a system by analyzing method stereotypes. Our approach provides a characterization of patterns through the evolution history and gives a description of semantic changes at the method- and class-level performed in a commit. The main differences of our work is that we identify key features of commits by looking at how commits are impacted by design changes, and we categorize commits based on the stereotypes of methods participating in high-level design changes.

5.9 Conclusions

We present a categorization of commits that was derived from an empirical investigation of open source systems written in C++ and based on method stereotypes distribution. The implemented tool automatically reverse engineers a commit type and labels the commit with this information. The case study conducted shows that the commit types identified by the tool exist in evolution histories of real systems, and in the studied systems all commits fit into at least one commit type. In addition, we showed

how the resulting information can be used to create a commit label which combines an overview along with detailed information about the commit.

We are investigating the correlation between the commit type and maintenance type (e.g., bug fix, feature addition, and refactoring). Our initial results on one open source system show that this correlation exists. However, additional analysis is required for further conclusions.

For future work we also plan to infer from the commit label information about specific time durations/points in the history, and further investigate the questions whether commit types and labels reveal good/bad design changes and added/deleted (or edited) features or concepts.

The next chapter presents another application of method stereotypes to system evolution analysis. We now analyze the evolutionary patterns of design with respect to milestones, such as system releases. The changes in method stereotypes distributions through the releases, the distributions stability, and their correlation with respect to the different release types - 'bug fixes', 'refactorings', and 'adding new features' - have been investigated.

CHAPTER 6

EVOLUTION OF METHOD STEREOTYPES

In this chapter, we present our ongoing research on method stereotypes evolution: a case study for two open-source C++ projects investigated over more than twenty different releases. Why do we need to know that the method stereotype evolves? During the evolution of a software project, the original design changes, and those changes are embodied by the system's atomic blocks - methods. Therefore, we would like to be able to trace those changes, having historical data in the form of source code that is available from subversion repositories. Finding evolution patterns of design at the method level allows us to generalize this knowledge across projects and predict common situations in the future.

The next section describes the importance of analyzing evolution changes in system releases. Section 5.26.2 describes a case study for two open source projects, HippoDraw and QuantLib, of 20 and 23 releases respectively. In Section 5.36.3 we present the results on method stereotypes evolution and their correlation to release types. Section 5.46.4 describes patterns discovered, which is followed by a discussion of the results (Section 6.5).5.5

6.1 Motivation

The common metrics to measure system-size changes during system evolution are number of lines of code, number of commits, number of files, and number of classes

[Dong, Godfrey 2008] [Purushothaman, Perry 2005], [Alali, Kagdi, Maletic 2008], [Hattori, Lanza 2008] [Hammad, Collard, Maletic 2010], however these are poor predictors of specific design changes. On the other hand, the fact that the number of the method stereotype, for example ‘*predicate collaborator*’ or ‘*predicate incidental*’, is increased twice at release *x.i.ii* is almost certainly an indicator of an ‘adding new features’ design change. We analyze whether and how the changes in method-stereotype distribution across multiple releases help us understand the evolution of the system.

Software evolution can be analyzed at different levels of abstraction: design pattern evolution [Dong, Zhao, Sun 2010], tracing design changes at the package-, class-, and method-level [Dong, Godfrey 2008], [Kim, Pan, Whitehead 2006],[Kim, Whitehead, Bevan 2006]. We consider system evolution as evolution of the method stereotypes which is the fine-grained level closest to source code, and we have the infrastructure and the tool to automatically extract this information. Main questions we are interested in are: at what points in the history of a project is there a significant change in the stereotypes of methods, and what triggers these changes?

6.2 Case Study

The two open-source medium-sized software system *HippoDraw* and *QuantLib* are used in the case study. Our tool, *StereoCode*, was applied to extract method stereotype information. *HippoDraw* is an open-source C++ 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. *QuantLib* is a finance library for modeling, trading, and risk management. The source code for both is well written,

follows a consistent object-oriented style, and the version history and detailed documentation is available. The overview of the projects analyzed is given in Table 15.

Table 15. The overview of the projects, HippoDraw and QuantLib, in the case study.

System	Number of releases	Date		Number of methods		Number of files	
		first release	last release	first release	last release	first release	last release
HippoDraw	20	03/22/2004	10/01/2007	2585	3411	484	692
QuantLib	23	11/21/2000	01/11/2008	401	5262	113	1695

6.3 Results and Observations

In this section we present our observations from the exploratory case study where we tried to answer the following questions:

1. How does the distribution of method stereotypes evolve over time? Is the trend for stereotype evolution the same as for the system size?
2. Does the initial stereotypes distribution hold through the project releases?
3. How different are the distributions of method stereotypes for different release types: ‘bug fixes’, ‘changes’, and ‘adding new features’ releases?
4. Which stereotypes change more in the distribution? What are release-to-release increases and decreases?
5. What relationships exist between stereotype categories: collaborators vs. non-collaborators, accessors vs. mutators vs. external collaborators, degenerate vs. regular?

6.3.1 Evolution of Stereotype

Research questions. How does the size of the system evolve? Does the number of methods of a particular stereotype change with the same trend as the system size? Which stereotypes are different and how: is the trend decreasing, flat or erratic?

The size evolution in terms of the number of methods is shown in Figure 32. HippoDraw system size increases roughly logarithmically, while QuantLib size evolution is closer to exponential. The possible explanation for this fact is that HippoDraw is a one-developer application while QuantLib is an open-source library with multiple developers. The system size of HippoDraw stabilizes at some point and the size curve is almost flat for the last eight releases: starting from the release 1.20.6 the system size is 3397 methods, then it is increased insignificantly (up to 3411 methods) in the next-to-last release. QuantLib's size is constantly increased, and the most drastic changes are observed in the last 3 releases.

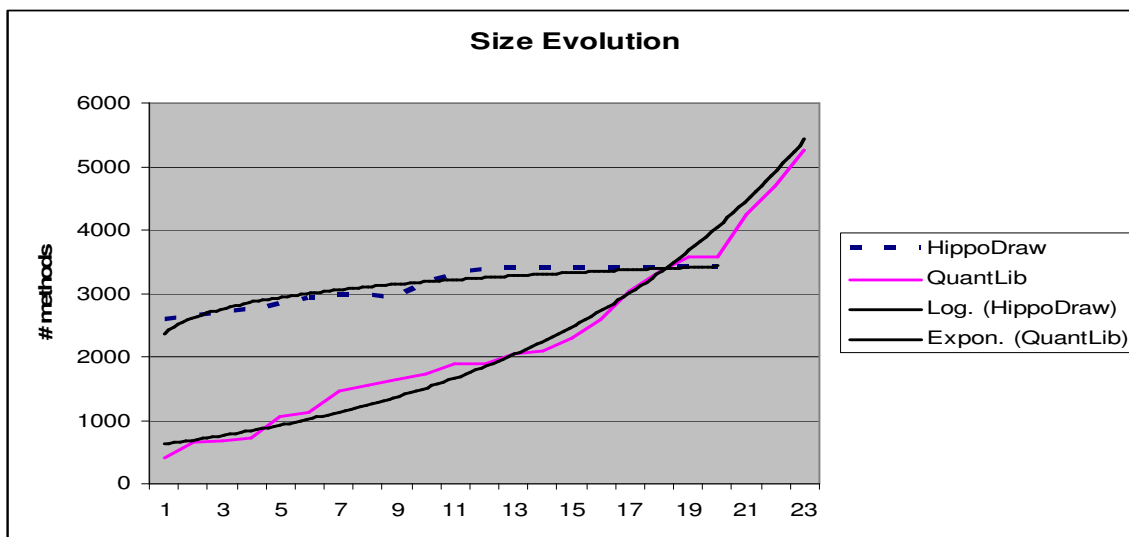


Figure 32. The evolution of the projects size in terms of the number of methods through the releases analyzed.

Now consider how the method stereotypes evolve. Note that in this case study (comparing with the case study in previous chapters) we additionally considered another type of method, *Degenerate::stateless*: it has no data members read/written directly, and it can have only one call to other class methods. The results of analyzing changes in the method stereotypes distribution follow.

HippoDraw.

There exists *stability of the method-stereotypes distribution after some point in time* starting at the release 1.20.0 along with the system stabilization.

A number of stereotypes follow the *system size trend* and have the *overall increase* of the stereotype percentage including stereotype predicate, predicate collaborator (Figure 33), property collaborator, controller, collaborator empty, and factory stateless (*clone*). *Clone* is a simple factory method which has the only purpose to create an external object and return it to the client. Clone does not read or write the object's state.

Possible explanation for the trend of these stereotypes is that when the system primarily evolves by adding new functionality but not just fixing bugs, then establishing new relationships is crucial. As a result, the collaborative aspects of the system increase, and methods which are responsible for behavior but not for supporting the structure of the system start playing a more important role. HippoDraw is a data analysis system, and accessing data seems to be more important than updating data; this results in increasing percentage of accessors in the stereotypes distribution.

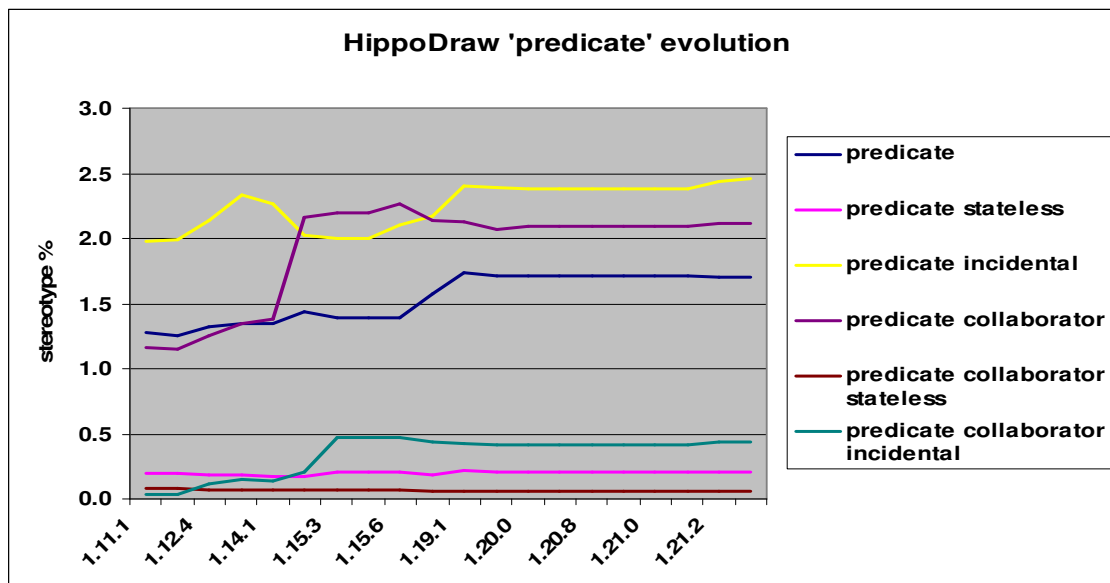


Figure 33. Evolution of the *predicate* stereotype in the HippoDraw system.

A number of stereotypes have the *overall decrease* of the stereotype percentage including get collaborator, property, and void-accessor.

A number of stereotypes stayed relatively *flat*, i.e. they grow almost at the same rate as the system evolves, including get/set, set collaborator, command, non-void-command, command collaborator, factory, and degenerate (stateless, incidental). *Stateless* are simple methods which have just one call. They are completely flat. *Incidental* methods are mostly flat with the exception of the predicate category. We have mostly the non-collaborational categories in this group. All the flat or relatively flat categories do not play an important role in system evolution and are mostly responsible for supporting system structure or simple system behavior.

The following stereotypes are *erratic* with some ‘spikes’ during the evolution, while overall the stereotype percentage is slightly increased/decreased in the system distribution: void-accessor collaborator is very spiky before the stable releases but overall

has a small increase; non-void-command collaborator has a small overall decrease but with some spikes before stabilization. In general, stereotypes in this group are a small part of the system and present non-prevalent categories: *void-accessors (collaborators)* are not very typical and present *degenerate accessors* when a class data-member accessed is returned through a parameter or is not returned at all; *non-void-commands* are approximately 1/5 of the large *command* category. More analysis is needed to find whether specific design changes exist which trigger those spikes.

QuantLib.

We follow the same classification as in the case for HippoDraw and have four categories of stereotypes distribution: *overall increase, overall decrease, flat, erratic*. However most of the stereotypes are ‘spiky’ while comparing them with the HippoDraw evolution.

The following stereotypes have an *overall increase* of the stereotype percentage: get/set collaborator, predicate, predicate collaborator, property collaborator (Figure 34), void-accessor collaborator, command collaborator, and factory.

The following stereotypes have an *overall decrease*: get, property (Figure 34), command, non-void-command, non-void-command collaborator, and incidental.

Here we have the same trend as for HippoDraw. In the ‘*overall increase*’ group all the stereotypes but one belong to the collaborational group and they are mostly accessors. Mutators do not increase: for QuantLib they are mostly in the *overall decrease* group, but in the *flat* group for HippoDraw.

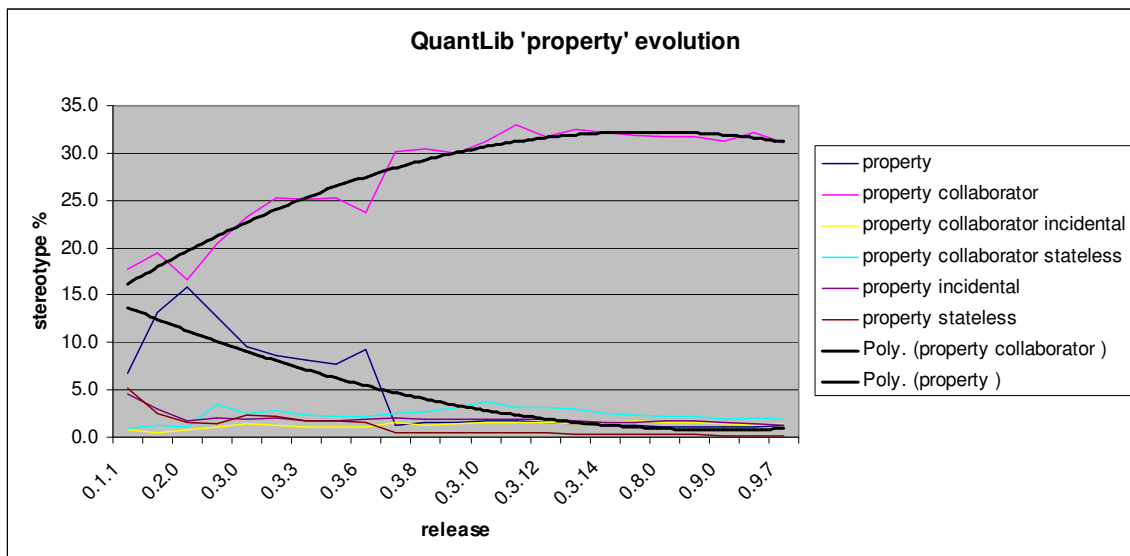


Figure 34. Evolution of of the *property* stereotype in the QuantLib system.

The stereotypes stateless are the only stereotypes in the *flat* category. The majority of stereotypes for the QuantLib system are not flat.

The following stereotypes are *erratic*: void-accessor, set, factory, property collaborator stateless, command stateless, and non-void-command stateless.

Except for the degenerated categories, only void-accesor and a special kind of factory are very unstable. Void-accesor has a low importance in the system (stereotype percentage is about 2).

6.3.2 Stereotype Distribution Stability

Research question. Does the initial stereotypes distribution hold through the project releases?

Figure 35 shows the distribution of the method stereotypes for the first and last releases of HippoDraw. As seen, the system is *stable*, and despite the small changes,

there is no considerable redistribution. The top method stereotypes - command, property, factory, and controller - are the same during the system's evolution.

QuantLib is unstable. There is a big difference between the first and last release's distribution. The changes in the stereotype percentage are significant, and redistribution is massive (see Figure 36). Order of importance is almost the same through the evolution: property, get, and command. However, these stereotypes decrease in favor of their collaborators; controller exchanged its place with the non-void command. An explanation of this fact is that QuantLib is an open-source finance library for modeling, trading, and risk management. The library is exploited across different research institutions, banks, and software companies, and should reflect constant and quick changes in economical life.

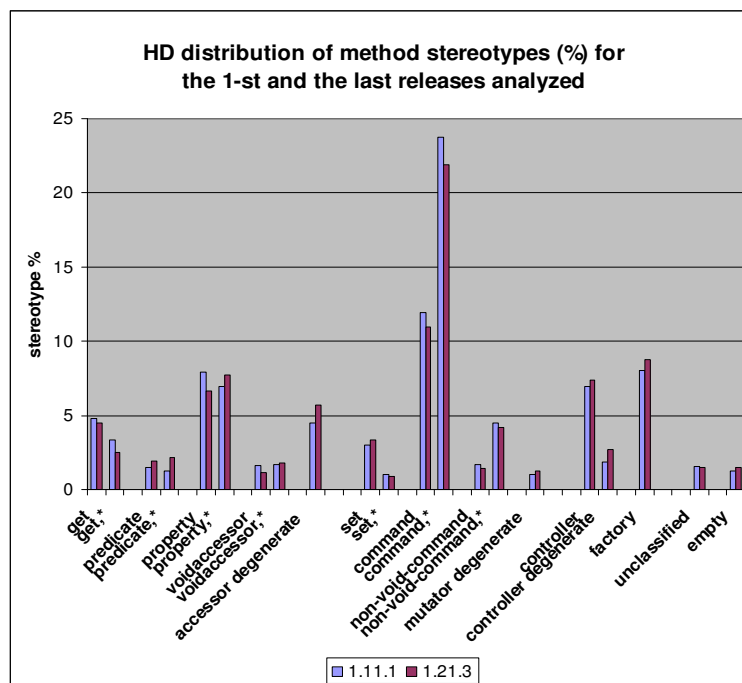


Figure 35. Method stereotypes distribution for the first-and-last releases of HippoDraw (* stands for the stereotype *collaborator*).

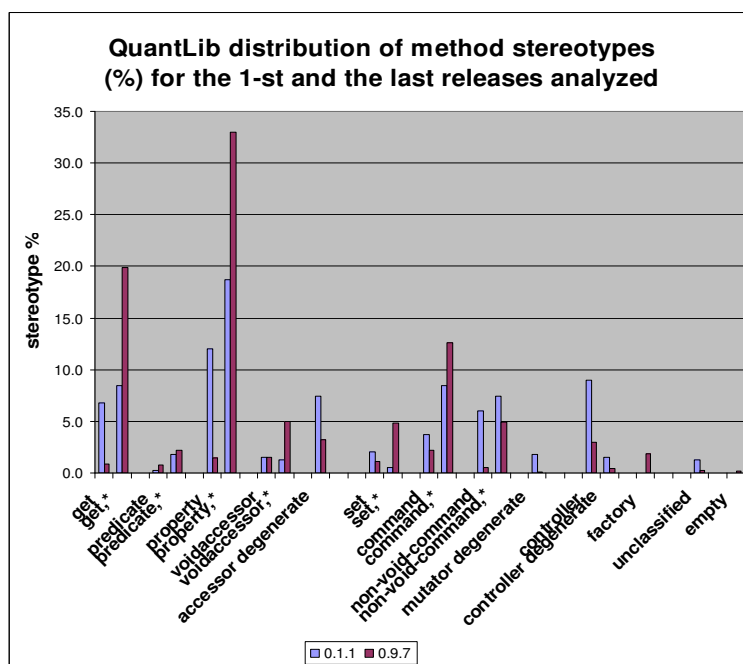


Figure 36. Method stereotypes distribution for the first-and-last releases of QuantLib (* stands for the stereotype *collaborator*).

6.3.3 Stereotype Distribution vs. Release Types

Research question. How does the method stereotype percentage change over the different release types: ‘bug fixes’, ‘changes/refactoring’ and ‘adding new features’.

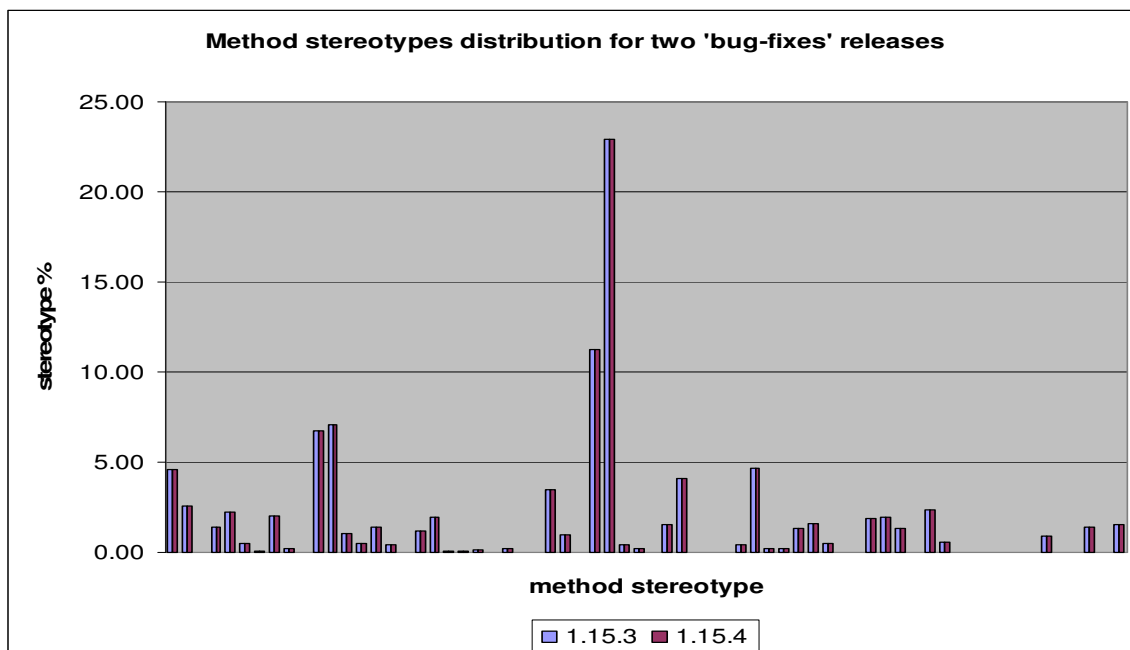


Figure 37. Method stereotypes distribution in two *bug fixes* releases of HippoDraw.

Here we analyzed only data for the HippoDraw system, because the information about release types was available on the project website provided by the original developer (unfortunately, we did not have that classification for the QuantLib system). The experiments showed that the method-stereotypes distribution is *stable in 'bug fixes'* releases (Figure 37), and there are *changes* in the method-stereotypes distribution in *'refactoring/changes'* and *'adding new features'* releases.

6.4 Patterns discovered

6.4.1 HippoDraw

The distribution of *get/set* has some contrast to other accessors/mutators. *Collaborators* have a higher percentage than their primary category. However, *get/set-collaborator* has a lower percentage than *get/set* and is almost flat with a small decrease in the beginning. *Get* and *set* are almost twice and three times as big as *get collaborator* and *set collaborator*, respectively. The explanation for this fact is that collaborations for *get* and *set* are less important, because native types represent a simple concept (e.g., size, counter, etc.), and simple *get/set* is a natural thing to use.

Collaborators for *predicate* and *property* are lower in the beginning. Overall, the raw number and the percentage of *predicates* both increase. It makes sense and means that in the beginning of implementation, the source code had to undergo more refactoring. This is a motivating practice and could be an indicator of good design. *Predicate collaborator* underwent a massive increase in the 1.14.1 release before the *predicate* did (Figure 33). The same holds for *predicate incidental* in the 1.12.2 release. Both massive increases can be interpreted as a preparation for implementation of new features in further releases.

Raw numbers for *predicate collaborator* continue to grow but the percentage goes down after the 1.5.6 release, which means that the system grows faster than *predicate collaborator* does.

Property has the same pattern as *predicate* does: initially the percentage of *collaborator* is lower than that of its primary stereotype. It is easier initially to write *predicates and properties* based on native types, and through the system evolution it gets

more complex. *Predicate/property* based on user-defined types becomes more important when the system evolves (1.13.1 and 1.15.3 releases are points of changes).

There is a very low percent of command-stateless in contrast to predicate/property-stateless. The main purpose of command methods is executing complex updates to an object state, and therefore methods with one call in the command category are not very common in the system.

6.4.2 QuantLib

Collaborators constantly play a more significant role over the project evolution. There is significant redistribution in favor of collaborators with respect to non-collaborational method stereotypes. *Collaborators* have a higher percentage than their primary category for all the stereotypes during the whole period of evolution, because many mathematical and financial objects are involved. However, in the last release we observe a drastic increase of collaborators with respect to non-collaborational stereotypes - approximately 10 times (see Figure 38 and Figure 39). The explanation for this fact is that a large amount of more complex objects started playing a more important role in financial calculations and engines.

Accessors have a larger distribution than mutators and play an even more important role towards the last release. The main objective of the library is to provide financial information for further usage in developers programs or applications, therefore all the accessors (get, predicate, property, void-accessor) and collaborators display a constant increase through the evolution history; updating information is not very

essential. Despite the overall raise of command collaborator, the proportion of all the accessors increases (approximately from 58% to 68%) compared to all the mutators.

Role of controller methods decreases. Number of controllers is decreased almost 3 times. Most likely classes were refactored to perform their tasks with their own data member objects, not delegating it to external objects. However, more detailed investigations of documentation and release notes are required to define specific reasons. Also, a number of factory methods increased, which could partially substitute roles of controller methods.

Degenerate methods decreased significantly. The percentage of degenerate methods went down from 1.7% to 0.3%, which indicates a good practice – initially non-functional methods were refined and implemented.

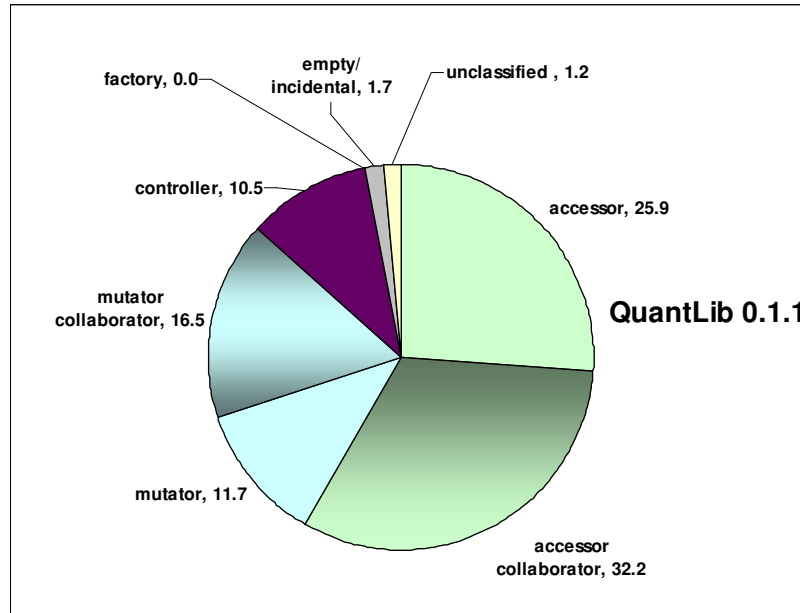


Figure 38. Stereotype Category distribution for the first releases of QuantLib (0.1.1).

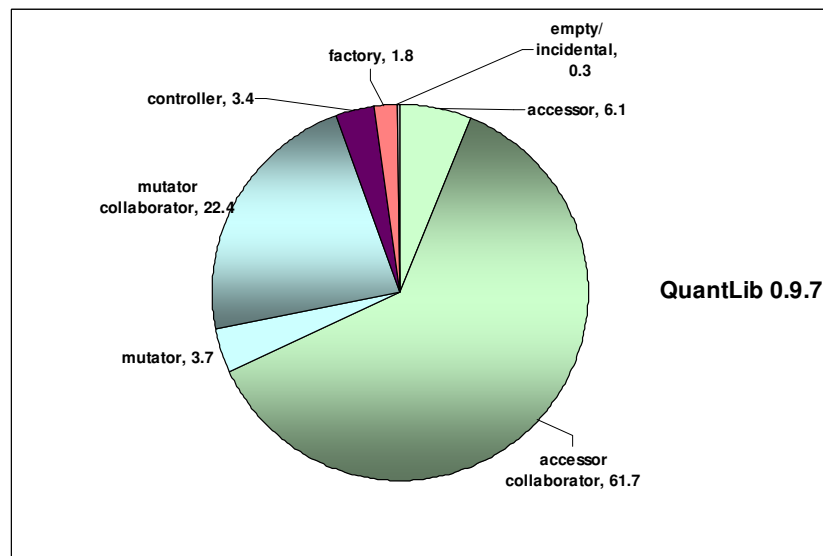


Figure 39. Stereotype Category distribution for the last releases of QuantLib (0.9.7).

6.5 Discussion

The results show that the method stereotypes distribution across releases is indicative of types of design changes and correlates with system size evolution for several

stereotypes. For the system with detailed information about the release types (HippoDraw), the method-stereotypes distribution was stable in ‘bug fix’ releases (8 releases) and changed in ‘refactoring/changes’ or ‘adding new features’ releases. For primary categories, in most of the cases, the frequency of collaborator methods was higher than that of non-collaborators. In general, accessors increased, while mutators remained stable or decreased.

We plan to extend this work by studying the historical data of a broader range of systems to identify evolutionary patterns related to method and class stereotypes. This will also examine whether architecture style, domain, programming style and number of developers reflect common evolution trends of a software system.

Additionally, we plan to more thoroughly investigate change patterns on the method- and class-level through the evolution of a project. Detailed analysis of release notes and documentation is needed to correlate specific development and maintenance activities to particular method and class stereotypes and their sequences in the design changes.

CHAPTER 7

CONCLUSIONS

The dissertation addresses several research issues related to reverse engineering, design recovery, and program understanding. The first issue deals with characterization of software at different levels of abstraction: method, class and system. The basis for automatic identification of patterns of design at all levels is the method stereotypes and the two perspectives of method stereotypes distributions. The second issue deals with characterization of changes in software during its evolution history. Automatically classifying commits and uncovering evolution patterns of method stereotypes help developers gain a high-level perspective of design over a system's evolution.

7.1 Contributions

The main contributions of the presented research are discovering emergent laws of stereotypes in object-oriented source code and their relationships with design, as well as developing a new approach to automatically classify software with respect to method stereotypes.

The first specific research contribution is the taxonomies of object-oriented method stereotypes [Dragan, Collard, Maletic 2006] and class stereotypes [Dragan, Collard, Maletic 2010] that reveal roles and responsibilities of methods and classes with respect to structural, behavioral, collaborative, and control features. This is the first broad investigation on the topic of method stereotypes with respect to reverse engineering and design recovery. Reverse engineering method stereotypes allowed recovering a class

design model with respect to the boundary, control, and entity class's stereotypes from existing object-oriented code.

The second contribution is designing approaches and tools to characterize and reverse engineer patterns of design at the method- [Dragan, Collard, Maletic 2006], class- [Dragan, Collard, Maletic 2010] and system-levels [Dragan, Collard, Maletic 2009]. Based on the method stereotype taxonomy the notion of a *signature* and its two perspectives, *Stereotype* and *Stereotype Category*, was introduced and serves as a foundation for the automatic identification of class stereotypes, classifying software at the system level and categorizing commits. The tools *StereoCode* and *StereoClass* were implemented to automatically identify the methods and class stereotypes and redocument source code with the stereotypes. Hierarchical and partitional clustering algorithms were used to classify software at the system level. The results show that the frequency and distribution of the method stereotypes is a good indicator of system architecture/design.

The third contribution is an application of the method stereotypes to historical data and analysis of evolutionary patterns of commits stored in a version control system, and system releases. The *commit signature*, based on the method stereotypes distribution in a commit, was used to categorize commits that impact the design changes to a class or classes over the system evolution. The *StereoCommit* tool was implemented to automatically identify the commit types. An application of the commit categorization is a *commit label*. The *commit label* combines an overview along with detailed information about the commit and is useful for analyzing the system evolution in maintenance activities.

The final contribution is the evaluation of the approaches and the tools by performing empirical studies. A wide range of about 30 open source object-oriented software systems, of a different size and different domains, was investigated and the results can serve as a benchmark for further investigations and studies.

7.2 Future work

This work forms the basis for a number of avenues of research in design recovery, and we plan to extend our work in a few directions. The construction of design-quality metrics based on stereotype classification is an interesting and useful application. 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 stereotypes.

The automatic identification of class stereotypes supports better program comprehension and forms a foundation for a number of applications based on class stereotypes. We plan to integrate annotation features into editing in Eclipse plug-in for srcDoc [Shearer, Collard 2007]; incorporating method and class stereotype information in UML class diagrams to improve program comprehension, and for automated layout of class diagrams with respect to architectural importance. The proposed class stereotypes could be used to not only characterize design and implementation solutions but also to evaluate and improve design or use as an indicator of bad design in need of refactoring. A few class stereotypes are candidates for refactoring in particular situations, and we will investigate this as future work.

We plan to expand the work at the system level by studying the change of signatures over the evolution of a system. This will examine whether the signature of a system changes over the development of a project, and at what point the signature becomes stable.

The correlation between the commit type and maintenance type (e.g., bug fix, feature addition, and refactoring) is also being investigated. Our initial results on one open source system show that this correlation exists. However, additional analysis is required for further conclusions. For future work we also plan to infer from the commit label information about specific time durations/points in the history, and further investigate the questions whether commit types and labels reveal good/bad design changes and added/deleted (or edited) features or concepts.

While our tools are specifically for C++ the approach is easily extended to other object-oriented programming languages (ex., Java, C#).

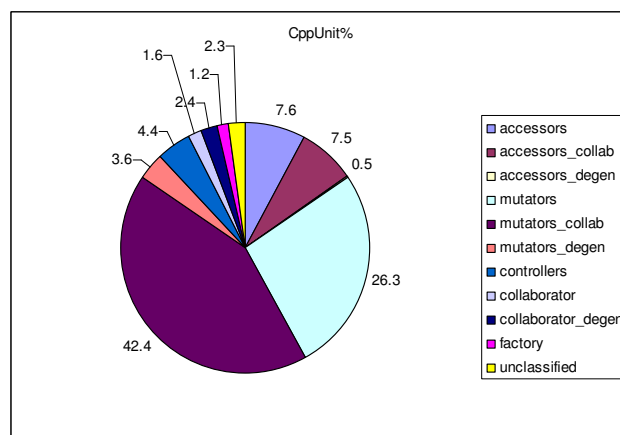
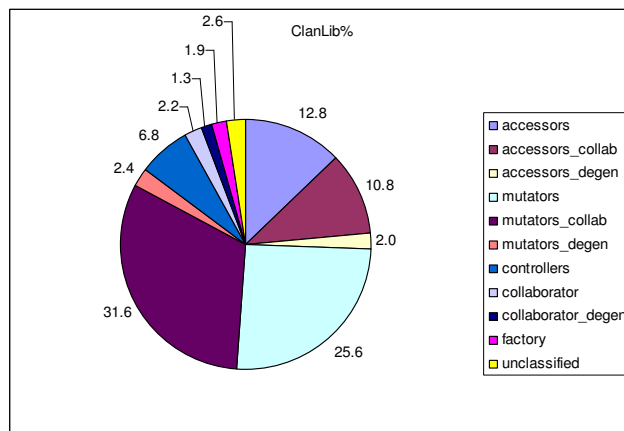
APPENDIX A

METHOD STEREOTYPES DISTRIBUTIONS IN THE CLASSIFIED SYSTEMS

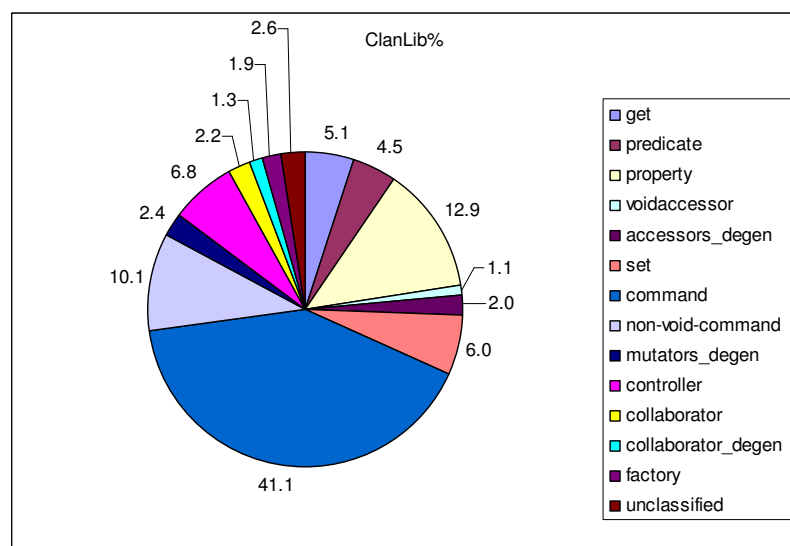
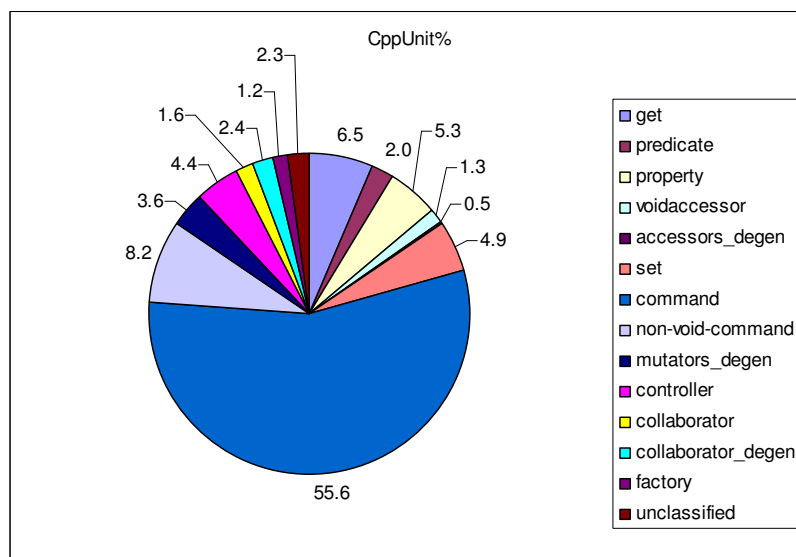
This appendix gives both Stereotype Perspective and Stereotype Category Perspective of method stereotypes distributions for all 21 systems used in the case study of classifying software.

A.1 Distributions for systems in Mutator Pattern

A.1.1 Stereotype Category Perspective

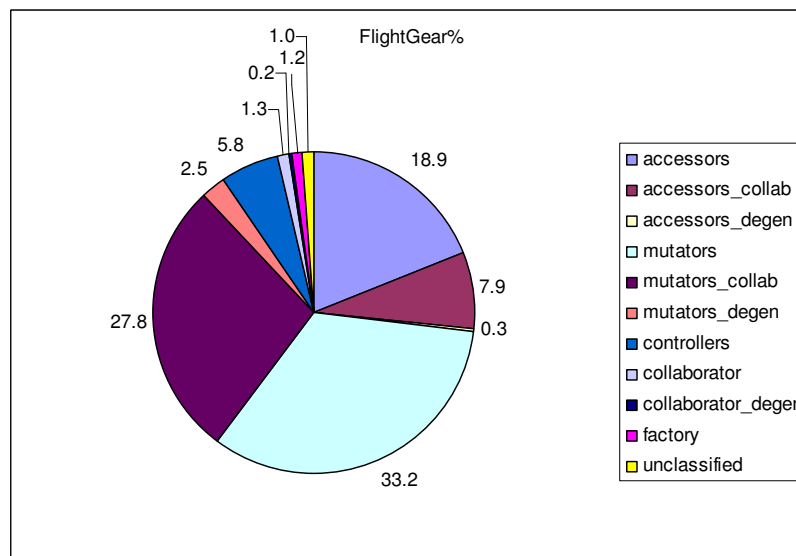
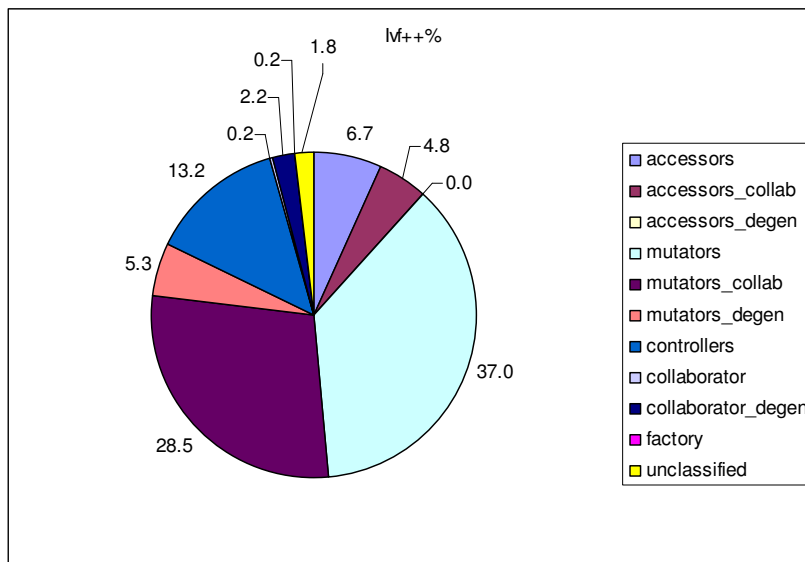


A.1.2 Stereotype Perspective

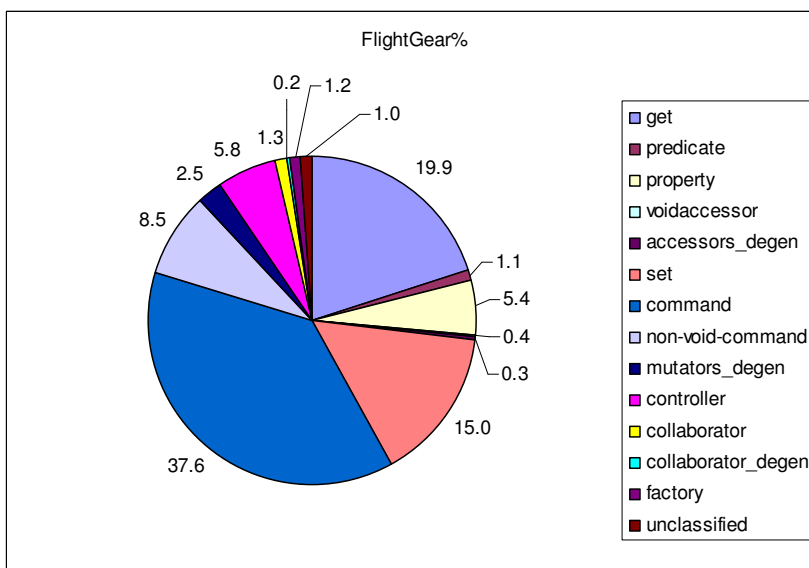
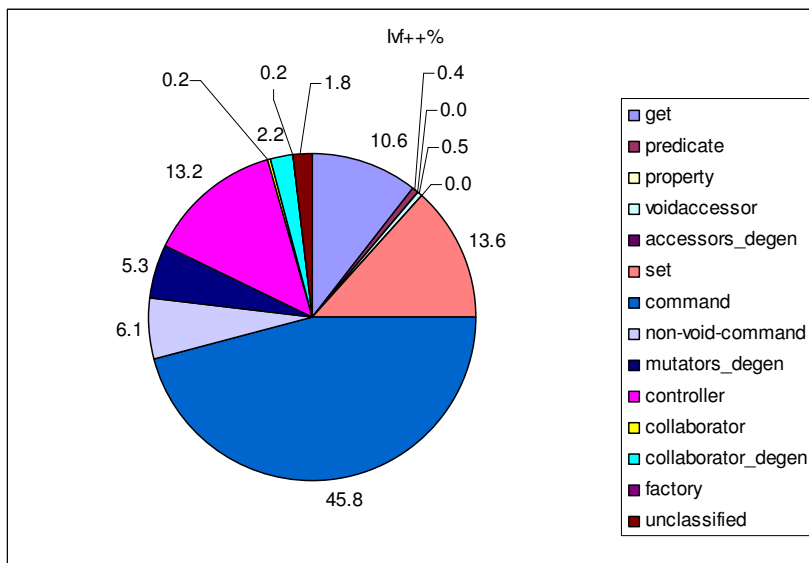


A.2 Distributions for systems in Mutator-Data Storage Pattern

A.2.1 Stereotype Category Perspective

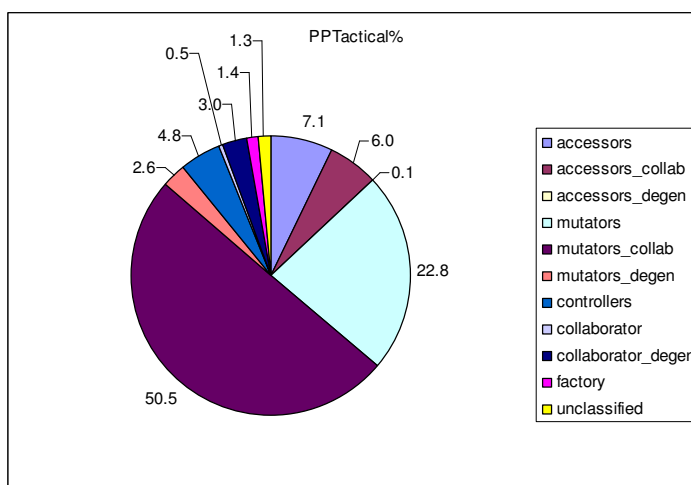
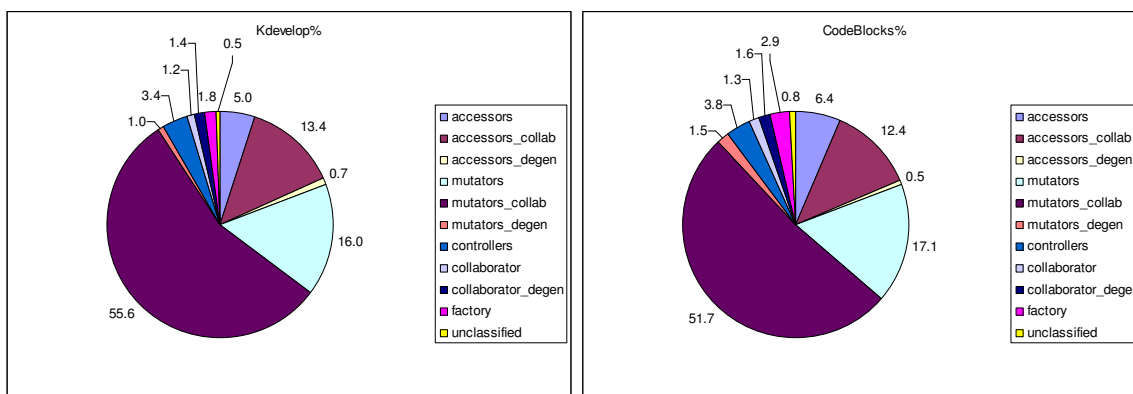


A.2.2 Stereotype Perspective

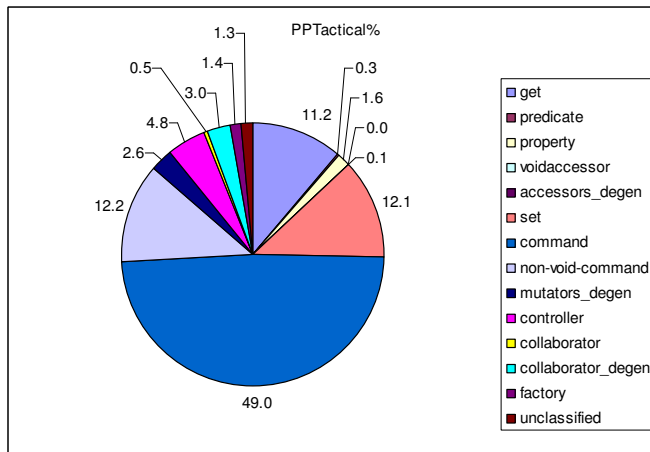
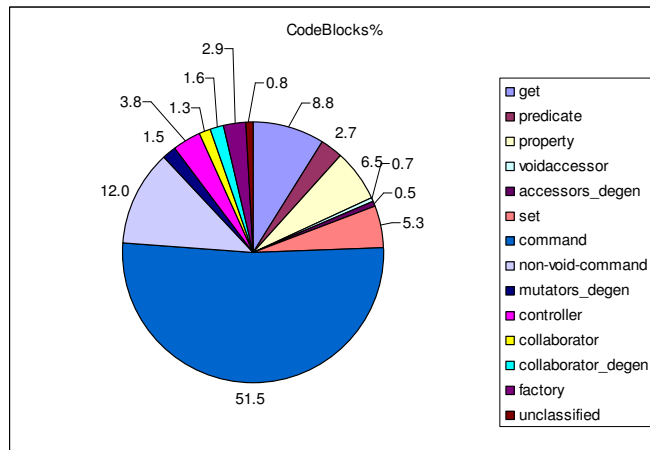
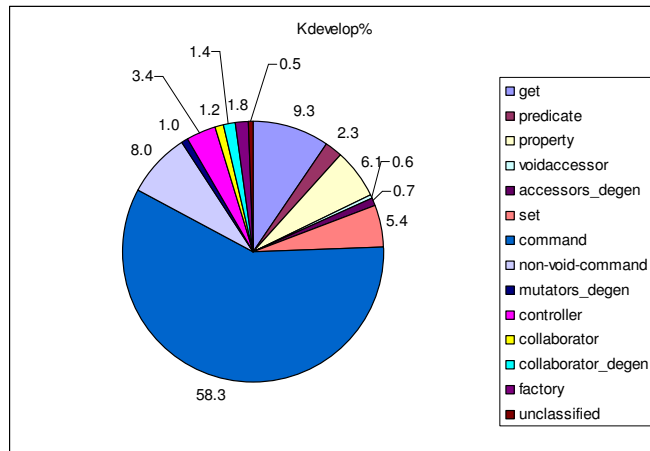


A.3 Distributions for systems in Mutator-Collaborator Pattern

A.3.1 Stereotype Category Perspective

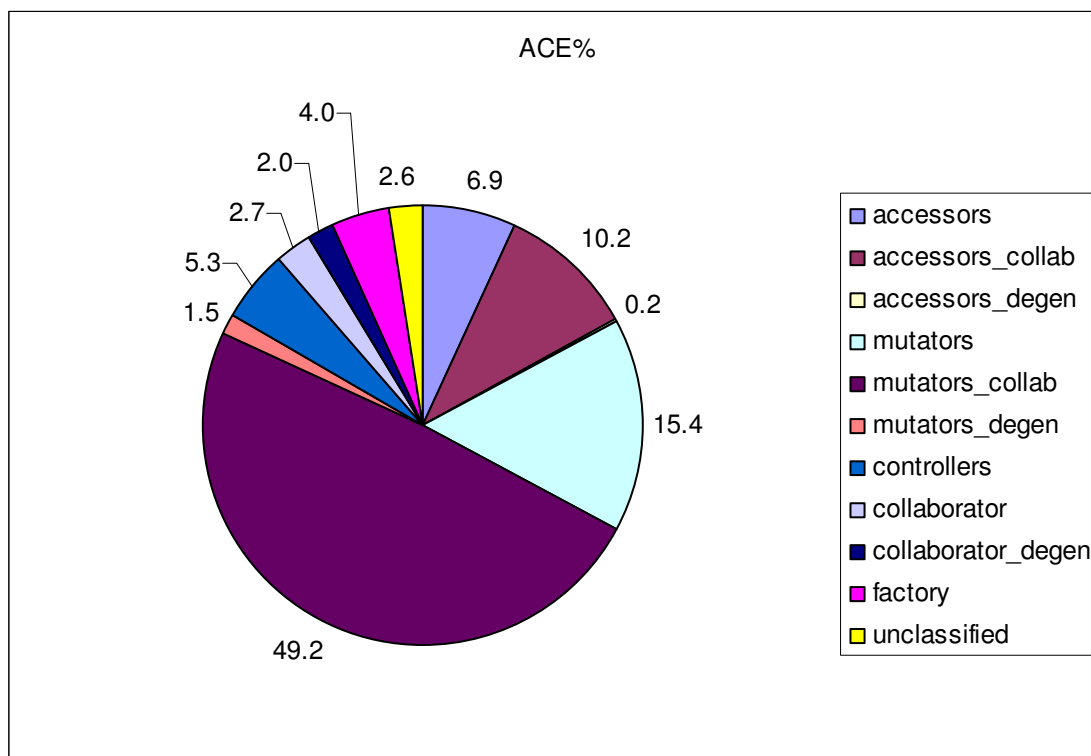


A.3.2 Stereotype Perspective

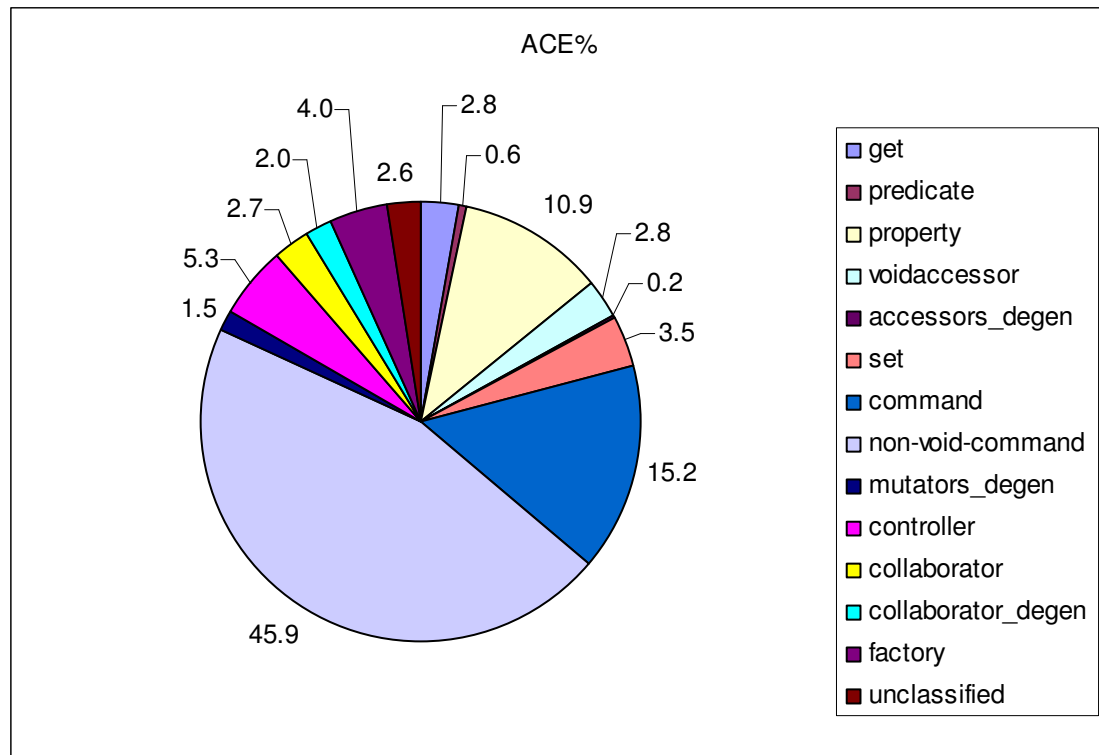


A.4 Distributions for systems in Non-void-Mutator-Collaborator Pattern

A.4.1 Stereotype Category Perspective

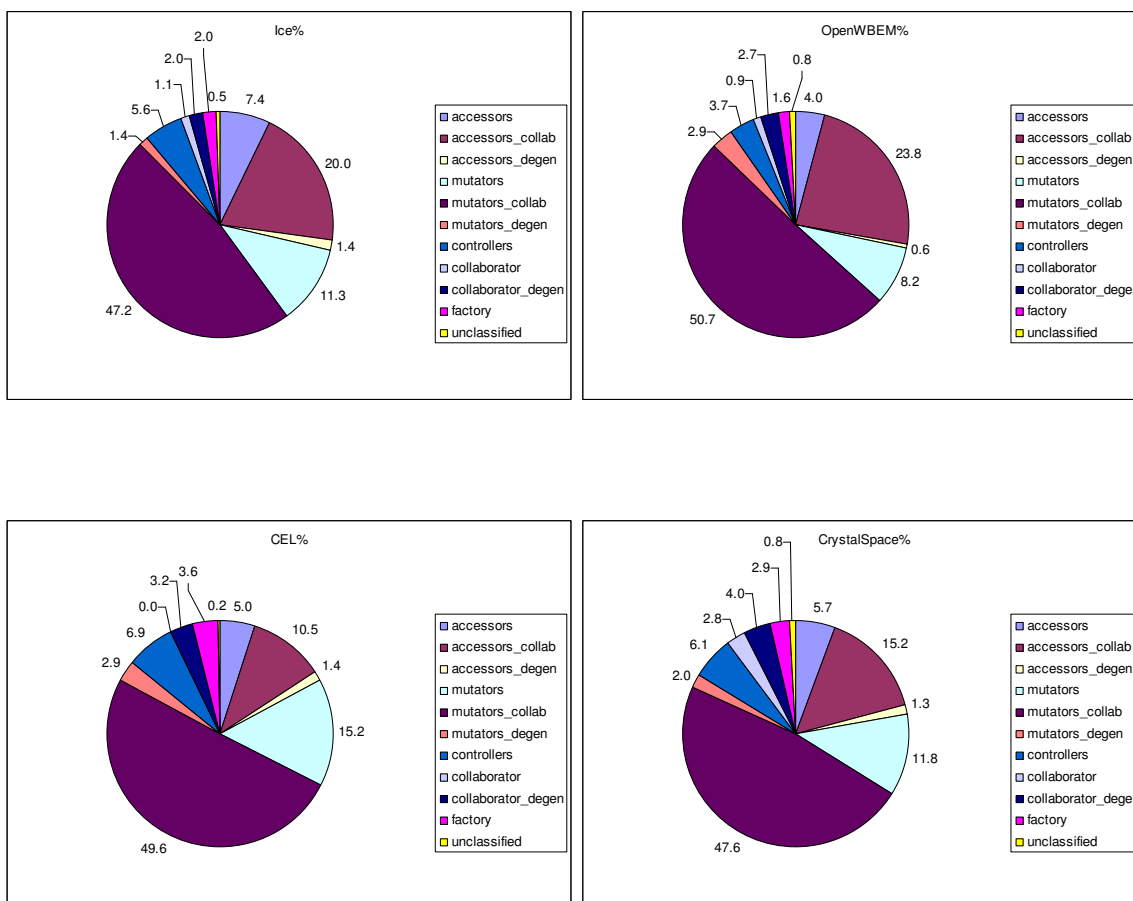


A.4.2 Stereotype Perspective

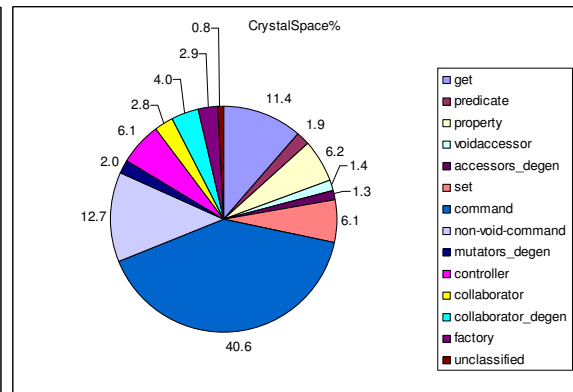
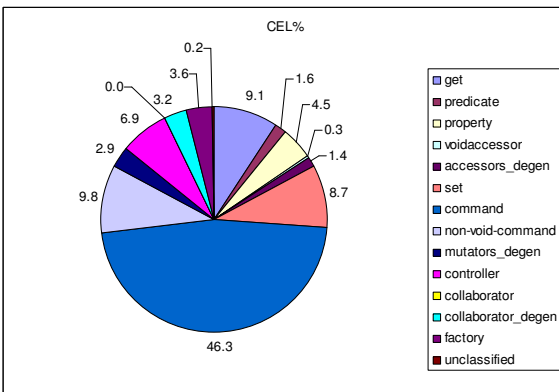
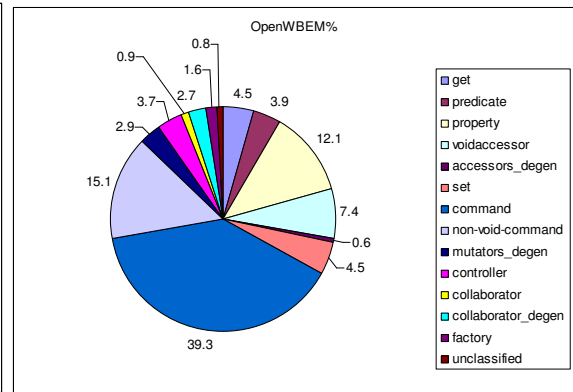
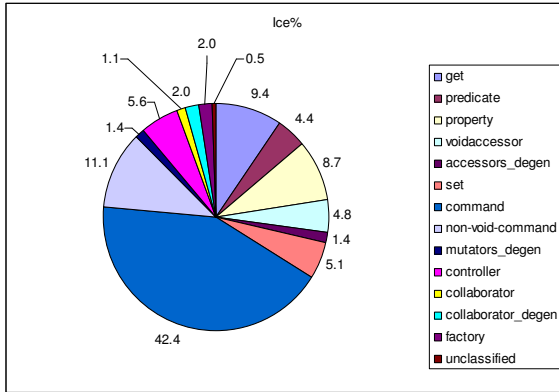


A.5 Distributions for systems in Mutator-Accessor-Collaborator Pattern

A.5.1 Stereotype Category Perspective

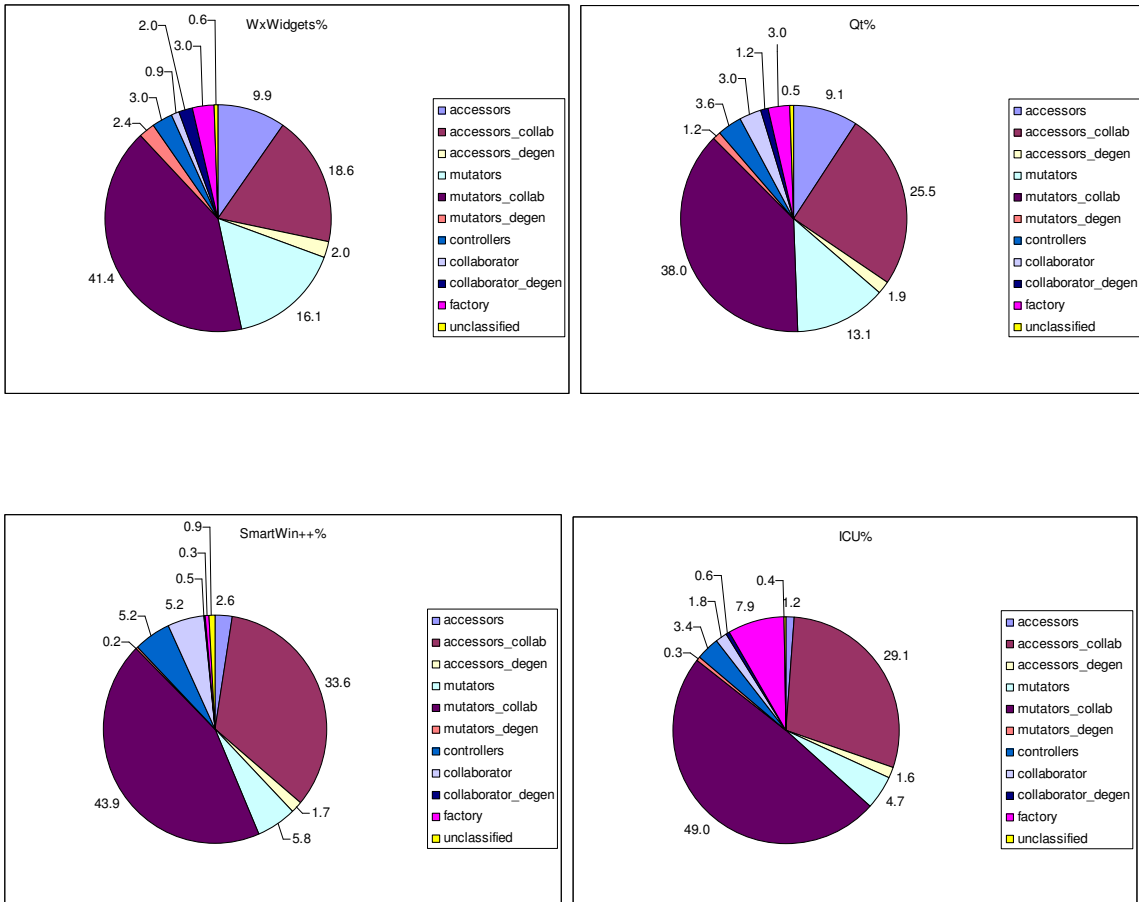


A.5.2 Stereotype Perspective

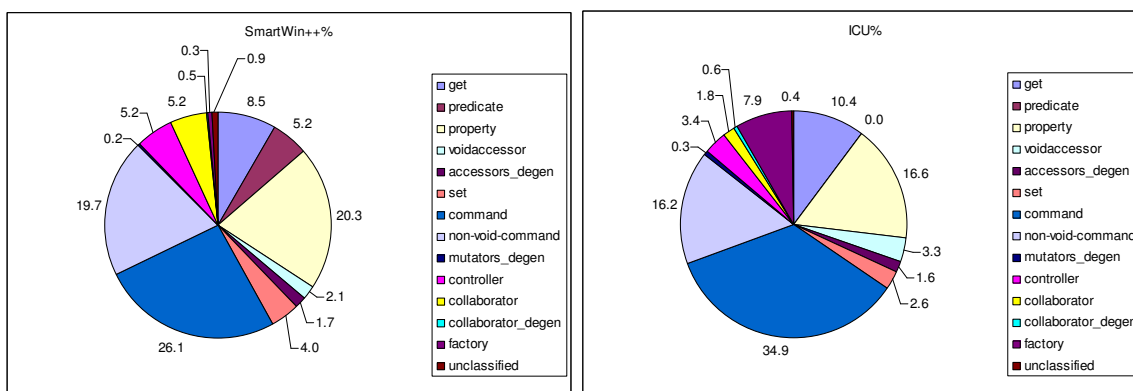
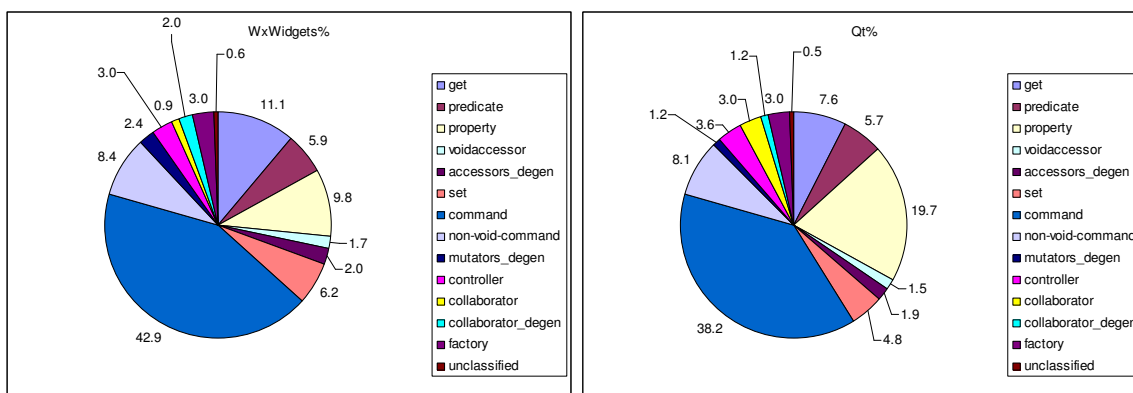


A.6 Distributions for systems in Accessor-Mutator-Collaborator Pattern

A.6.1 Stereotype Category Perspective

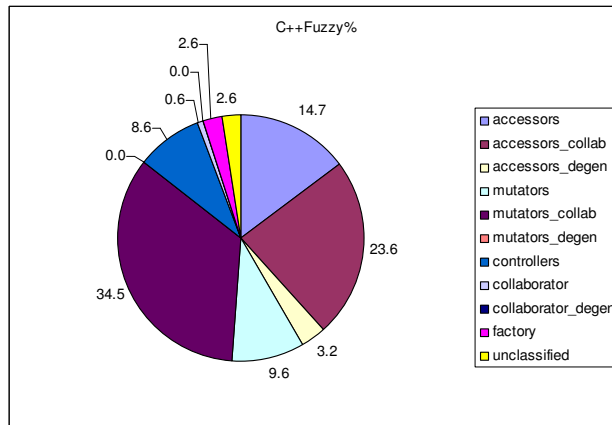
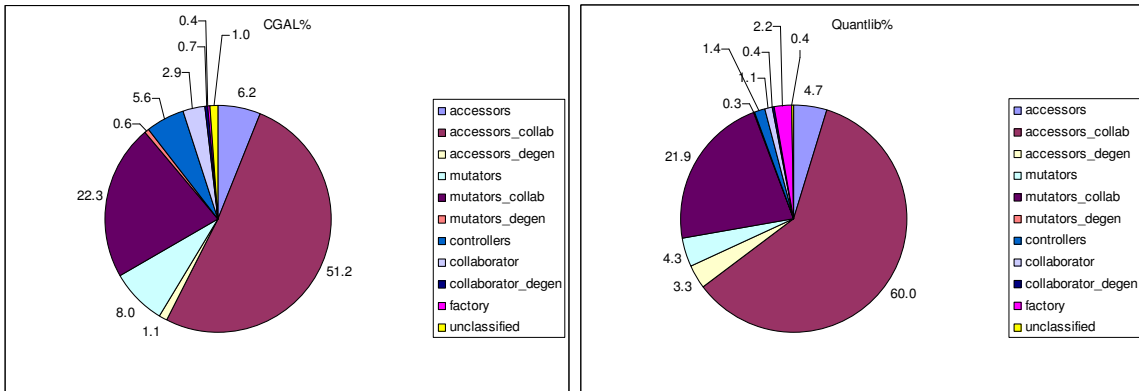


A.6.2 Stereotype Perspective

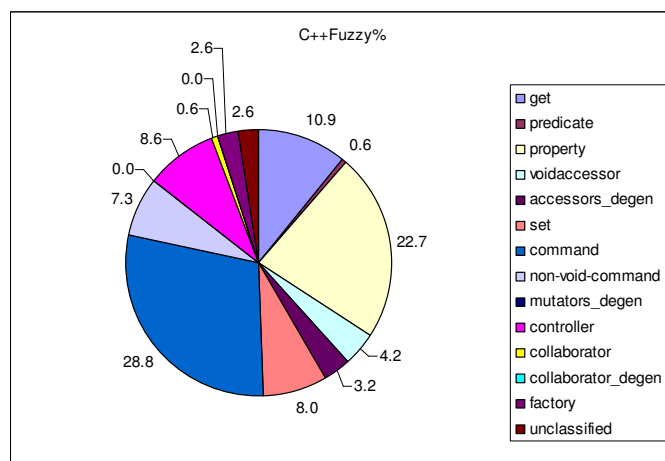
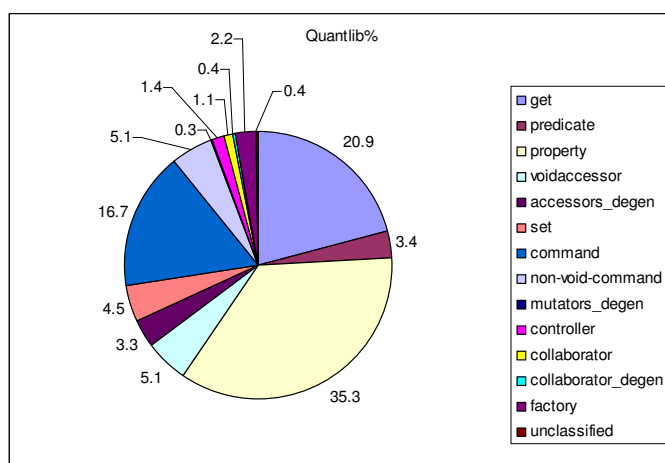
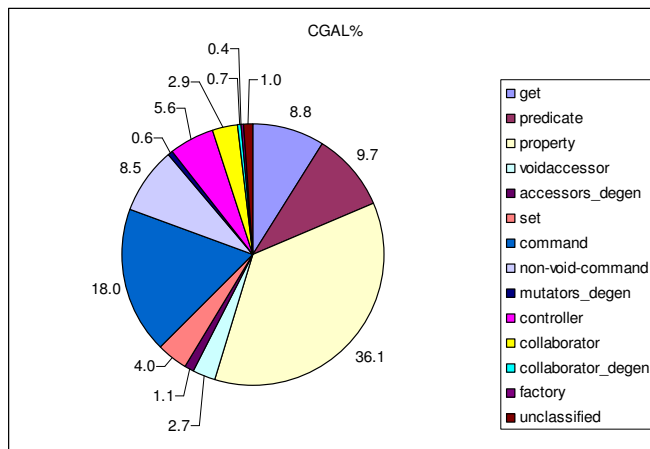


A.7 Distributions for systems in Accessor-Collaborator Pattern

A.7.1 Stereotype Category Perspective

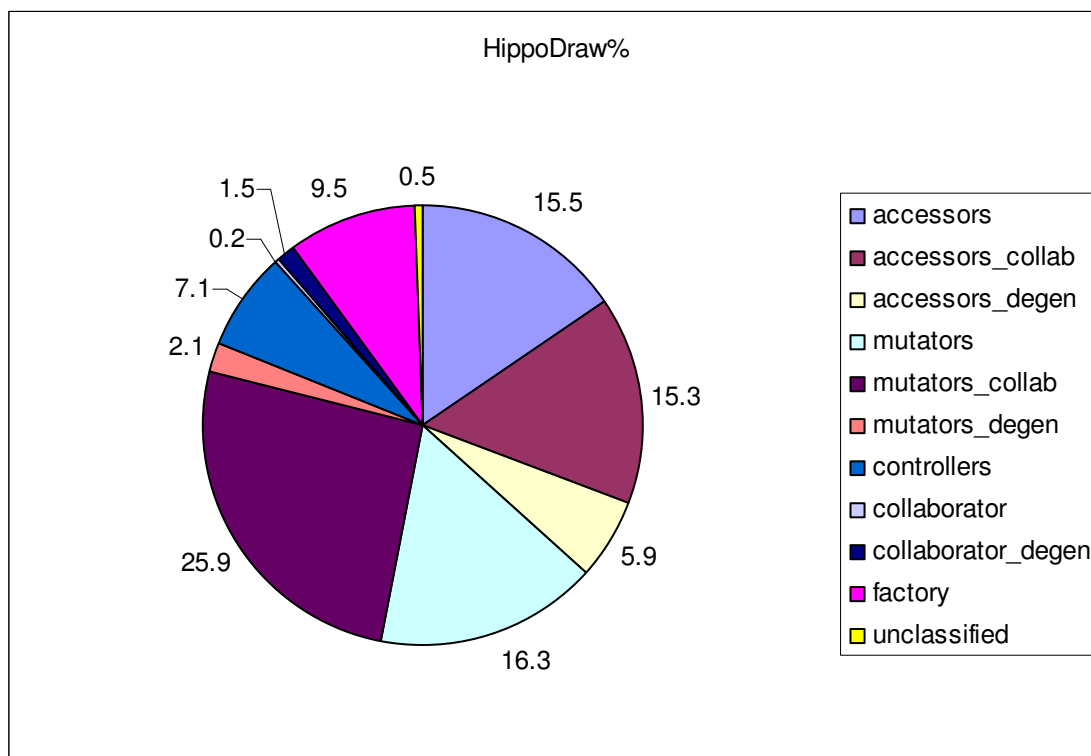


A.7.2 Stereotype Perspective

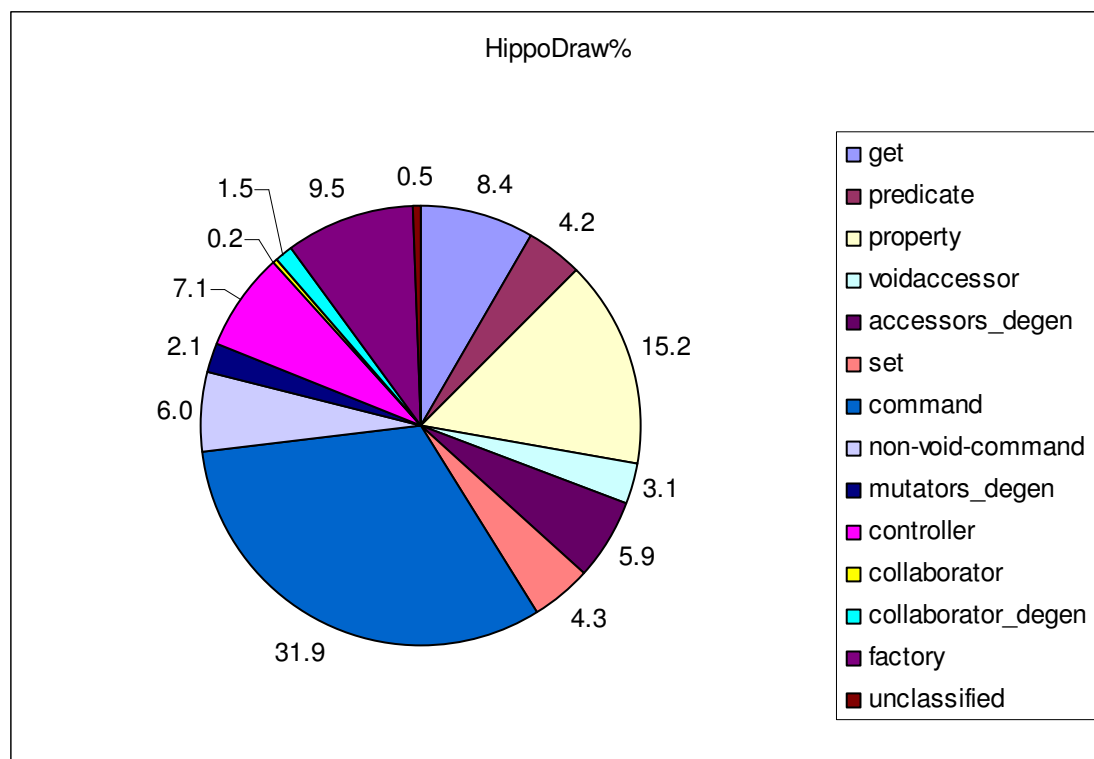


A.8 Distributions for systems in Accessor-Mutator-Controller Pattern

A.8.1 Stereotype Category Perspective

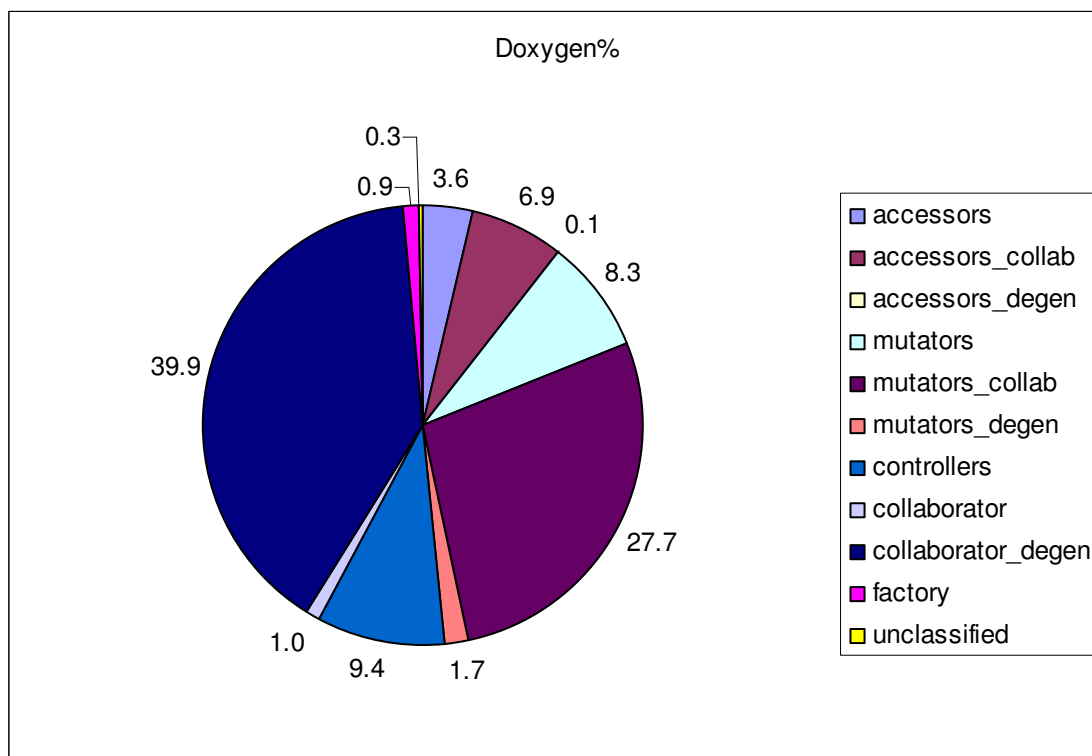


A.8.2 Stereotype Perspective

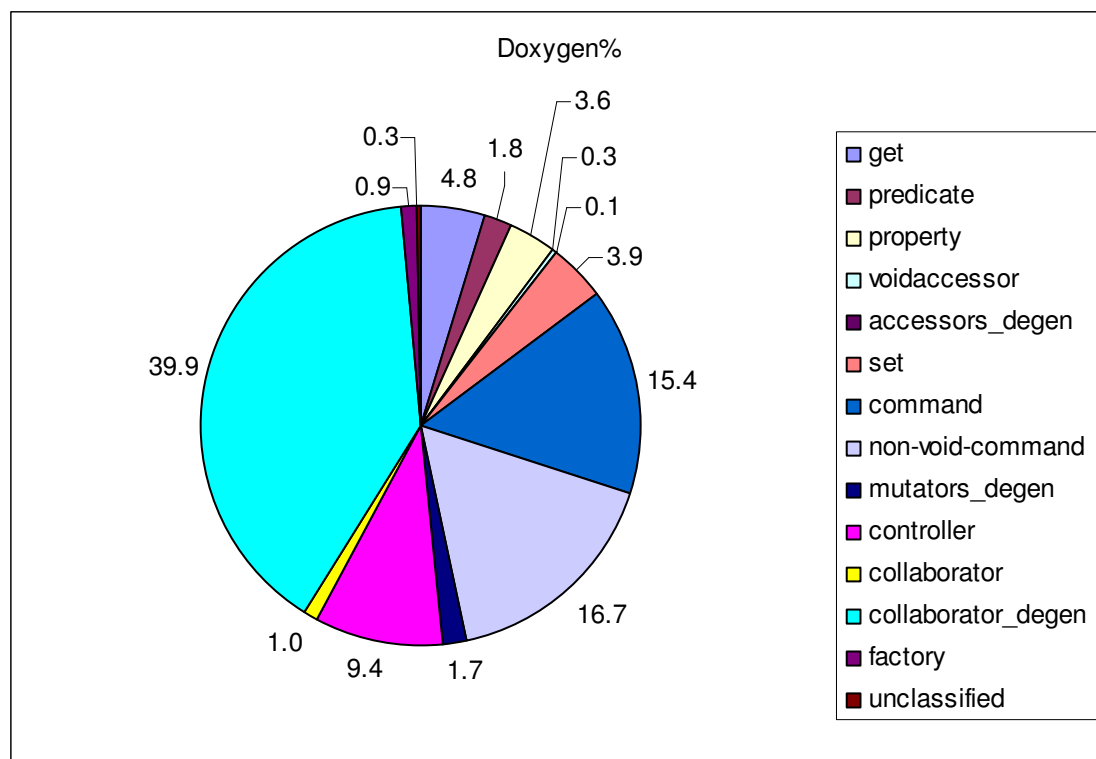


A.9 Distributions for systems in Controller-Collaborator Pattern

A.9.1 Stereotype Category Perspective



A.9.2 Stereotype Perspective



APPENDIX B

SOURCE CODE OF HIPPODRAW CLASSES REDOCUMENTED BY THE *STEREOCODE* AND *STEREOCLASS* TOOLS

This appendix shows source code of the HippoDraw classes redocumented with the method stereotypes and their distributions. Comments added to the source files by the *StereoCode* and *StereoClass* tools are shown in bold.

B.1 Entity - class Range

```
/* Class report: Method stereotypes distribution
Range, get, 3
Range, predicate, 2
Range, property, 3
Range, set, 4
Range, command, 4
Range, command collaborator, 1
*/
/** @file
hippodraw::Range class interface
Copyright (C) 2000-2004, 2006 The Board of Trustees of The
Leland Stanford Junior University. All Rights Reserved.
...
*/
...
class MDL_HIPPOPLOT_API Range {

private:
    /** The minimum in the range.
    */
    double m_min;
    ... //more declarations here

/** @stereotype property */
inline
double
Range::
length () const
```



```

{
    return (m_max - m_min);
}

template < class Iterator >
/** @stereotype command collaborator */
void
Range::
setRange ( Iterator first, Iterator end )
{
    m_min = DBL_MAX;
    m_max = -DBL_MAX;
    m_pos = DBL_MAX;

    while ( first != end ) {
        double test = *first; // input might be pointer to float.
        m_min = std::min ( m_min, test );
        m_max = std::max ( m_max, test );
        if ( test > 0.0 ) m_pos = std::min ( m_pos, test );
        ++first;
    }
}

/** @file
hippodraw::Range class implementation
...
*/
... //constructors here
/** @stereotype get */
double
Range::low() const
{
    return m_min;
}
/** @stereotype set */
void
Range::setLow ( double x )
{
    m_min = x;
    assert ( m_min <= m_max );
}
/** @stereotype get */
double
Range::high() const
{
    return m_max;
}
/** @stereotype set */

```

```

void
Range::setHigh ( double x )
{
    m_max = x;
    assert ( m_min <= m_max );
}
/** @stereotype get */
double
Range::pos() const
{
    return m_pos;
}
/** @stereotype set */
void
Range::setPos (double x)
{
    m_pos = x;
    assert ( m_min <= m_max );
}
/** @stereotype command */
void
Range::setRange ( double low, double high, double pos )
{
    m_min = low;
    m_max = high;
    m_pos = pos;
    assert ( m_min <= m_max );
}
/** @stereotype command */
void
Range::setLength ( double val, bool high_hold )
{
    if( high_hold ){
        m_min = m_max - val;
    } else {
        m_max = m_min + val;
    }
    assert ( m_min <= m_max );
}
/** @stereotype predicate */
bool
Range::includes ( double val ) const
{
    return val >= m_min && val <= m_max;
}
/** @stereotype predicate */
bool
Range::

```

```

excludes ( double value ) const
{
    return value < m_min || value > m_max;
}
/** @stereotype property */
double
Range::
fraction ( double value ) const
{
    return ( value - m_min ) / ( m_max - m_min );
}
/** @stereotype set */
void Range::setEmpty ( bool yes )
{
    m_empty = yes;
}
/** @stereotype command */
void Range::setUnion ( const Range & range )
{
    if ( m_empty ) {
        m_min = range.m_min;
        m_max = range.m_max;
        m_pos = range.m_pos;
        m_empty = false;
    }
    else {
        m_min = min ( m_min, range.m_min );
        m_max = max ( m_max, range.m_max );
        m_pos = min ( m_pos, range.m_pos );
    }
    assert ( m_min <= m_max );
}
/** @stereotype command */
void Range::setIntersect ( const Range & range )
{
    if ( m_min > range.m_max || m_max < range.m_min ) return;
    m_min = max ( m_min, range.m_min );
    m_max = min ( m_max, range.m_max );
    m_pos = max ( m_pos, range.m_min );

    assert ( m_min <= m_max );
}
/** @stereotype property */
int
Range::numberOfBins ( double width ) const
{
    assert ( m_max > m_min );
}

```

```

    double number = (m_max - m_min) / width;

#ifdef _MSC_VER
    return static_cast < int > ( number+0.5 );
#else
    return static_cast < int > ( rint( number ) );
#endif
}

```

B.2 Minimal Entity - class Point

```

/* Class report: Method stereotypes distribution
Point, get, 3
Point, set, 1
Point, command, 3
*/
/** @file
Point class interface
...
*/
...
class MDL_HIPPOPLOT_API Point
{
private:
    double m_x;
    double m_y;
    double m_z;
    ... // more declarations here
/** @stereotype get */
inline
double
Point::
getX() const
{
    return m_x;
}
/** @stereotype get */
inline
double
Point::
getY() const
{
    return m_y;
}
/** @stereotype get */
inline double
Point::
getZ() const

```

```

{
    return m_z;
}
/*
 * HippoPlot Point class implementation
 ...
 */
...
... // constructors here
/** @stereotype command */
void Point::setPoint( double x, double y ) {
    m_x = x;
    m_y = y;
}
/** @stereotype command */
void Point::setPoint( double x, double y, double z ) {
    m_x = x;
    m_y = y;
    m_z = z;
}
/** @stereotype command */
void Point::moveBy ( double x, double y )
{
    m_x += x;
    m_y += y;
}
/** @stereotype set */
void Point::setZ ( double z )
{
    m_z = z;
}

```

B.3 Data Provider (and Entity) - class BinnerAxis

```

/* Class report: Method stereotypes distribution
BinnerAxis, get, 2
BinnerAxis, get collaborator, 1
BinnerAxis, property, 3
BinnerAxis, void-accessor, 2
BinnerAxis, non-void-command, 1
BinnerAxis, predicate incidental, 1
*/
/** @file
hippodraw::BinnerAxis class implementation
...
*/
... // constructors here
/** @stereotype get */

```

```

const string &
BinnerAxis::
name () const
{
    return m_name;
}
/** @stereotype predicate incidental */
bool
BinnerAxis::hasEqualWidths () const
{
    return false;
}
/** @stereotype property */
double
BinnerAxis::axisGetLow() const
{
    return m_range.low();
}
/** @stereotype property */
double
BinnerAxis::axisGetHigh() const
{
    return m_range.high();
}
/** @stereotype get collaborator */
const Range &
BinnerAxis::
getRange() const
{
    return m_range;
}
/** @stereotype get */
int
BinnerAxis::axisNumberOfBins () const
{
    return m_num_bins;
}
/** @stereotype void-accessor */
void
BinnerAxis::setStartRange ( bool dragging ) const
{
    if ( m_dragging == false ) {
        m_range_start = m_range;
    }

    m_dragging = dragging;
}
/** @stereotype void-accessor */

```

```

void
BinnerAxis::setStartWidth ( bool dragging ) const
{
    if ( m_dragging == false ) {
        m_width_start = m_width;
    }

    m_dragging = dragging;
}
/** @stereotype non-void-command */
const vector< double > & BinnerAxis::binEdges ()
{
    if( m_bin_edges.size() == 0 )
    {
        m_bin_edges.resize( m_num_bins + 1);

        m_bin_edges[0] = m_range.low();

        for( int i = 0; i < m_num_bins; i ++ )
            m_bin_edges[i] = m_bin_edges[i-1] + axisBinWidth( i );

        m_bin_edges[ m_num_bins + 1 ] = m_range.high();
    }

    return m_bin_edges;
}
/** @stereotype property */
double
BinnerAxis::
calcBinWidth ( int parm, bool dragging ) const
{
    setStartWidth ( dragging );

    double multiplier = ( 50 - parm ) / 50.0;
    int num_start = getNob ( m_width_start );
    if ( num_start == 1 ) {
        multiplier *= 4.0;
    }
    double num_new = num_start + num_start * multiplier;

    num_new = std::max ( 1.0, num_new );
    m_num_bins = static_cast < int > ( num_new );
    double new_width = calcWidthParm ( m_num_bins );

    return new_width;
}

```

B.4 Commander (and Boundary) - class DrawBorder

```

/* Class report: Method stereotypes distribution

DrawBorder, get collaborator, 1
DrawBorder, set collaborator, 1
DrawBorder, command, 1
DrawBorder, command collaborator, 1
*/
/** @file
hippodraw::DrawBorder class implementation
...
*/
... // constructors here
/** @stereotype set collaborator */
void DrawBorder::setView (ViewBase * view)
{
    m_view = view;
}
/** @stereotype get collaborator */
ViewBase * DrawBorder::getView ()
{
    return m_view;
}
/** @stereotype command collaborator */
void DrawBorder::draw()
{
    Rect rect = m_view->getDrawRect();
    double width = rect.getWidth();
    double height = rect.getHeight();

    width = width - 2;
    height = height - 2;

    vector <double> vx (8);
    vector <double> vy (8);

    vx [0] = 2;
    vy [0] = 2;
    vx [1] = width;
    vy [1] = 2;

    vx [2] = width;
    vy [2] = 2;
    vx [3] = width;
    vy [3] = height;
}

```



```

vx [4] = width;
vy [4] = height;
vx [5] = 2;
vy [5] = height;

vx [6] = 2;
vy [6] = height;
vx [7] = 2;
vy [7] = 2;

m_view->drawViewLines ( vx, vy, Line::Solid, Color(180, 180,
180), 0 );

// Now draw the knobs.
drawKnob ( 2, 2); //Upper Right.
drawKnob ( width / 2, 2); //Upper Middle.
drawKnob ( width, 2); //Upper Left.

drawKnob ( 2, height / 2); //Middle Left.
drawKnob ( width, height / 2); //Middle Right.

drawKnob ( 2, height); //Lower left.
drawKnob ( width / 2, height); //Lower middle.
drawKnob ( width, height); //Lower right.
}
/** @stereotype command */
void DrawBorder::drawKnob( double x, double y)
{
    int size = 2;

    vector <double> vx (8);
    vector <double> vy (8);

    vx [0] = x-size;
    vy [0] = y-size;
    vx [1] = x+size;
    vy [1] = y-size;

    vx [2] = x+size;
    vy [2] = y-size;
    vx [3] = x+size;
    vy [3] = y+size;

    vx [4] = x+size;
    vy [4] = y+size;
    vx [5] = x-size;

```

```

    vy [5] = y+size;

    vx [6] = x-size;
    vy [6] = y+size;
    vx [7] = x-size;
    vy [7] = y-size;

    m_view->drawViewLines ( vx, vy, Line::Solid, Color(180, 180,
180), 0 );
}

```

B.5 Boundary (and Data Provider) - class DataView

```

/* Class report: Method stereotypes distribution
DataView, get collaborator, 1
DataView, property collaborator, 9
DataView, set collaborator, 1
DataView, command collaborator, 1
*/
/** @file
hippodraw::DataView class implementation
...
*/
... // constructors here
/** @stereotype get collaborator */
const Rect &
DataView::
getMarginRect () const
{
    return m_margin_rect;
}
/** @stereotype set collaborator */
void
DataView::
setMarginRect ( const Rect & rect )
{
    m_margin_rect = rect;
}
/** @attention In the implementation, make sure the left margin
stops
    growing at same time maximum Y tick labels stop growing.
*/
/** @stereotype command collaborator */
void
DataView::
prepareMarginRect ( )
{

```

```

const Rect draw = getDrawRect();
float width = draw.getWidth();
float height = draw.getHeight();

float marginXLeft = draw.getHeight () * 0.20;
marginXLeft = std::min ( marginXLeft, 55.0f );
float marginXRight = 20.0 ;

// Get a pointer to the plotter.
PlotterBase* plotter = getPlotter();

// Set and adjust top margin
float marginYTop = 30.0;
if ( m_plotter -> hasAxis ( Axes::Z ) )
{
    marginYTop = 70.0;
}
const FontBase* titlefont = plotter->titleFont();
if (titlefont) {
    marginYTop = marginYTop+titlefont->pointSize()-9.0;
}
const FontBase* zfont = plotter->labelFont(Axes::Z);
if (zfont) {
    marginYTop = marginYTop+zfont->pointSize()-7.0;
}

// Set and adjust bottom margin
float marginYBottom = 34.0 ;
const FontBase* labelfont = plotter->labelFont(Axes::X);
if (labelfont) {
    marginYBottom = marginYBottom+labelfont->pointSize()-11.0;
}

// Add additional margins. Now it can be added by PNG title,
labels.
marginYTop+=plotter->getTopMargin()+plotter->getZMargin();
marginYBottom+=plotter->getBottomMargin();
marginXLeft+=plotter->getLeftMargin();

double aspect_ratio = m_plotter -> getAspectRatio ();

float marginWidth = width - marginXLeft - marginXRight;
float marginHeight =height - marginYTop - marginYBottom;

if ( aspect_ratio > 0.0 ) {
    if ( marginWidth /aspect_ratio > marginHeight ){
        marginWidth = aspect_ratio*marginHeight;
    }
}

```

```

    else {
        marginHeight = marginWidth/aspect_ratio;
    }
}

m_margin_rect.setRect ( marginXLeft, marginYTop,
                        marginWidth, marginHeight );
}
/** @stereotype property collaborator */
float
DataView::
userToMarginX ( double x ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();
    double diff = x - user_rect.getX ();
    double scale = m_margin_rect.getWidth() / user_rect.getWidth
();
    double margin_x = m_margin_rect.getX () + diff * scale;
    return margin_x;
}
/** @stereotype property collaborator */
float
DataView::
userToInvertedMarginX ( double x ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();
    double diff = x - user_rect.getX ();
    double scale = m_margin_rect.getWidth() / user_rect.getWidth
();
    double margin_ix = m_margin_rect.getX() +
m_margin_rect.getWidth() - diff*scale;
    return margin_ix;
}
/** @stereotype property collaborator */
float
DataView::
userToMarginY ( double y ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();
    return m_margin_rect.getY ()
+ ( y - user_rect.getY () )
* m_margin_rect.getHeight () / user_rect.getHeight ();
}
/** @stereotype property collaborator */
float
DataView::
userToInvertedMarginY ( double y ) const
{

```

```

const Rect & user_rect = m_plotter -> getUserRect ();
return m_margin_rect.getY ()
    + m_margin_rect.getHeight ()
    - ( y - user_rect.getY () )
    * m_margin_rect.getHeight () / user_rect.getHeight ();
}

/** @stereotype property collaborator */
float
DataView::
userToMarginColor ( double c ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();

    return m_margin_rect.getX ()
        + ( c - user_rect.getZ () )
        * m_margin_rect.getWidth () / user_rect.getDepth ();
}

/** @stereotype property collaborator */
double
DataView::
marginToUserX ( double x ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();

    return user_rect.getX ()
        + ( x - m_margin_rect.getX() )
        / ( m_margin_rect.getWidth () / user_rect.getWidth() );
}

/** @stereotype property collaborator */
double
DataView::
marginToInvertedUserX ( double x ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();

    return user_rect.getX ()
        + ( m_margin_rect.getX() + m_margin_rect.getWidth() - x )
        / ( m_margin_rect.getWidth () / user_rect.getWidth() );
}

/** @stereotype property collaborator */
double
DataView::
marginToUserY ( double y ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();

    return user_rect.getY ()

```

```

    + ( y - m_margin_rect.getY() )
    / ( m_margin_rect.getHeight () * user_rect.getHeight () );
}
/** @stereotype property collaborator */
double
DataView::
marginToInvertedUserY ( double y ) const
{
    const Rect & user_rect = m_plotter -> getUserRect ();

    return user_rect.getY () +
        ( m_margin_rect.getY()
          + m_margin_rect.getHeight() - y )
        / ( m_margin_rect.getHeight () / user_rect.getHeight () ) ;
}

```

B.6 Factory - class QtViewFactory

```

/* Class report: Method stereotypes distribution
QtViewFactory, collaborator factory, 3
*/
/** @file
QtViewFactory implemenatation
...
*/
... // constructor here
/** @stereotype collaborator factory*/
ViewFactory * QtViewFactory::instance ()
{
    if ( m_instance == 0 ) {
        m_instance = new QtViewFactory ();
    }

    return m_instance;
}
/** @stereotype collaborator factory */
ViewBase *
QtViewFactory::
createView ( PlotterBase * plotter ) const
{
    QtView * view = new QtView ( plotter );

    return view;
}
/** @stereotype collaborator factory */
FontBase *
QtViewFactory::
createFont () const

```

```

{
    return new QFont ();
}

```

B.7 Controller - class DisplayController

This class is too big (1778 LOC, 93 methods) - only the class report is shown.

```

/* Class report: Method stereotypes distribution
DisplayController, get collaborator, 2
DisplayController, predicate collaborator, 2
DisplayController, property collaborator, 2
DisplayController, void-accessor collaborator, 5
DisplayController, command collaborator, 15
DisplayController, non-void-command collaborator, 4
DisplayController, controller, 45
DisplayController, collaborator factory, 18
*/

```

B.8 Pure Controller (and Factory) - class BinnerAxisXML

```

/* Class report: Method stereotypes distribution
BinnerAxisXML, controller, 1
BinnerAxisXML, collaborator factory, 2
*/
/** @file
BinnerAxisXML implementation
...
*/
... // constructors here
/** @stereotype collaborator factory */
XmlElement *
BinnerAxisXML::
createElement ( const BinnerAxis & binner )
{
    XmlElement * tag = BaseXML::createElement ();
    setAttributes ( tag, binner );

    return tag;
}
/** @stereotype controller */
void
BinnerAxisXML::
setAttributes ( XmlElement * tag, const BinnerAxis & binner )
{
    const string & type = binner.name();
    tag->setAttribute ( m_type, type );
}

```

```

const Range & range = binner.getRange ();
double high = range.high();
double low = range.low ();

tag->setAttribute ( m_high, high );
tag->setAttribute ( m_low, low );

double width = binner.getBinWidth ();
tag->setAttribute ( m_width, width );
}
/** @stereotype collaborator factory */
BinnerAxis *
BinnerAxisXML::
createObject ( const XmlElement * element )
{
    string type;
    bool ok = element->attribute ( m_type, type );
    assert ( ok );

    BinnerAxisFactory * factory = BinnerAxisFactory::instance ();
    BinnerAxis * binner = factory->create ( type );

    double high = 1.0;
    double low = 0.0;
    ok = element->attribute ( m_high, high );
    ok &= element->attribute ( m_low, low );
    assert ( ok );
    Range range ( low, high );
    binner->setRange ( range, false );

    double width = -1.0;
    ok = element->attribute ( m_width, width );
    assert ( ok );
    binner->setBinWidth ( width );

    return binner;
}

```

B.9 Pure Controller (and Small) - class AxisTickXML

```

/* Class report: Method stereotypes distribution
AxisTickXML, controller, 2
*/
/** @file
AxisTickXML class implementation
...
*/

```



```

... // constructors here
/** @stereotype controller */
void
AxisTickXML::
setAttributes ( XmlElement & tag,
               const AxisTick & tick )
{
    double value = tick.value ();
    tag.setAttribute ( m_value, value );

    const string & label = tick.content ();
    tag.setAttribute ( m_label, label );
}
/** @stereotype controller */
void
AxisTickXML::
setAttributes ( AxisTick * tick,
               const XmlElement * element )
{
    double value;
    bool ok = element -> attribute ( m_value, value );
    tick -> setValue ( value );

    string label;
    ok = element -> attribute ( m_label, label );
    tick -> setContent ( label );
}

```

B.10 Large Class (and Boundary) - class FunctionController

This class is too big (1212 LOC, 52 methods) - only the class report is shown.

```

/* Class report: Method stereotypes distribution
FunctionController, void-accessor collaborator, 2
FunctionController, command collaborator, 17
FunctionController, non-void-command collaborator, 9
FunctionController, controller, 10
FunctionController, collaborator factory, 13
*/

```

B.11 Lazy Class - class BinsBase

```

/* Class report: Method stereotypes distribution
BinsBase, get, 3
BinsBase, set, 3
BinsBase, incidental, 3

```

```

BinsBase, empty, 1
*/
/** @file
BinsBase  class implementation
...
*/
... // constructors here
/** @stereotype get */
const string &
BinsBase::
name () const
{
    return m_name;
}
/** @stereotype get */
bool BinsBase::isDirty ()
{
    return m_values_dirty;
}
/** @stereotype set */
void BinsBase::setDirty()
{
    m_values_dirty = true;
}
/** @stereotype incidental */
double BinsBase::scaleFactor () const
{
    return 1.0;
}
/** @stereotype incidental */
double BinsBase::getZValue ( double, double ) const
{
    return 0;
}
/** @stereotype get */
bool
BinsBase::
isEmpty () const
{
    return m_empty;
}
/** @stereotype set */
void
BinsBase::
scaleNumberOfEntries ( double number )
{
    m_scale_factor = number;
}

```

```

/** @stereotype set */
void
BinsBase::
setEntriesScaling ( bool on )
{
    m_is_scaling = on;
}
/** @stereotype empty */
void
BinsBase::
setMinEntries ( int entries )
{
}
/** @stereotype incidental */
int
BinsBase::
getMinEntries ()
{
    return -1;
}

```

B.12 Degenerate Class – class AxisRep2D

```

/* Class report: Method stereotypes distribution
AxisRep2D, collaborator incidental, 2
AxisRep2D, collaborator factory, 1
*/
/** @file
hippodraw::AxisRep2D class implementation
...
*/
... // constructors here
/** @stereotype collaborator factory */
AxisRepBase * AxisRep2D::clone()
{
    return new AxisRep2D( *this );
}

/** @stereotype collaborator incidental */
void
AxisRep2D::
drawZLabels( const AxisModelBase &,
             ViewBase &, const std::string & )
{
    assert( false );
    // Should never be called.
}

```

```

/** @stereotype collaborator incidental */
void
AxisRep2D::
drawAllZTicks ( const AxisModelBase &,
                const TransformBase &,
                ViewBase & )
{
    // Should never be called;
    assert( false );
}

```

B.13 Data Class - class AxisTick

```

/* Class Report: Method stereotypes distribution
AxisTick, get, 2
AxisTick, set, 2
*/
/* HippoPlot AxisTick class implementation
...
*/
... // constructors here
/** @stereotype get */
double
AxisTick::value ( ) const
{
    return m_v;
}
/** @stereotype set */
void
AxisTick::setValue ( double v )
{
    m_v = v;
}
/** @stereotype get */
const string &
AxisTick::content ( ) const
{
    return m_c;
}
/** @stereotype set */
void
AxisTick::setContent ( const std::string & s )
{
    m_c = s;
}

```

RERERENCES

- [Abbott 1983] Abbott, R., (1983), "Program Design by Informal English Descriptions", *Communications of the ACM*, vol. 26, no. 11.
- [Alali, Kagdi, Maletic 2008] Alali, A., Kagdi, H., and Maletic, J. I., (2008), "What's a Typical Commit? A characterization of Open Source Software Repositories", in *Proceedings of IEEE 16th International Conference on Program Comprehension (ICPC '08)* Amsterdam, The Netherlands June 10-13, pp. 182-191.
- [Albin-Amiot et al. 2001] Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.-G., and Jussien, N., (2001), "Instantiating and detecting design patterns: Putting bits and pieces together ", in *Proceedings of 16th IEEE International Conference on Automated Software Engineering (ASE'2001)*, pp. 166-173.
- [Andriyevska et al. 2005] Andriyevska, O., Dragan, N., Simoes, B., and Maletic, J. I., (2005), "Evaluating UML Class Diagram Layout based on Architectural Importance", in *Proceedings of 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'05)*, Budapest, Hungary September 25, pp. 14-20.
- [Anquetil, Fourier, Lethbridge 1999] Anquetil, N., Fourier, C., and Lethbridge, C. T., (1999), "Experiments with Clustering as a Software Remodularization Method", in *Proceedings of Working Conference on Reverse Engineering*, pp. 235-255.

- [Antoniol, Fiutem, Cristoforetti 1998a] Antoniol, G., Fiutem, R., and Cristoforetti, L., (1998a), "Design Pattern Recovery in Object-Oriented Software", in Proceedings of 6th IEEE International Workshop on Program Comprehension (IWPC'98), Ischia, Italy, June 24-26, pp. 153-160.
- [Antoniol, Fiutem, Cristoforetti 1998b] Antoniol, G., Fiutem, R., and Cristoforetti, L., (1998b), "Using Metrics to Identify Design Patterns in Object-Oriented Software", in Proceedings of 5th IEEE International Symposium on Software Metrics (METRICS'98), Bethesda, MD, November 20-21, pp. 23 - 34.
- [Arevalo, Ducasse, Nierstrasz 2003a] Arevalo, G., Ducasse, S., and Nierstrasz, O., (2003a), "Understanding Classes using X-Ray Views", in Proceedings of 2nd. International Workshop on MASPEGHI 2003 (MANaging SPEcialization/Generalization Hierarchies) in ASE 2003, pp. 9-18.
- [Arevalo, Ducasse, Nierstrasz 2003b] Arevalo, G., Ducasse, S., and Nierstrasz, O., (2003b), " XRay Views: Understanding the Internals of Classes", in Proceedings of 18th IEEE International Conference on Automated Software Engineering, pp. 267-270.
- [Atkinson, Kuhne, Henderson-Sellers 2002] Atkinson, C., Kuhne, T., and Henderson-Sellers, B., (2002), "Stereotypical Encounters of the Third Kind", in Proceedings of UML, pp. 100-114.
- [Aversano et al. 2007] Aversano, L., Canfora, G., Cerulo, L., Grosso, C. D., and Penta, M. D., (2007), "An Empirical Study on the Evolution of Design Patterns", in Proceedings of 6th Joint Meeting of the European Software Engineering

Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Dubrovnik, Croatia, pp. 385-394.

[Basili, Briand, Melo 1996] Basili, V. R., Briand, L. C., and Melo, W. L., (1996), "A Validation of Object-Oriented Design Metrics as Quality Indicators", *IEEE Transactions on Software Engineering*, vol. 22, no. 10, October, pp. 751-761.

[Beck, Cunningham 1989] Beck, K. and Cunningham, W., (1989), "A laboratory for teaching object oriented thinking", in Proceedings of Object-Oriented Programming Systems, Languages and Applications (OOPSLAConference on), pp. 1-6.

[Bennett, Rajlich, 73-87 2000] Bennett, K. H., Rajlich, V., and 73-87, I.-F. o. S. T., (2000), "Software maintenance and evolution: a roadmap", in Proceedings of International Conference on Software Engineering - The Future of Software Engineering Track, pp. 73-87.

[Bieman et al. 2003] Bieman, J., Straw, G., Wang, H., Munger, P. W., and Alexander, R. T., (2003), "Design patterns and change proneness: An examination of five evolving systems", in Proceedings of 9th Ninth International Software Metrics Symposium (Metrics' 2003), Bethesda, MD, November 20-21, pp. 40-49.

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

[Briand, Daly, Wüst 1999] Briand, L. C., Daly, J., and Wüst, J., (1999), "A unified framework for coupling measurement in objectoriented systems", *IEEE Transactions on Software Engineering*, vol. 25, no. 1, January, pp. 91-121.

- [Briand, Daly, Wüst 1997] Briand, L. C., Daly, J. W., and Wüst, J., (1997), "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", in Proceedings of 4th International Software Metrics Symposium (METRICS'97), Albuquerque, NM, November 5-7, pp. 43-53.
- [Brown 1996] Brown, K., (1996), *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk* North Carolina State University, Raleigh NC, Master Thesis.
- [Brown et al. 1998] Brown, W. J., Malveau, R. C., Brown, H. W., McCormick III, W. H., and Mowbray, T. J.,(1998),*Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*, 1st ed., John Wiley and Sons.
- [Bruegge, Dutoit 2000] Bruegge, B. and Dutoit, A.,(2000),*Object-Oriented Software Engineering Conquering Complex and Changing Systems*, Prentice Hall.
- [Chidamber, Kemerer 1991] Chidamber, S. R. and Kemerer, C. F., (1991), "Towards a Metrics Suite for Object Oriented Design", in Proceedings of OOPSLA'91, pp. 197-211.
- [Chidamber, Kemerer 1994] Chidamber, S. R. and Kemerer, C. F., (1994), "A Metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493.
- [Clarke, Malloy, Gibson 2003] Clarke, P. J., Malloy, B. A., and Gibson, J. P., (2003), "Using a Taxonomy Tool to Identify Changes in OO Software", in Proceedings of 7th European Conference on Software Maintenance and Reengineering, pp. 213-222.

- [Collard, Maletic, Marcus 2002] Collard, M. L., Maletic, J. I., and Marcus, A., (2002), "Supporting Document and Data Views of Source Code", in Proceedings of ACM Symposium on Document Engineering (DocEng'02), McLean VA, November 8-9, pp. 34-41.
- [Conallen 2002] Conallen, J.,(2002), *Building Web Applications with UML* 2nd ed., Addison-Wesley.
- [De Lucia et al. 2009] De Lucia, A., Deufemia, V., Gravino, C., and Risi, M., (2009), "Design pattern recovery through visual language parsing and source code analysis", *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177-1193.
- [Deitel, Deitel 2001] Deitel, H. M. and Deitel, P. J.,(2001),*C++ How to Program*, 3rd ed., Prentice Hall.
- [Dekel, Gil 2003] Dekel, U. and Gil, Y., (2003), "Revealing Class Structure with Concept Lattices ", in Proceedings of 10 th Working Conference on Reverse Engineering (WCRE'03), Victoria, Canada, 13–16 November, pp. 353–365.
- [Demeyer, Ducasse, Lanza 1999] Demeyer, S., Ducasse, S., and Lanza, M., (1999), "A Hybrid Reverse Engineering Approach Combining Metrics and Program Vizualization", in Proceedings of Working Conference on Reverse Engineering (WCRE '99), pp. pp. 175-186.
- [Demeyer, Ducasse, Nierstrasz 2000] Demeyer, S., Ducasse, S., and Nierstrasz, O., (2000), "Finding refactorings via Change Metrics", in Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA ' 00), pp. pp.166-177.

- [Dong, Zhao, Sun 2010] Dong, J., Zhao, Y., and Sun, Y., (2010), "Design pattern evolutions in QVT", *Software Quality Journal*, vol. 18, no. 2, pp. 269-297.
- [Dong, Godfrey 2007] Dong, X. and Godfrey, M. W., (2007), "A Hybrid Program Model for Object-Oriented Reverse Engineering", in Proceedings of IEEE International Conference on Program Comprehension, Banff, AB, Canada, pp. 81-90.
- [Dong, Godfrey 2008] Dong, X. and Godfrey, M. W., (2008), "Identifying Architectural Change Patterns in Object-Oriented Systems", in Proceedings of IEEE International Conference on Program Comprehension Amsterdam, The Netherlands, pp. 33-42.
- [Dragan 2005] Dragan, N., (2005), *Method Stereotypes and their Automatic Identification*, Kent State University, Kent, Ohio, Masters Thesis.
- [Dragan, Collard, Maletic 2006] Dragan, N., Collard, M. L., and Maletic, J. I., (2006), "Reverse Engineering Method Stereotypes", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06), Philadelphia, Pennsylvania USA, pp. 24-34.
- [Dragan, Collard, Maletic 2009] Dragan, N., Collard, M. L., and Maletic, J. I., (2009), "Using Method Stereotype Distribution as a Signature Descriptor for Software Systems", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'09), Edmonton, Canada September 20-26, pp. 567-570.
- [Dragan, Collard, Maletic 2010] Dragan, N., Collard, M. L., and Maletic, J. I., (2010), "Automatic Identification of Class Stereotypes", in Proceedings of 26th IEEE

- International Conference on Software Maintenance (ICSM'10), Timisoara, Romania September 12-18, pp. to appear.
- [Eick, Steffen, Summer 1992] Eick, S., Steffen, J. L., and Summer, E. E., (1992), "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", *IEEE Transactions on Software Engineering*, vol. 18, no. 11, November, pp. 957-968.
- [Eisenbarth, Koschke, Simon 2003] Eisenbarth, T., Koschke, R., and Simon, D., (2003), "Locating Features in Source Code", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, March, pp. 210 - 224.
- [Fenton 1991] Fenton, N. E.,(1991),*Software Metrics : A Rigorous Approach*, New York, Chapman & Hall.
- [Fisher 1987] Fisher, H. D., (1987), "Knowledge acquisition via incremental conceptual clustering ", *Machine Learning*, vol. 2, no. 2, pp. 139-172.
- [Fluri, Giger, Gall 2008] Fluri, B., Giger, E., and Gall, H. C., (2008), "Discovering Patterns of Change Types", in Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08), L'Aquila, Italy, pp. 463-466.
- [Fowler 1999] Fowler, M.,(1999),*Refactoring: Improving the Design of Existing Code*, Addison-Wesley.
- [Fowler 2000] Fowler, M.,(2000),*UML Distilled: A Brief Guide to the Standard Object Modelling Language*, 3rd ed., Addison-Wesley.
- [Fuggetta 1993] Fuggetta, A., (1993), "A Classification of CASE Technology", *IEEE Computer*, vol. 26, no. 12, December, pp. 25-38.

- [Gall, Lanza 2006] Gall, H. C. and Lanza, M., (2006), "Software Evolution: Analysis and Visualization", in Proceedings of 28-th International Conference on Software Engineering (ICSE'06), Shanghai, China, pp. 1055-1056.
- [Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J.,(1995),*Design Patterns*, Addison-Wesley.
- [Genero et al. 2008] Genero, M., Cruz-Lemus, J. A., Caivano, D., Abrahão, S. M., Insfrán, E., and Carsí, J. A., (2008), "Does the use of stereotypes improve the comprehension of UML sequence diagrams?" in Proceedings of 2nd International Symposium on Empirical Software Engineering and Measurement (ESEM'08), Kaiserslautern, Germany, October 9-10, pp. 300-302.
- [Gil, Maman 2005] Gil, J. and Maman, I., (2005), "Micro Patterns in Java Code", in Proceedings of Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'05), San-Diego, California USA.
- [Gîrba et al. 2007] Gîrba, T., Ducasse, S., Kuhn, A., Marinescu, R., and Daniel, R., (2007), " Using concept analysis to detect co-change patterns"", in Proceedings of 9th International Workshop on Principles of Software Evolution (IWPSE'07), Dubrovnik, Croatia, pp. 83-89.
- [Gogolla, Henderson-Sellers 2002] Gogolla, M. and Henderson-Sellers, B., (2002), "Analysis of UML Stereotypes within the UML Metamodel", in Proceedings of UML, pp. 84-99.
- [Greevy, Ducasse 2005] Greevy, O. and Ducasse, S., (2005), "Characterizing the Functional Roles of Classes and Methods by Analyzing Feature Traces ", in

Proceedings of 6th International Workshop on Object-Oriented Reengineering (WOOR'05).

[Guéhéneuc, Guyomarc'h, Sahraoui 2010] Guéhéneuc, Y.-G., Guyomarc'h, J.-Y., and Sahraoui, H., (2010), "Improving design-pattern identification: a new approach and an exploratory study", *Software Quality Journal*, vol. 18, pp. 145-174.

[Guéhéneuc, Sahraoui, Zaidi 2004] Guéhéneuc, Y.-G., Sahraoui, H., and Zaidi, F., (2004), "Fingerprinting design patterns ", in Proceedings of 11th Working Conference on Reverse Engineering (WCRE'04), pp. 172-181.

[Hammad, Collard, Maletic 2009] Hammad, M., Collard, M. L., and Maletic, J. I., (2009), "Automatically Identifying Changes that Impact Code-to-Design Traceability", in Proceedings of 17th IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada, May 17-19, pp. 20-29.

[Hammad, Collard, Maletic 2010] Hammad, M., Collard, M. L., and Maletic, J. I., (2010), "Automatically Identifying Changes that Impact Code-to-Design Traceability During Evolution ", *Journal of Software Quality* vol. 18, no. , to appear, accepted for publication April, 2010.

[Hattori, Lanza 2008] Hattori, L. P. and Lanza, M., (2008), "On the Nature of Commits", in Proceedings of 4th International ERCIM Workshop on Software Evolution and Evolvability (EVOL'08), pp. 63 - 71.

[Henderson-Sellers 1996] Henderson-Sellers, B.,(1996),*Software Metrics*, U. K., Prentice Hall.

- [Hindle et al. 2009] Hindle, A. J., German, D. M., Godfrey, M. W., and Holt, R. C., (2009), "Automatic Classification of Large Changes into Maintenance Categories", in Proceedings of IEEE International Conference on Program Comprehension (ICPC'09), Vancouver, Canada 17-19 May 2009.
- [HippoDraw] HippoDraw, "HippoDraw MainPage", Date Accessed: 08/02, <http://www.slac.stanford.edu/grp/ek/hippodraw/index.html>.
- [Hitz, Montazeri 1995] Hitz, M. and Montazeri, B., (1995), "Measuring Coupling and Cohesion in Object-Oriented Systems", in Proceedings of International Symposium on Applied Corporate Computing, Monterrey, Mexico, October.
- [HotDraw 1999] HotDraw, (1999), "HotDraw HomePage", Date Accessed: 08/02, <http://st-www.cs.uiuc.edu/users/brant/HotDraw/HotDraw.html>.
- [Hutchens, Basili 1985] Hutchens, H. D. and Basili, R. V., (1985), "System Structure Analysis: Clustering with Data Bindings ", *IEEE Transactions on Software Engineering*, vol. 11, no. 8, pp. 749-757.
- [Kagdi, Poshyvanyk 2009] Kagdi, H. and Poshyvanyk, D., (2009), "Who Can Help Me with this Change Request?" in Proceedings of IEEE 17th International Conference on Program Comprehension (ICPC '09) Vancouver, BC May 17-19, pp. 273-277.
- [Kim, Pan, Whitehead 2006] Kim, S., Pan, K., and Whitehead, E. J. J., (2006), "Micro Pattern Evolution", in Proceedings of International Workshop on Mining Software Repositories (MSR '06) Shanghai, China.

- [Kim, Whitehead, Bevan 2006] Kim, S., Whitehead, E. J. J., and Bevan, J., (2006), "Properties of Signature Change Patterns", in Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM'06) Philadelphia, Pennsylvania USA, 24-27 Sept. 2006, pp. 4-13.
- [Kim, Whitehead Jr. 2008] Kim, S. and Whitehead Jr., E. J., (2008), "Classifying Software Changes: Clean or Buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181-196.
- [Koschke, Eisenbarth 2000] Koschke, R. and Eisenbarth, T., (2000), "A Framework for Experimental Evaluation of Clustering Techniques", in Proceedings of 8th International Workshop on Program Comprehension (IWPC'00), Limerick, Ireland, June 2000, pp. 201-210.
- [Kuzniarz, Staron, Wohlin 2004] Kuzniarz, L., Staron, M., and Wohlin, C., (2004), "An empirical study on using stereotypes to improve understanding of UML models", in Proceedings of 12th IEEE International Workshop on Program Comprehension, Bari, Italy, pp. 14.
- [Lanza 1999] Lanza, M., (1999), *Combining Metrics and Graphs for Object-Oriented Reverse Engineering*, University of Berne, Switzerland, M.S. Thesis Thesis.
- [Lanza 2003] Lanza, M., (2003), *Object-Oriented Reverse Engineering - Coarse-grained, Fine-grained, and Evolutionary Software Visualization*, University of Berne, Switzerland, PhD. Dissertation Thesis.
- [Lanza, Ducasse 2001a] Lanza, M. and Ducasse, S., (2001a), "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), pp. pp. 300-311.

[Lanza, Ducasse 2001b] Lanza, M. and Ducasse, S., (2001b), "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), pp. 300-311.

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

[Li, Henry 1993] Li, W. and Henry, S., (1993), "Object-Oriented Metrics that Predict Maintainability", *Systems and Software*, vol. 23, no. 2, pp. pp. 111-122.

[Lorenz, Kidd 1994] Lorenz, M. and Kidd, J.,(1994),*Object-Oriented Software Metrics: A Practical Approach*, Prentice-Hall.

[Maletic, Collard 2004] Maletic, J. I. and Collard, M. L., (2004), "Supporting Source Code Difference Analysis", in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11-17, pp. 210-219.

[Mancoridis et al. 1999] Mancoridis, S., Mitchell, B. S., Chen, Y., and Gansner, E. R., (1999), "Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures", in Proceedings of IEEE International Conference on Software Maintenance, Oxford, England, August 30 - September 03, pp. 50-62.

- [Mäntylä 2003] Mäntylä, M., (2003), *Bad smells in software - a taxonomy and an empirical study*, Helsinki University of Technology, PhD Dissertation Thesis.
- [Maqbool, Babri 2007] Maqbool, O. and Babri, A. H., (2007), "Hierarchical Clustering for Software Architecture Recovery", *IEEE Transactions on Software Engineering*, vol. 33, no. 11, November 2007, pp. 759-780.
- [Marinescu 2004] Marinescu, R., (2004), "Detection strategies: Metrics-based rules for detecting design flaws ", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), pp. 350–359.
- [Martin 2002] Martin, R. C.,(2002),*Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall.
- [Mayrhauser, Vans 1995] Mayrhauser, A. and Vans, M. A., (1995), "Program Comprehension During Software Maintenance and Evolution", *Computer*, vol. Vol. 28 , No. 8, pp. 44-55.
- [Mitchell, Mancoridis 2006] Mitchell, S. B. and Mancoridis, S., (2006), "On the Automatic Modularization of Software Systems Using the Bunch Tool", *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193-208.
- [Moha et al. 2008] Moha, N., Guéhéneuc, Y.-G., Le Meur, A.-F., and Duchien, L., (2008), "A domain analysis to specify design defects and generate detection algorithms ", in Proceedings of 11th International Conference on Fundamental Approaches to Software Engineering (FACE'2008).

- [Montes de Oca, Carver 1998] Montes de Oca, C. and Carver, L. D., (1998), " Identification of Data Cohesive Subsystems Using Data Mining Techniques", in Proceedings of International Conference on Software Maintenance, pp. 16-23.
- [Muller 1986] Muller, H. A., (1986), *Rigi - A Model for Software system Construction, Integration,, and Evaluation based on Module Interface specifications*, Rice University, PhD Dissertation Thesis.
- [Munro 2005] Munro, M. J., (2005), "Product Metrics for Automatic Identification of “Bad Smell” Design Problems in Java Source-Code", in Proceedings of 11th International Software Metrics Symposium (METRICS 2005), pp. 15.
- [Ng, Guéhéneuc 2007] Ng, K.-Y. and Guéhéneuc, Y.-G., (2007), "Identification of behavioral and creational design patterns through dynamic analysis", in Proceedings of 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'2007), pp. 34-42.
- [Pelleg, Moore 2000] Pelleg, D. and Moore, A. W., (2000), " X-means: Extending K-means with efficient estimation of the number of clusters", in Proceedings of 17th International Conference on Machine Learning, San Francisco, CA, pp. 727-734.
- [Poshyvanyk, Marcus 2007] Poshyvanyk, D. and Marcus, A., (2007), "Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code ", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC 2007), Banff, Canada, June 26-29, pp. 37-48

- [Purushothaman, Perry 2005] Purushothaman, R. and Perry, D. E., (2005), "Toward understanding the rhetoric of small source code changes", *Transactions on Software Engineering* vol. 31, no. 6, pp. 511-526.
- [Raghavan et al. 2004] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V., (2004), "Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases", in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04), Chicago, Illinois, September 11 - 14, pp. 188-197.
- [Ricca et al. 2010] Ricca, F., Di Penta, M., Torchiano, M., Tonella, P., and Ceccato, M., (2010), "Influence Web Application Comprehension Tasks Supported by UML Stereotypes: A Series of Four Experiments", *IEEE Transactions on Software Engineering*, vol. 36, no. 1, pp. 96-118.
- [Richner, Ducasse 2002] Richner, T. and Ducasse, S., (2002), "Using Dynamic Information for the Iterative Recovery of Collaborations and Roles ", in Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02), Montreal, Canada, October 3-6, pp. 34-43.
- [Riehle, Berczuk 2001] Riehle, D. and Berczuk, S., (2001), "Types of Member Functions in C++", <http://www.riehle.org/computer-science/industry/publications.html>.
- [Riel 1996] Riel, A. J.,(1996),*Object-Oriented Design Heuristics*, Addison-Wesley.
- [Robbes, Ducasse, Lanza 2005] Robbes, R., Ducasse, S., and Lanza, M., (2005), "Microprints: A pixelbased semantically rich visualization of methods", in

Proceedings of ESUG 2005 (13th International Smalltalk Conference - Academic Track) pp. 172 - 188.

- [Robillard 2005] Robillard, M. P., (2005), "Automatic generation of suggestions for program investigation", *ACM SIGSOFT Software Engineering Notes*, vol. 38, no. 5, May, pp. 11-20.
- [Rosenberg, Scott 1999] Rosenberg, D. and Scott, K.,(1999),*Use Case Driven Object Modeling with UML: A practical Approach*, Addison-Wesley.
- [Salvitch 1999] Salvitch, W.,(1999),*Problem Solving with C++: The Object of Programming*, 2nd ed., Addison-Wesley.
- [Sharif, Maletic 2009] Sharif, B. and Maletic, J. I., (2009), "The Effect of Layout on the Comprehension of UML Class Diagrams: A Controlled Experiment", in Proceedings of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'09), Edmonton, Canada September 25, pp. 11-18.
- [Shearer, Collard 2007] Shearer, D. and Collard, M. L., (2007), "Enforcing Constraints Between Documentary Comments and Source Code ", in Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC'07), Banff, Alberta, Canada, pp. 271-276.
- [Siff, Reps 1999] Siff, M. and Reps, T. W., (1999), "Identifying Modules via Concept Analysis ", *IEEE Transactions on Software Engineering*, vol. 25, no. 6, pp. 749-768.

- [Staron, Kuzniarz, Wohlin 2006] Staron, M., Kuzniarz, L., and Wohlin, C., (2006), "Empirical assessment of using stereotypes to improve comprehension of UML models: A set of experiments", *Journal of Systems and Software*, vol. 79, no. 5, pp. 727-742.
- [Storey, Muller 1995] Storey, M.-A. D. and Muller, H. A., (1995), "Manipulating and documenting software structures using SHriMP views", in Proceedings of 11th International Conference on Software Maintenance (ICSM'95), pp. 275.
- [Stroustrup 2000] Stroustrup, B.,(2000),*The C++ Programming Language*, Addison-Wesley.
- [Tonella 2001] Tonella, P., (2001), "Concept Analysis for Module Restructuring", *IEEE Transactions on Software Engineering*, vol. 27, no. 4, July, pp. 351-363.
- [Tonella 2003] Tonella, P., (2003), "Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis", *Transactions on Software Engineering*, vol. 29, no. 6, June 2003, pp. 495-509.
- [Tremblay, Cheston 2001] Tremblay, J.-P. and Cheston, G. A.,(2001),*Data Structures and Software Development in an Object-Oriented Domain*, Prentice Hall.
- [Tzerpos, Holt 1999] Tzerpos, V. and Holt, R. C., (1999), "MoJo: A Distance Metric for Software Clustering", in Proceedings of WCRE'99, Atlanta, October.
- [Vaucher, Sahraoui, Vaucher 2008] Vaucher, S., Sahraoui, H., and Vaucher, J., (2008), "Discovering New Change Patterns in Object-Oriented Systems", in Proceedings of 15th Working Conference on Reverse Engineering (WCRE'08), pp. 37-41.

- [Weiss 1999] Weiss, M. A.,(1999),*Data Structures & Algorithm Analysis in C++*, Addison-Wesley.
- [Wirfs-Brock 1993] Wirfs-Brock, R., (1993), "Stereotyping: a technique for characterizing objects and their interactions", *Object Magazine*, vol. 3, no. 4, pp. 50-53.
- [Wirfs-Brock, B., Wiener 1994] Wirfs-Brock, R., B., W., and Wiener, L., (1994), "Responsibility-Driven Design: Adding to Your Conceptual Toolkit", *ROAD*, vol. 2, pp. 27-34.
- [Workman 2002] Workman, D., (2002), "A Class and Method Taxonomy for Object-Oriented Programs", *Software Engineering Notes*, vol. 27, no. 2, pp. 53-58.
- [Yusuf, Kagdi, Maletic 2007a] Yusuf, S., Kagdi, H., and Maletic, J. I., (2007a), "Assessing the Comprehension of UML Class Diagrams via Eye Tracking", in *Proceedings of IEEE International Conference on Program Comprehension*, Banff, Alberta, Canada, pp. 113-122.
- [Yusuf, Kagdi, Maletic 2007b] Yusuf, S., Kagdi, H., and Maletic, J. I., (2007b), "Assessing the Comprehension of UML Diagrams via Eye Tracking ", in *Proceedings of 15th IEEE International Conference on Program Comprehension (ICPC 2007)*, Banff, Canada, June 26-29, pp. 113-122. .
- [Zaidman, Demeyer 2008] Zaidman, A. and Demeyer, S., (2008), "Automatic Identification of Key Classes in a Software System Using Webmining Techniques", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 6, pp. 387-417